**嵌入式学习笔记**

胡文耀

The true light is not without ever having dark time,just never been just darkness masking.

The real hero is not without ever having the humble sentiment,but never be humble sentiment that yield.

So before you have to defeat an enemy alien,you come to overcome the internal enemy ,you don’t have to be afraid of falling into ,as long as you can pull and update ceaselessly.

# 目录

[目录 2](#_Toc9549)

[第一章 C语言基础编程 3](#_Toc13831)

[1.1 C语言编译过程 3](#_Toc31800)

[1.2 指针和数组 4](#_Toc9845)

[1.3 数据结构 4](#_Toc16261)

[1.4 文件IO读写操作 4](#_Toc17007)

[1.4.1 Linux 文件概述 4](#_Toc20236)

[1.4.2 标准IO函数编程 4](#_Toc10772)

[1.4.3 文件IO编程 7](#_Toc5012)

[第二章 Linux学习 10](#_Toc12221)

[2.1 Linux常用命令介绍 10](#_Toc14325)

[2.2 makefile工程管理 12](#_Toc5414)

[2.3 动态库与静态库制作 14](#_Toc14392)

[2.4 正则表达式 14](#_Toc2694)

[2.5 Shell 编程 14](#_Toc24256)

[第三章 多进程与多线程编程 14](#_Toc18322)

[第四章 网络编程 14](#_Toc18378)

[第五章 Linux开发环境搭建 14](#_Toc5516)

[5.1 VMWare网络配置 14](#_Toc1466)

[5.2 SAMBA服务器配置 17](#_Toc6853)

[5.3 TFTP服务器配置 18](#_Toc5397)

[5.4 NFS服务器配置 18](#_Toc31551)

[5.5 SSH服务器配置 19](#_Toc1311)

# 第一章 C语言基础编程

## gcc编译器1.1 C语言编译过程

常见的编译错误：1 段错误 2 逻辑性错误 3 随机性错误

GDB调试：

1. gcc -g test.c -o test
2. gdb test
3. 命令：

* **l 1,20**:显示第1行至20行
* **b 18**:18行设置为断点
* **r**:运行
* **n**:单步运行，不进入函数体
* **s**:单步运行，进入函数体
* **c**:继续运行
* **info b**:查看断电信息
* **d 2**:删除断点2
* **p i**:查看变量i的值
* **q**:退出

## 1.2 指针和数组

## 1.3 数据结构

常用的数据结构有列表，链表

## 1.4 文件IO读写操作

### 1.4.1 Linux 文件概述

1. 文件分类

|  |  |  |
| --- | --- | --- |
| - |  | 普通文件 |
| b |  | 块设备文件 |
| c |  | 字符设备文件 |
| d |  | 目录文件 |
| l |  | 链接文件 |
| p |  | 管道文件 |
| s |  | 套接字 |

1. 文件系统：

* 磁盘文件系统
* 网络文件系统
* 虚拟文件系统

1. 标准IO三种缓冲（标准输入：stdin；标准输出：stdout；标准错误输出：stderr）

* 全缓冲 ：缓冲区满； 用fflush刷新；程序结束）
* 行缓冲 ：（遇到’\n’；用fflush刷新；结束进程； 利用printf可以输出）
* 无缓冲 ： 标准错误输出

### 1.4.2 标准IO函数编程

分别用IO输入输出函数进行说明

#### 1.4.2.1 文件的打开与关闭

|  |  |
| --- | --- |
| 所需头文件 | #include<stdio.h> |
| 函数原型 | FILE \*fopen(const char \*path,const char \*mode) |
| 函数参数 | path:路径；mode:打开方式 |
| 函数返回值 | 成功：指向File的文件指针 |
| 失败：NULL |

Mode取值说明

|  |  |
| --- | --- |
| r或rb | 打开只读文件，该文件必须存在 |
| r+或r+b | 打开可读写文件，该文件必须存在 |
| w或wb | 打开只写文件，擦除原有文件，若文件不存在则创建 |
| w+或w+b | 打开可读写文件，擦除原有文件，若文件不存在则创建 |
| a或ab | 以附加方式打开只写文件，数据追加，若文件不存在则创建 |
| a+或a+b | 以附加方式打开可读写文件，数据追加，若文件不存在则创建 |
|  |  |

* fclose(FILE \*stream) 成功：0 失败：EOF (#include<stdio.h>)
* perror(const char \*s) s标准错误输出信息 (#include<stdio.h>)

#### **1.4.2.2 字符输入输出函数**

|  |  |
| --- | --- |
| 所需头文件 | #include<stdio.h> |
| 函数原型 | int getc(FILE \*stream);  int getchar(void); (从stdin读取）  int fgetc(FILE\* stream); |
| 函数参数 | stream:流文件 |
| 函数返回值 | 成功：读取的字符 ; 失败：EOF |

|  |  |
| --- | --- |
| 所需头文件 | #include<stdio.h> |
| 函数原型 | int puts(char c,FILE \*stream);  int putchar(char c); (输出至stdout)  int fput(char c,FILE \*stream); |
| 函数参数 | stream:流文件 |
| 函数返回值 | 成功：输出的字符 ；失败：EOF |

