目录

[1-1 ※写一个"标准"宏MIN ，这个宏输入两个参数并返回较小的一个 2](#_Toc485643486)

[1-2 ※用预处理指令#define 声明一个常数，用以表明1年中有多少秒 2](#_Toc485643487)

[1-3 ※把上面的define.c 写成一个Makefile文件并编译出可执行文件 2](#_Toc485643488)

[1-4 ※交换两个数的值 3](#_Toc485643489)

[1-5 ※int main（int argc char \*argv[]）的用法 3](#_Toc485643490)

[1-6 ※fork（）的用法 4](#_Toc485643491)

[1-7 ※exit（）用法 6](#_Toc485643492)

[1-8 ※sleep（）用法 6](#_Toc485643493)

[1-9 ※用fprintf()/fscanf() 6](#_Toc485643494)

[1-10 ※用extern 存储类 7](#_Toc485643495)

[1-11 ※大端小端判断 8](#_Toc485643496)

[1-12 ※Linux命令 set 9](#_Toc485643497)

[1-13 ※Linux命令 man 1,2,3.. 9](#_Toc485643498)

[1-15 ※C语言字符操作 9](#_Toc485643499)

[1-16※struct的初始化 10](#_Toc485643500)

[1-17※关键字volatile有什么含意?并给出三个不同的例子 10](#_Toc485643501)

[1-18※unsigned int a=1; int b=-20 (a+b)>0 10](#_Toc485643502)

[1-19※Linux I/O操作 open read write open（系统） 11](#_Toc485643503)

[1-20※Linux socket 本地通信 12](#_Toc485643504)

[1-21※同步和异步使用 16](#_Toc485643505)

[1-22※select（）轮询函数 16](#_Toc485643506)

# 1-1 ※写一个"标准"宏MIN ，这个宏输入两个参数并返回较小的一个

#include<stdio.h>

**#define MIN(x,y) x<y? x:y //后面的比较：输出小的输 x,Y的值顺序不要反**

int main()

{

static int a=5;

static int b=9;

int c;

scanf("plate 输入数字：%d%d\n",&a,&b);

printf("a=%d\n b=%d\n",a,b);

**c=MIN(a,b); //使用**

printf("c= %d\n",c);

}

* ***用宏的风险***

#include <stdio.h>

#define MAX( a,b) a>b? a:b //编译器会展开为 ++a>b ++a:b a的值加两次

//int max(int a,int b) //用着种方法就不会产生加两次问题只加一次

//{

// return("max=%d\n",a>b? a:b);

//}

int main()

{

int c=2,d=2,m;

//printf("m=%d\n",max(++c,d));//输出为3

printf("max==%d\n",MAX(++c,d)); //输出为4

}

# 1-2 ※用预处理指令#define 声明一个常数，用以表明1年中有多少秒

#include<stdio.h>

**#define COUNT\_YEAR\_MIN ((unsigned long)(60 \* 60 \* 24 \* 365))//注长度**

int main()

{

int c;

**c=COUNT\_YEAR\_MIN; //使用**

printf("c=%d\n",c);

}

# 1-3 ※把上面的define.c 写成一个Makefile文件并编译出可执行文件

edit: define.o

**cc -o define define.o @ 必须tab键开头 @**

define.o:define.c

cc -c define.c

clean:

rm **define** define.o

**带变量的makefile**

**objects = define.o**

edit:$(objects)

cc -o define $(objects)

define.o: define.c

cc -c define.c

clean:

rm define $(objects)

# 1-4 ※交换两个数的值

#include<stdio.h>

**void swap(int \*a,int \*b)**

{

int c;

c=\*a;

\*a=\*b;

\*b=c;

}

int main()

{

int d=10;

int f=55;

swap(&d,&f);

printf("%d\n%d\n",d ,f);

}

# 1-5 ※int main（int argc char \*argv[]）的用法

#include <stdio.h>

int main(int argc,char\* argv[])

{

int i;

for (i=0;i<argc;i++)

{

printf("%s\n",argv[i]);

printf("%d\n",i);

printf("%d\n",argc);

}

system("pause");

return 0;

}

**解释：**

argc：表示执行可运行程序后的参数

argv[]: 此可执行文件的存储路径、程序变量

**如：**执行./main 无参数

argc: 为1

argv[i]：为工程的路径和名称如：C:\Qt\Qt5.3.2\Tools\QtCreator\bin\build-untitled2-Desktop\_Qt\_5\_3\_MinGW\_32bit-Debug\debug\debug\untitled2.exe

运行结果：

C:\Qt\Qt5.3.2\Tools\QtCreator\bin\build-untitled2-Desktop\_Qt\_5\_3\_MinGW\_32bit-Debug\debug\debug\untitled2.exe

i=0

argc=1

**如：**执行./main sss.c(什么都行)

argc: 为2

argv[i]：如下

运行结果：

C:\Qt\Qt5.3.2\Tools\QtCreator\bin\build-untitled2-Desktop\_Qt\_5\_3\_MinGW\_32bit-Debug\debug\debug\untitled2.exe

i=0

argc=2

ssss.c

i=1

argc=2

# 1-6 ※fork（）的用法

#include<sys/types.h>

#include<unistd.h>

#include<stdio.h>

int main(int argc,char\* argv[])