#### 1.4.2.3 字符串（行）输入输出函数

**一行行读取输入数据**

|  |  |
| --- | --- |
| 所需头文件 | #include<stdio.h> |
| 函数原型 | char \*gets(char \*s);  char \*fgets(char \*s,int size,FILE \*stream); |
| 函数参数 | s:字符缓冲区首地址 ； size:输入字符长度 |
| 函数返回值 | 成功 ：s ; 失败或到达文件末尾：NULL |

|  |  |
| --- | --- |
| 所需头文件 | #include<stdio.h> |
| 函数原型 | int puts(const char \*s);  int fputs(const char\*s); |
| 函数参数 | s:存放字符串的缓冲区地址首地址 |
| 函数返回值 | 成功 ：s ; 失败：NULL |

#### **1.4.2.3 以指定大小为单位读写文件fread(),fwrite().**

|  |  |
| --- | --- |
| 所需头文件 | #include<stdio.h> |
| 函数原型 | size\_t fread(void\*ptr,size\_t size,size\_t nmemb,FILE\* stream);  size\_t fwrite(void\*ptr,size\_t size,size\_t nmemb,FILE\* stream); |
| 函数参数 | ptr:缓冲区 (char buf[BUFSIZ];)  size：每个记录的大小  nmemb:写入的记录数 （一般为 1）  stream: 流文件 |
| 函数返回值 | 成功：nmemb数目；失败：EOF |

#### 1.4.2.5 文件流定位

fseek()函数

|  |  |
| --- | --- |
| 所需头文件 | #include<stdio.h> |
| 函数原型 | int fseek(FILE\*stream,long offset,int whence); |
| 函数参数 | stream:需要定位的流文件； |
| offset：相对于基准值的偏移量 |
| whence:基准值  SEEK\_SET:文件开头  SEEK\_END:文件末尾  SEEK\_CUR:当前读写位置 |
| 函数返回值 | 成功：0 ；失败：EOF |

ftell()函数

|  |  |
| --- | --- |
| 所需要头文件 | #include<stdio.h> |
| 函数原型 | long ftell(FILE\* stream) |
| 函数参数 | stream :需要定位的流文件 |
| 函数返回值 | 成功：返回当前的读写位置 ；失败：EOF |

Example:

....

fseek(fp,0,SEEK\_END);

printf(“the size of %s is %ld\n”,argv[0],ftell(fp)); //当前文件指针移动过位置

运行结果：the size of txt is 305

#### 1.4.2.6 格式化输入输出函数

|  |  |
| --- | --- |
| 所需头文件 | #include<stdio.h> |
| 函数原型 | int scanf(const char \*format,...); (默认stdin)  int fscanf(FILE\*fp,const char \*format,...);  int sscanf(char\*buf,const char \*format,...); |
| 函数参数 | format：输入的格式  fp:作为输入的流  buf:作为输入的缓冲区 |
| 函数返回值 | 成功：输出字符数  失败：EOF |

scanf(“%c”,&ch); 若输入数字，则返回值为-1;

|  |  |
| --- | --- |
| 所需头文件 | #include<stdio.h> |
| 函数原型 | int printf(const char\*format,...);  int fprintf(FILE\*fp,const char\*format,...);  int sprintf(FILE\*fp,const char\*format,...); |
| 函数参数 | format:输出的格式  fp:作为输出的流  buf:作为输出的缓冲区 |
| 函数返回值 | 成功：输出的字符数  失败：EOF |

### 1.4.3 文件IO编程

Linux的文件系统由两层结构构建：第一层是虚拟文件系统（VFS),第二层是各种不同的具体的文件系统

应用程序

用户态

系统调用(open,read,write)

内核态

内存文件系统

网络文件系统

设备（硬盘，闪存等）

VFS

#### 1.4.3.1 文件的打开与关闭

* open()函数

|  |  |  |
| --- | --- | --- |
| 所需头文件 | #include<sys/stat.h>  #include<fcntl.h> | |
| 函数原型 | int open(const char\*pathname,int flags,int perms); | |
| 函数参数 | pathname:被打开的文件名(包括路径); | |
| flags | O\_RDONLY:只读  O\_WRONLY:只写  O\_RDWR:读写  O\_CREAT:若文件不存在则创建  O\_EXCL:测试文件是否存在  O\_NOCTTY:若打开的是终端文件，则该终端不会成为当前进程的控制终端。  O\_TRUNC:若文件存在，则删除文件全部内容，文件大小为0；  O\_APPEND:以添加的方式打开文件 |
| perms: 新建文件的存取权限 | |
| 函数返回值 | 成功：返回文件描述符  失败：-1 | |

* close()函数

|  |  |
| --- | --- |
| 所需头文件 | #include<unistd.h> |
| 函数原型 | int close(int fd); |
| 函数参数 | fd:文件描述符 |
| 函数返回值 | 成功：0 ；失败：-1 |

#### 1.4.3.2 文件的读写操作

* read()函数

|  |  |
| --- | --- |
| 所需头文件 | #include<unistd.h> |
| 函数原型 | ssize\_t read(int fd,void\*buf,size\_t count); |
| 函数参数 | fd:文件描述符  buf：文件缓冲区  count: 读出的数据字节数 |
| 函数返回值 | 成功：读到的字节数  0：文件末尾  -1：错误 |

* write()函数

|  |  |
| --- | --- |
| 所需头文件 | #include<unistd.h> |
| 函数原型 | ssize\_t write(int fd,void\*buf,size\_t count); |
| 函数参数 | fd:文件描述符  buf:文件缓冲区  count：写入数据字节数 |
| 函数返回值 | 成功：写入的字节数  -1：错误 |