{

pid\_t pid;

pid = fork();

if (pid < 0 ) {

printf("error!");

} else if( pid ==0 ) {

printf("This is the child process!\n");

} else{

printf("This is the parent process!\n child process id = %d\n", pid);

}

return 0;

}

疑问：为什么每次执行结果不一样

无固定顺序是因为子程序与父程序并发导致的，因为在用fork（）函数创建的3个进程同时在cpu中运行，因为3个进程的本质是一样的，所以就会抢占资源，因此形成一个队列，因为每次的排序不同，故此输出的结果也会不同。

**●fork的用法二**

#include<sys/types.h>

#include<unistd.h>

#include<stdio.h>

int main(int argc,char\* argv[])

{

pid\_t pid;

pid = fork();

if(pid < 0)

printf("error\n");

else if(pid == 0){ //等于0是子进程

printf("The child pid = %d\n",getpid());

}

else{

printf("The parent pid = %d\n",getpid());

}

}

Fork 返回值：**0 成功创建子进程**

>0 成功创建父进程，返回进程PID

-1 失败返回父进程

**运行结果：**

**The parent pid = 16938**

**The child pid = 16939**

# 1-7 ※exit（）用法

主要用到子函数中

exit（0）：正常运行程序并退出程序；

exit（1）：非正常运行导致退出程序；

* + main函数调用return
  + 调用exit
  + 调用\_exit
  + 调用abort
  + 被一个信号终止

区别 abort与exit

exit会做一些释放工作**：释放所有的静态的全局的对象，缓存，关掉所有的I/O通道**，然后终止程序。如果有函数通过atexit来注册，还会调用注册的函数。不过，如果atexit函数扔出异常的话，就会直接调用terminate。  
abort：立刻terminate程序，**没有任何清理工作**。

# 1-8 ※sleep（）用法

Sleep（1）：1s 秒

Usleep( 1000) 1000 毫秒

# 1-9 ※用fprintf()/fscanf()

#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

#define FILE\_PATH "./demo.txt" // 文件路径

int main(){

int i=20;

char name[20]="yang"; // 姓名

int age=20; // 年龄

int studNo=45454; // 学号

float score=222; // 平均分

FILE \*fp; // 文件指针

// 判断文件是否能够正确创建/打开

if( (fp=fopen(FILE\_PATH,"wt+")) == NULL ){

perror(FILE\_PATH);

exit(1);

}

// 从控制台输入学生信息并写入文件

printf("(please enter以空格分隔)：\n");

// scanf("%s %d %d %f", name, &age, &studNo, &score);

while(1)

{

static age=555;

studNo--;

score--;

fprintf(fp,"%s\t%d\t%d\t%f\n", name, age, studNo, score);

// 刷新缓冲区，将缓冲区的内容写入文件

fflush(fp);

// 重置文件内部位置指针，让位置指针指向文件开头

rewind(fp);

// 从文件中读取学生信息

//printf("read file:\n");

while(fscanf(fp, "%s\t%d\t%d\t%f", name, &age, &studNo, &score) != EOF){

printf("%s %d %d %f\n", name, age, studNo, score);

}

if(i<0) break;

i--;

}

fclose(fp);

return EXIT\_SUCCESS;

}

# 1-10 ※用extern 存储类

extern 修饰符通常用于当有**两个或多个文件共享相同的全局变量或函数**的时候。

**如 extern\_1.c**

#include <stdio.h>

**extern void swap(int \*a,int \*b)**

{

int c;

c=\*a;

\*a=\*b;

\*b=c;

}

**如extern\_2.c**

#include <stdio.h>

int main()

{

int i=0,h=99999;

printf("before i= %d\t h= %d\n",i,h);

**swap(&i,&h);//能使用其他文件的函数**

printf("after i= %d\t h= %d\n",i,h);

return 0;

}

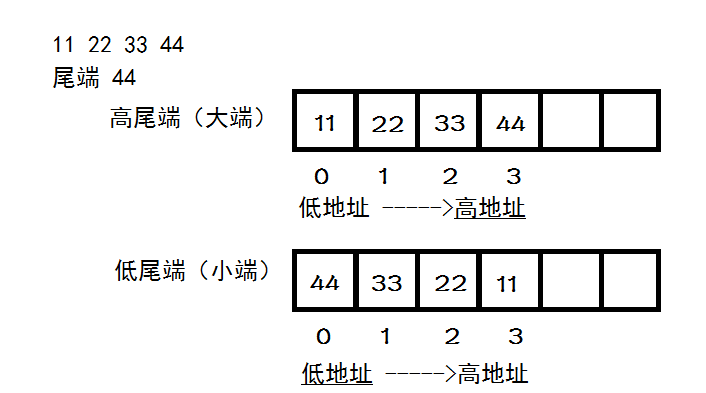
输出：before i= 0 h= 99999

after i= 99999 h= 0

# 1-11 ※大端小端判断

大端模式：是指数据的高字节保存在内存的低地址中，而数据的低字节保存在内存的高地址端。

小端模式，是指数据的高字节保存在内存的高地址中，低位字节保存在在内存的低地址端。

intel芯片是小端(修改分区表时要注意)，**单片机一般为大端**

编程检测：

#include <stdio.h>

int check()