#### 1.4.3.3 文件定位

* lseek()函数

|  |  |
| --- | --- |
| 所需头文件 | #include<unistd.h>  #include<sys/types.h> |
| 函数原型 | off\_t lseek(int fd, off\_t offset, int whence) |
| 函数参数 | offset:相当于基准点whence的偏移量。以字节为单位，正数表示向前移动，负数表示向后移动。 |
| whence:  SEEK\_SET:文件开头  SEEK\_CUR:文件当前位置  SEEK\_END:文件末尾 |
| 函数返回值 | 成功：文件当前读写位置  失败：-1 |

#### 1.4.3.4 文件锁

* fcntl()函数

|  |  |  |
| --- | --- | --- |
| 所需头文件 | #include<sys/types.h>  #include<unistd.h>  #include<fcntl.h> | |
| 函数原型 | int fcntl(int fd, int cmd,...); | |
| 函数参数 | cmd | F\_GETLK:检测文件锁状态 |
| F+SETLK:设置lock描述的文件锁 |
| F\_SETLKW:阻塞 |
| 函数返回值 | 成功：0  错误：-1 | |

如果cmd和锁操作相关，则第三个参数的类型为struct \*flock,定义如下：

struct flock{

...

short l\_type;

off\_t l\_start;

short l\_whence;

off\_t l\_len;

pid\_t l\_pid;

...

}

flock 结构成员含义：

|  |  |
| --- | --- |
| l\_type | F\_RDLCK:读取锁（共享锁） |
| F\_WRLCK:写入锁（排斥锁） |
| F\_UNLCK:解锁 |
| l\_start | 加锁区域在文件中的相对位移量（字节），与l\_whence决定加锁区域起始。 |
| l\_whence | SEEK\_SET:当前位置为文件开头 |
| SEEK\_CUR:当前位置为文件指针的位置 |
| SEEK\_END：当前位置为文件的结尾 |
| l\_len | 加锁区域的长度 |
| l\_pid | 具有阻塞当前进程的锁，进程号存放l\_pid中，仅由F\_GETLK返回 |

若要加锁整个文件，可以将l\_start=0,l\_whence=SEEK\_SET,l\_len=0.

# 第二章 Linux学习

Linux是一款免费开源的操作系统。

## 2.1 Linux常用命令介绍

1. ls 命令：

|  |  |
| --- | --- |
| ls -a | 显示所有文件，包括隐藏文件 |
| ls -l | 显示文件属性 |

1. cp 命令：

|  |  |
| --- | --- |
| cp -av source\_dir target\_dir | 将整个目录复制，两目录完全一致 |
| cp -fr source\_dir target\_dir | 将整个目录复制，并且是非链接方式复制 |

1. diff 命令:

|  |  |
| --- | --- |
| Diff -ruN dir1 dir2 > <patch\_file> | 比较目录1和目录2中的文件 |
|  |  |

1. Du命令：

|  |  |
| --- | --- |
| du | 计算当前目录的容量 |
| du -sm /root | 计算/root 目录的容量并以M为单位 |

1. Find命令：

|  |  |
| --- | --- |
| Find -name <path> file | 在/path目录下查找看是否有文件file |
|  |  |

1. Grep命令：

|  |  |
| --- | --- |
| grep -r “chars” ./ | 在当前目录的所有文件中查找字符串”chars”  -r 表示递归查找子目录 |
|  |  |

1. 启动和关闭计算机命令：

|  |  |
| --- | --- |
| reboot | 重新启动计算机 |
| halt | 关闭计算机 |
| init 0 | 关闭所有的应用程序和服务，进入纯净的操作系统环境 |
| init 1 | 重新启动应用及服务 |
| init 6 | 重新启动计算机 |

1. 压缩和解压命令：

|  |  |
| --- | --- |
| tar -xfzv file.tgz | 将文件file.tgz解压 |
| tar -zcvf file.tgz <source> | 将文件或者目录<source>压缩为file.tgz |
| gzip directory.tar | 将覆盖源文件生产压缩的directory.tar.gz |
| gunzip directory.tar.gz | 覆盖源文件解压生成不压缩的directory.tar |

1. dmesg,uname命令：

|  |  |
| --- | --- |
| dmesg | 显示kernel启动及驱动装载信息 |
| uname -a | 显示操作系统的类型 |
| getconf LONG\_BIT | 显示操作系统位数 |

1. 文件目录管理命令：

|  |  |
| --- | --- |
| Su root | 切换到超级用户 |
| chmod 666 file | 将文件file设置为可读写 |
| chmod a+x file | 将file文件设置为可执行，脚本类文件一定要这样设置一个，否则得用bash file 才能执行 |
| chown user /dir | 将/dir目录设置为user所有 |

1. 登录网络服务器：

|  |  |
| --- | --- |
| telnet 192.168.1.1 | 登录IP为192.168.1.1的Telnet服务器 |
| ftp 192.168.1.1 | 登录到FTP服务器 |
| sudo ifconfig eth0 up | 重启网卡 |
| sudo apt-get install ufw  sudo ufw enable  sudo ufw disable  sudo ufw status | 安装ufw  开启防火墙  关闭防火墙  查看防火墙信息 |

1. 系统有关的命令：

|  |  |
| --- | --- |
| ps | 显示当前系统进程信息 |
| ps -ef | 显示系统所有进程信息 |
| kill -9 500 | 将进程编号为500的程序杀死 |
| top | 显示系统进程的活动情况，按占CPU资源百分比来比分 |
| free | 显示系统内存及swap使用情况 |
| time program | 在program程序结束后，将计算出program运行使用的时间 |
| ifconfig eth0 192.168.1.2 | 捆绑网卡1的IP地址为192.168.1.2 |

1. 磁盘管理的命令

|  |  |
| --- | --- |
| fdisk /dev/hda | 就像执行了dos的fdisk一样 |
| mount -t ext2 /dev/hda1 /mnt | 把/dev/hda1装载到/mnt目录 |
| mount -t nfs 192.168.1.1:/sharedir /mnt | 将nfs服务的共享目录sharedir加载到/mnt/nfs目录 |
| umount /mnt | 将/mnt目录卸载，/mnt目录必须处于空闲状态 |
| sync | 刷新缓冲区，使内容与磁盘同步 |
| mkfs.ext2 /dev/hda1 | 格式化/dev/hda1 为ext2格式 |
| dd if=/dev/zero of=root.ram bs=1024,count=1024 | 生成一个大小为1M的块设备，可以把它当做硬盘的一个分区使用 |
| mknod /dev/hda1 b 3 1 | 创建块设备hda1，主设备号为3，从设备号1，即master硬盘的第一个分区 |
| mknod /dev/tty1 c 4 1 | 创建字符设备tty1，主设备号为4，从设备号为1，即第一个tty终端 |
| df | 显示文件系统装载的相关信息 |

## 2.2 makefile工程管理

Make 只编译改动的文件，不用全部编译。

1. **递归定义展开**：

aa=$(bb)

bb=$(cc)

cc=$(dd)

dd=hello world

echo:

@echo$(dd)

@echo$(cc)

@echo$(bb)

@echo$(aa)

1. **简单定义**：

aa:=$(bb)

bb:=$(cc)

cc:=$(dd)

dd:=hello world

echo :

@echo $(aa)

@echo $(bb)

@echo $(cc)

@echo $(dd)

makefile 默认执行makefile中的第一条规则。

1. 自定义变量：

OBJ=main.o show.o sum.o

main:$(OBJ)

gcc $(OBJ) -o main

1. 预定义变量：

CC=gcc

CFLAGS=-g -Wall

OBJ=main.o show.o sum.o

main:$(OBJ)

$(CC) $(CFLAGS) $(OBJ) -o main

main.o:main.c

$(CC) $(CFLAGS) -c main.c -o main.o

show.o:show.c

$(CC) $(CFLAGS) -c show.c -o show.o

sum.o:sum.c

$(CC) $(CFLAGS) -c sum.c -o sum.c

(上面等效于）

%.o:%.c

$(CC) $(CLFAGS) -c %.c -o %.o

clean:

$(RM) $(OBJ) main .\*.sw?

## 2.3 动态库与静态库制作

* 静态库：

gcc -c fun.c -o fun.o

ar crv libfun.a fun.o

gcc main.c -o main -lfun -L.

直接运行：./main

-L 告诉gcc搜索链接库报刊当前路径(-L../io,表示包括上一级路径下io目录)

-lfun 告诉gcc生成可执行程序要链接libfun.a

* 动态库：

gcc -fPIC -Wall -c fun.c (生成fun.o文件)

gcc -Shared -fPIC -o libfun.so (生成fun.so文件)

等效于 gcc fun.o -Shared -fPIC -o libfun.so

gcc main.c -o main -lfun -L. (链接库文件，如果同一目录中同时存在同名的静态库和动态库，则优先链接动态库)

运行: ./main, 前需添加搜索路径：

1. sudo mv libfun.so /lib
2. sudo mv libfun.so /usr/lib

3. sudo vi /etc/ld.so.conf.d/libc.conf (执行时，Linux通过/etc/ld.so.cache文档链接动态库)

## 2.4 正则表达式

## 2.5 Shell 编程

# 多进程与多线程编程

## 3.1 多进程

### 3.11 进程编程

Linux是一个支持多任务的操作系统，多任务操作系统有三个基本概念：任务，进程和线程。

**进程：一个程序的一次执行。**

在Linux中获得当前进程的进程号(PID)-- getpid(),获得父进程号(pid)-- getppid();

Linux内核通过唯一的进程标识符PID来标识每个进程。虚拟内存管理为每个进程分配4GB大小的虚拟空间，4GB的进程地址空间分为两部分：用户空间和内核空间。

用户空间包括以下几个功能区域：

1. 只读段。包括程序代码(.init和.text) 和只读数据(.rodata)。
2. 数据段。存放全局变量和静态变量。其中初始化数据段(.data)存放显示初始化的全局变量和静态变量。未初始化数据段(.bss)存放未初始化的全局和静态变量。
3. 栈。自动分配，存放函数参数值，局部变量值，返回地址等。
4. 堆。动态分配，程序员动态分配和释放。
5. 共享库的内存映射区域。Linux动态链接器和其他共享库代码的映射区域。

Linux系统每一个进程都会有”\proc”与之对应的一个目录，例如进程号13703的一个应用程序，”cat /proc/13703/maps”,可查看该进程的内存映射情况。

线程：线程是一种轻量级的进程，同一进程的线程共享其内存资源和资源。

### 3.1.1 fork()

|  |  |
| --- | --- |
| 所需头文件 | #include<sys/types.h>  #include<unistd.h> |
| 函数原型 | pid\_t fork(void); |
| 函数返回值 | 0: 子进程 |
| 子进程：父进程（大于0） |
| -1：错误 |

getpid() 获得当前进程的进程号。

### 3.1.2 exec函数族

fork()创建的子程序几乎复制了父进程的全部内容，exec用于在一个进程中执行另一个进程的程序的方法。执行完后，当前进程除了进程号外其他堆栈信息全部被替换。可执行文件一般为二进制文件 或者Linux下任何可执行的脚本文件。

Linux中使用exec函数族主要为以下两种情况：

1. 当进程不能为系统或者用户做出任何贡献时，调用exec函数族任意函数重生。
2. 如果进程想执行另一个程序，可以先fork()新建一个进程，然后调用exec函数，产生一个新的进程。

|  |  |
| --- | --- |
| 所需头文件 | #include<unistd.h> |
| 函数原型 | int execl(const char\*path,const char\*arg); |
| int execv(const char \*path,char\*const argv[]); |
| int execle(const char\*path,const char\*arg,...,char\*const envp[]); |
| int execve(const char\*path,char\*const argv[],char\* const envp[]); |
| int execlp(const char\*file,const char\*arg,...); |
| int execvp(const char\*file,char \*const argv[]); |
| 函数返回值 | -1:错误 |

l:参数传递为逐个列举的方式。

v:参数传递为构造指针数组方式。

e:可传递新进程环境变量

p:可执行文件查找方式为文件名

example1:

if(fork()==0)

{

if((ret=execlp(“ps”,”ps”,”-ef”,NULL))<0)

{

printf(“execlp error\n”);

}

}

example2:

if(fork()==0)

{

/\*调用execl()，给出ps程序所在的完整路径\*/

if((ret=execl(“/bin/ps”,”ps”,”-ef”,NULL))<0)

{

printf(“execl error\n”);

}

}

example3:

char \*envp[]={“PATH=/tmp”,”USER=harry”,NULL}; //字符串数组

if(fork()==0)

{

/\*调用execle()函数，注意这里也要指出env的完整路径\*/

if((ret=execle(“/usr/bin/env”, ”env”,NULL,envp))<0)

{

printf(“execle error\n”);

}

}

example4:

char\*arg[]={“env”,NULL};

char\*envp[]={“PATH=/tmp”,”USER=harry”,NULL};

if(fork()==0)

{

if((ret=execve(“/usr/bin/env”,arg,envp))<0)

{

printf(“execve error\n”);

}

}

### 3.1.3 进程处理函数

**exit() 和\_exit()：**

|  |  |
| --- | --- |
| 所需头文件 | exit： #include<stdlib.h> |
| \_exit: #include<unistd.h> |
| 函数原型 | exit: void exit(int status) |
| \_exit: void \_exit(int status) |
| 函数返回值 |  |

**wait()**

|  |  |
| --- | --- |
| 所需头文件 | #include<sys/types.h>  #include<sys/wait.h> |
| 函数原型 | pit\_t wait(int \*status) |
| 函数参数 | status:保存子进程结束时的状态 |
| 函数返回值 | 成功：已回收的子进程的进程号  失败：-1 |

**waitpid():**

|  |  |  |
| --- | --- | --- |
| 所需头文件 | #include<sys/types.h>  #include<sys/wait.> | |
| 函数原型 | pit\_t waitpid(pid\_t pid,int\*status, int options) | |
| 函数参数 | pid>0:回收进程ID等于pid的子进程 | |
| pid=-1:回收任何一个子进程，此时和wait()作用一样 | |
| pid=0:回收其组ID等于调用进程的组ID的任一子进程 | |
| pid<-1:回收其组ID等于pid的绝对值的任一子进程 | |
| status:同上 | |
| options: | WNOHANG:若指定的子进程没有结束，则waitpid()不阻塞，返回值为0 |
| WUNTRACED:指定pid的子进程被暂停 |
| 0:同wait()，阻塞父进程，直至子进程退出 |
| 函数返回值 | >0:已经结束运行的子进程的进程号 | |
| 0:使用 WNOHANG,且没有子进程退出 | |
| -1:错误 | |

### 3.1.4 守护进程

1. 创建子进程

pid=fork();

1. 创建新会话

setsid();

1. 创建子进程

pid=fork();

1. 关闭所有文件描述符

max\_fd=sysconf(\_SC\_OPEN\_MAX)

for(i=0;i<max\_fd,i++)

close(i);

1. 清除掩码

umask(0);

1. 修改当前目录

chdir(“/”);

1. 重定位文件描述符

open(“/dev/null”,O\_RDWR);

dup(0);

dup(0);

## 3.2 进程间的通信

## 3.3 多线程

# 网络编程

## 4.1 网络编程基础

### 4.1.1 IP地址

IP地址：网络号和主机号。

IPV4：192.168.1.1

IPV6：fabc.12fa.3423.a564.b323.c4c3.d321.3a24 (128位，8段，每段16位)

IP地址有两种不同的格式：十进制点分形式（用户）和32位二进制形式（网络传输）。

inet\_addr()和inet\_pton()：十进制点分式字符串—>二进制

inet\_ntop():二进制—>十进制点分式字符串

|  |  |
| --- | --- |
| 所需头文件 | #include<arpa/inet.h> |
| 函数原型 | int inet\_addr(const char \*strptr) |
| 函数参数 | strptr:要转化的IP地址字符串 |
| 函数返回值 | 成功：32位的IP地址（网络字节序）  错误：-1 |

|  |  |  |
| --- | --- | --- |
| 所需头文件 | #include<arpa/inet.h> | |
| 函数原型 | int inet\_pton(int family,const char \*src,void\*dst) | |
| 函数参数 | family: | AF\_INET：IPV4协议 |
| AF\_INET6：IPV6协议 |
| 函数返回值 | 成功：0  错误：-1 | |

|  |  |
| --- | --- |
| 所需头文件 | #include<arpa/inet.h> |
| 函数原型 | int inet\_ntop(int family,void\* src,char \*dst,int len) |
| 函数参数 | family:同上  src:二进制数据  dst:要转化的十进制字符串缓冲区  len:缓冲区长度 |
| 函数返回值 | 成功：返回dst  错误：NULL |

地址结构的相关处理：

struct sockaddr{ //16字节大小

unsigned short sa\_family; //地址簇

char sa\_data[14]; //14字节的协议地址，与下结构大小一样，实现强转.

}

struct sockaddr\_in{

short int sin\_family; //地址簇

unsigned short int sin\_port; //端口号

struct in\_addr sin\_addr; //地址

unsigned char sin\_zero[8];//填充0，保持与sockaddr同等大小

}

这两个结构体大小相等，通常用sockaddr\_in存储一个IP地址的信息，然后强转为sockaddr类型指针进行使用。

结构字段：

|  |  |
| --- | --- |
| 结构定义所在头文件 |  |
| sa\_family | AF\_INET: IPV4协议 |
| AF\_INET6：IPV6协议 |
| AF\_LOCAL:UNIX域协议 |
| AF\_LINK:链路地址协议 |
| AF\_KEY:密钥套接字 |

### 4.1.2 端口

端口是一个无符号整形，取值范围 0~65535

端口号是系统的一种资源，0至1023一般被系统使用。

TCP端口和UDP端口独立，互不影响。

IP可以表示网络中一个主机，端口号可以代表主机内部的一个套接字。

IP地址和端口，进行网络传输时，需进行字节序转换。

字节序分为两种：大端和小端

大端：高位字节存储在低位地址，低位字节存储在高位地址。

小端：和大端相反，通常PC采用小端

|  |  |
| --- | --- |
| 所需头文件 | #include<netinet.h> |
| 函数原型 | uint16\_t htons(unit16\_t hostshort) 端口  uint16\_t stohs(unit16\_t netshort) 端口  uint32\_t htonl(unit16\_t hostlong) IP地址  uint32\_t ntohl(unit32\_t netlong) IP地址 |
| 函数参数 | 16位或者32位数据 |
| 函数返回值 | 成功：返回转换后字节序数值  错误：-1 |

## 4.2 TCP网络编程

socket编程基本函数有:

socket(),bind(),listen(),accept(),connect(),send(),sendto(),recv(),recvfrom().

**socket()函数**

|  |  |  |
| --- | --- | --- |
| 所需头文件 | #include<sys/socket.h> | |
| 函数原型 | int socket(int family,int type,int protocol) | |
| 函数参数 | type | SOCK\_STREAM:流式套接字 TCP |
| SOCK\_DGRAM:数据报套接字 UDP |
| SOCK\_RAW:原始套接字 |
| protocol | 一般为 0 |
| 函数返回值 | 成功： 非负的套接字描述符  错误：-1 | |

## 4.3 JSON传输协议解析

jsonc库使用

作者：wangkangluo1 | 出处：博客园 | 2011/12/15 15:12:28 | 阅读58次

原文地址：

1: 下载json-c库源码文件

json-c-0.9.tar.gz

2: 编译

3: json-c常用函数

(1): 将一个json文件转换成object对象:

struct json\_object\* json\_object\_from\_file(char \*filename)

举例：

json\_object \*pobj = NULL;

pobj = json\_object\_from\_file("/home/boy/tmp/test/jsontest/test.json");

(2): 将json-object写回文件

int json\_object\_to\_file(char \*filename, struct json\_object \*obj)

举例:

json\_object\_from\_file("/home/boy/tmp/test/jsontest/test.json", pobj);

(3): 删除一个对象

void json\_object\_object\_del(struct json\_object\* jso, const char \*key);

(4): 增加一个对象

void json\_object\_object\_add(struct json\_object\* jso, const char \*key, struct json\_object \*val);

举例:

json\_object\_object\_add(pobj, "Name", json\_object\_new\_string("Andy"));

json\_object\_object\_add(pobj, "Age", json\_object\_new\_int(200));

(5): 释放对象

void json\_object\_put(struct json\_object \*jso);

4:　使用举例

(1): 新建一个x.json文件

{

"item1": "I love JSON",

"foo": "You love Json",

"item3": "We love JSON",

"Name": "Andy",

"Age": 28

}

(2): 程序

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

#include <string.h>

#include "json.h"

void GetValByKey(json\_object \* jobj, const char \*sname)

{

json\_object \*pval = NULL;

enum json\_type type;

pval = json\_object\_object\_get(jobj, sname);

if(NULL!=pval){

type = json\_object\_get\_type(pval);

switch(type)

{

case json\_type\_string:

printf("Key:%s value: %s\n", sname, json\_object\_get\_string(pval));

break;

case json\_type\_int:

printf("Key:%s value: %d\n", sname, json\_object\_get\_int(pval));

break;

default:

break;

}

}

}

int main(void)

{

json\_object \*pobj = NULL;

//pobj = json\_tokener\_parse("{ \"abc\": 12, \"foo\": \"bar\", \"bool0\": false, \"bool1\": true, \"arr\": [ 1, 2, 3, null, 5 ] }");

//printf("new\_obj.to\_string()=%s\n", json\_object\_to\_json\_string(pobj));

//json\_parse(pobj);

pobj = json\_object\_from\_file("/home/boy/tmp/test/jsontest/test.json");

GetValByKey(pobj, "foo");

json\_object\_object\_del(pobj, "foo");

json\_object\_object\_add(pobj, "foo", json\_object\_new\_string("fark"));

json\_object\_object\_add(pobj, "Age", json\_object\_new\_int(200));

GetValByKey(pobj, "Age");

json\_object\_to\_file("/home/boy/tmp/test/jsontest/new.json", pobj);

json\_object\_put(pobj);

return 0;

}

Makefile

all:server

server:

gcc -std=c99 -msse2 -Wall -O3 -o main main.c -ljson -lcrypto -lm -lpthread

## 4.4数据库sqlite3使用

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*常用数据类型\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

 typedef struct sqlite3 sqlite3;       //代表一个sqlite3数据库

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 接口API\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

原型：int sqlite3\_open(const char\*, sqlite3\*\*);

说明： 第一个参数：数据库路径    第二个参数：指向数据库的指针

返回值：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 接口API\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

原型：int sqlite3\_close(sqlite3\*);

说明：

返回值：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 接口API\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

原型：const char \*sqlite3\_errmsg(sqlite3\*);

说明：

返回值：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 接口API\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

原型：int sqlite3\_errcode(sqlite3\*);

说明：

返回值：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 接口API\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

原型：int (\*sqlite\_callback)(void\*,int,char\*\*, char\*\*);

说明：  sqlite3\_exec的回调函数，当sqlite3\_exec的sql语句执行结果不为空时调用

参数： 1. sqlite3\_exec函数的第四参数，传递到这。

            2. 查询的表中的列数（每行的数据项个数）

            3. 列内容数组指针（数据项内容数组）

            4. 列名数组指针（数据项名数组）

返回值：错误码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 接口API\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

原型：int sqlite3\_exec(sqlite3\*, const char \*sql, sqlite\_callback, void\*, char\*\*);

说明： 执行sql语句，如果语句有输出则调用回调函数

参数：1. 数据库指针

           2. sql语句

           3. 回调函数

           4. 传递到回调函数的数据

           5. sql语句的执行结果

返回值：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 接口API\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

原型：

说明：

返回值：

======================SQLite语句，简单说明======================

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*创建表\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Create  table  admin(username text, age integer);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*插入数据\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

insert into 表名（字段列表） values(值列表);

例如：insert  into admin values('song',25);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*查询\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

select 字段名 from 表名;

select \* from admin;

select  distinct  field  from  table\_name;(distinct去掉重复项，将列中各字段值单个列出)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*删除数据\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 Delete from 表名 where 条件子句。

 delete from admin form where username='song';

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*修改\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 update 表名 set 字段名=值 where 条件子句。

 update admin set username='zhang',age=24 where username='song' and age=25;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*按条件分组\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

select \* from 表名 where 条件子句 group by 分组子句 having …order by排子句

 例如：

    select \* from admin;

    select \* from admin order by id desc(降序) | asc(升序);

    select username from admin group by username having count(\*)>1;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*多条件查询语句\*\*\*\*\*\*\*\*\*\*\*\*\*

 select 字段名 from 表名 where  子句1 按 子句二

 select \* from admin where username='song' and age=24;

 select \* from table\_name where field in (‘val1' , 'val2' , ‘val3' );

 select \* from table\_name where field between  val1 and val2;

 select \* from admin limit 5;        --限制输出数据记录数量

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*多条件排序\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

select 字段名 from 表名 order by 字段1 (desc)，字段2(desc);

select \* from admin order by t1 ,t2 desc;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*索引\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

例如

建立复合索引：create index  idxT1  on admin(username,age);

各自建立索引：create index  idxUsername on  admin(username);

                       create index  idxAge on admin(age);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*外键FOREIGN KEY（UNIQUE | PRIMARY KEY | NOT NULL）的用法()

     create table a(

a1 INTEGER  PRIMARY KEY | UNIQUE | NOT NULL,

a2 TEXT,

a3 INTEGER );

     create table b()(

        b1 INTEGER ,

        b2 TEXT,

        b3 INTEGER,

foreign key(b3) references a(a1)

);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*分页\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

select \* from account limit 5 offset 3;

或者  select \* from account limit 5,3;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*模糊查询\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 SELECT 字段 FROM 表 WHERE 某字段 LIKE 条件

（1）%：表示任意0个或多个字符

（2）\_ ：表示任意单个字符，匹配单个任意字符，常用来限制表达式的字符长度语句。

（3）[ ]：表示括号内所列字符中的一个（类似正则表达式）

      select \* from admin where username like ‘[张李王]三';

      表示搜索的是“张三”，“李三”或“王三”

  [4]：[^]表示不在括号所列之类的单个字符。

  [5]：查询内容包含通配符时,用“[ ]”括起来。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*删除表 | 索引\*\*\*\*\*\*\*\*\*\*\*\*\*\*

drop table [ IF EXISTS] admin;

drop index index\_name

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*查询记录数目\*\*\*\*\*\*\*\*\*\*\*\*\*

select count(\*) from table\_name;

# 第五章 Linux开发环境搭建

## 5.1 VMWare网络配置

1. 打开

在VMware下安装虚拟ubunt系统后网络的配置：

1、检查VMware中网络的设置：

按下图中设置



2、在命令行配置IP：

1》用ifconfig命令查看ip网络信息：



2》通过修改配置文件对网络进行修改：

1) 打开网络配置文件：

sudo vi /etc/network/interfaces

2)写入以下信息，然后保存退出

auto lo

iface lo inet loopback

auto eth0

iface eth0 inet static

address 192.168.7.x //设置本机IP地址：x的值为每位同学的**WindowsIP+60**

gateway 192.168.7.1

netmask 255.255.255.0

3) 重启网络服务：

sudo /etc/init.d/networking restart

3》检查ip是否配置成功：

1) 先用ifconfig检查一下ip，

2) 用ping命令检查本机与windo系统是否连通：

ping 192.168.7.10 ,如下图：



3、配置DNS(域名解析)服务器

1》打开DNS服务器配置文件：

sudo vi /etc/resolvconf/resolv.conf.d/tail

2》写入以下信息：

nameserver 202.96.128.86

3》重启本地服务：

sudo /etc/init.d/resolvconf restart

4》ping网址，例如：

ping <www.baidu.com> ,如下图：



## 5.2 SAMBA服务器配置

samba服务的安装及配置

一、安装：

sudo apt-get install samba

二、配置：

1、创建一个需要共享的目录，并修改权限：

lpf@ubuntu:~$ mkdir /tftpboot

lpf@ubuntu:~$ sudo chmod 0777 /tftpboot -R

2、打开配置文件：

lpf@ubuntu:~$ sudo vim /etc/samba/smb.conf

3、对创建的共享目录进行配置并保存退出：

[tftpboot]

path = /tftpboot

available = yes

browseable = yes

public = yes

writable = yes

三、重新samba服务：

lpf@ubuntu:~/1504$ sudo /etc/init.d/smbd restart

四、测试：

在windows中点击：

-->开始

--->运行

--->\\linux的IP （例如：\\192.168.7.5）

五、设置samba密码：(暂时先不做)

1》在配置文件 sudo vim /etc/samba/smb.conf中加入以下内容：

[tftpboot]

path = /tftpboot

available = yes

browseable = yes

public = yes

writable = yes

valid users = peter

2》设置samba密码：

sudo smbpasswd -a peter

3》重启samba服务：

sudo /etc/init.d/smbd restart

## 5.3 TFTP服务器配置

tftp服务器的安装与配置：

tftp主要用于嵌入式交叉开发环境的搭建，传输文件。

0、创建tftp的工作目录，并修改权限

sudo mkdir /tftpboot

sudo chmod 0777 /tftpboot

1、安装：

sudo apt-get install tftp tftpd openbsd-inetd

2、打开配置文件 /etc/inetd.conf ：

sudo vim /etc/inetd.conf

3、添加以下内容：

tftp dgram udp wait nobody /usr/sbin/tcpd /usr/sbin/in.tftpd /tftpboot

4、打开文件 /etc/default/tftpd-hpa ，

sudo vim /etc/default/tftpd-hpa

添加以下内容：

RUN\_DAEMON="yes"

OPTIONS = "-l -s /tftpboot"

5、重启服务：

sudo /etc/init.d/openbsd-inetd restart

6、测试：

在用户主目录位置下载zImage

tftp 192.168.7.6 登陆tftp服务器

tftp> get zImage 下载文件zImage

Received 8295 bytes in 0.1 seconds

tftp> quit 退出tftp服务器:

## 5.4 NFS服务器配置

nfs服务器的安装配置和使用:

1、将已经制作好的文件系统rootfs.tar.gz 拷贝到 /opt，并解压

sudo tar -xvf rootfs.tar.gz

2、安装nfs服务器：

1》安装NFSserver端口映射和服务

sudo apt-get install nfs-kernel-server

2》先重启portmap服务：

sudo /etc/init.d/portmap restart

3》编辑/etc/exports,添加目标系统的根文件系统映射目录，假

如目标系统的根文件系统最后映射到/opt/rootfs,则

----》打开文件sudo vim /etc/exports，添加以下内容：

/opt/rootfs \*(subtree\_check,rw,no\_root\_squash,async)

4》重启服务：

$sudo /etc/init.d/nfs-kernel-server restart

$sudo exportfs -a(最好每修改过/etc/exports 后需要执行一次)

3、测试：

1》在用户主目录中创建一个挂载点: sudo mkdir /mnt

2》挂载文件系统：/opt/rootfs

sudo mount 192.168.7.6:/opt/rootfs /mnt

3》查看mnt下是否可以看到rootfs中的内容,如果可以看到，nfs OK

## 5.5 SSH服务器配置

1》ssh的安装：

sudo apt-get install ssh

2》putty的使用：



3》登录：



4》配置：