{

union UN //union是共享内存的

{

char c;

int i;

}un;

un.i = 1;

return un.c;

}

int main(void)

{

if(check()==1)

printf("little\n");

else

printf("big\n");

return 0;

}

# 1-12 ※Linux命令 set

作用主要是**显示系统中已经存在的shell变量**，以及**设置shell变量的新变量值。**使用set更改shell特性时，符号"+"和"-"的作用分别是打开和关闭指定的模式  
来自: <http://man.linuxde.net/set>

**set 用来显示本地变量**  
env 用来显示环境变量  
export 用来显示和设置环境变量s

set 显示当前shell的变量，包括当前用户的变量  
env 显示当前用户的变量  
export 显示当前导出成用户变量的shell变量

每个shell有自己特有的变量（set）显示的变量

# 1-13 ※Linux命令 man 1,2,3..

1、standard command （标准命令）

**2、System calls （系统调用）  
3、Library functions （库函数）**方法是把一些常用到的函数编完放到一个文件里，供不同的人进行调用。调用的时候把它所在的文件名用#include<>加到里面就可以了。一般是放到lib文件里的。  
4、Special devices （设备说明）  
5、File formats （文件格式）

# 1-15 ※C语言字符操作

|  |  |
| --- | --- |
| 1 | **strcpy(s1, s2);** 复制字符串 s2 到字符串 s1。 |
| 2 | **strcat(s1, s2);** 连接字符串 s2 到字符串 s1 的末尾。 |
| 3 | **strlen(s1);** 返回字符串 s1 的长度。 |
| 4 | **strcmp(s1, s2);** 如果 s1 和 s2 是相同的，则返回 0；如果 s1<s2 则返回小于 0；如果 s1>s2 则返回大于 0。 |
| 5 | **strchr(s1, ch);** 返回一个指针，指向字符串 s1 中字符 ch 的第一次出现的位置。 |
| 6 | **strstr(s1, s2);** 返回一个指针，指向字符串 s1 中字符串 s2 的第一次出现的位置。 |

# 1-16※struct的初始化

#include <stdio.h>

#include <string.h>

struct stud //定义一个结构体

{

int age;

char \*name;

int score;

};

void out(struct stud \*s) //**定义一个指向结构体的指针**

{

printf("age=%d\n",s->age);//取出初始化结构体的值

printf("name=%s\n",s->name);/\*用指向该结构的指针,访问结构的成员，必须使用 -> 运算符\*/

printf("score=%d\n",s->score);

}

int main()

{

struct stud std= //初始化结构体stud 类型std

{ //**声明 std，类型为 stud**

.age = 18,

.name = "yang",

.score = 99,

};

out(&std); //**通过传 std 的地址来输出 std 信息**

}

# 1-17※关键字volatile有什么含意?并给出三个不同的例子

* Volatile修饰符告诉**编译器**，不要对**被修饰变量所参与的操作进行优化**。
* 一个定义为[volatile](http://www.baidu.com/s?wd=volatile&hl_tag=textlink&tn=SE_hldp01350_v6v6zkg6" \t "_blank)的变量是说这变量可能会被**意想不到地改变**，这样，[编译器](http://www.baidu.com/s?wd=%E7%BC%96%E8%AF%91%E5%99%A8&hl_tag=textlink&tn=SE_hldp01350_v6v6zkg6)就不会去**假设这个变量的值**了。精确地说就是，优化器在用到这个变量时必须每次都小心地重新读取这个变量的值，而不是使用保存在[寄存器](http://www.baidu.com/s?wd=%E5%AF%84%E5%AD%98%E5%99%A8&hl_tag=textlink&tn=SE_hldp01350_v6v6zkg6)里的备份。下面是[volatile](http://www.baidu.com/s?wd=volatile&hl_tag=textlink&tn=SE_hldp01350_v6v6zkg6)变量的几个例子：  
  　　**1). 并行设备的硬件[寄存器](http://www.baidu.com/s?wd=%E5%AF%84%E5%AD%98%E5%99%A8&hl_tag=textlink&tn=SE_hldp01350_v6v6zkg6" \t "_blank)（如：[状态寄存器](http://www.baidu.com/s?wd=%E7%8A%B6%E6%80%81%E5%AF%84%E5%AD%98%E5%99%A8&hl_tag=textlink&tn=SE_hldp01350_v6v6zkg6" \t "_blank)）  
  　　2). 一个中断服务子程序中会访问到的非[自动变量](http://www.baidu.com/s?wd=%E8%87%AA%E5%8A%A8%E5%8F%98%E9%87%8F&hl_tag=textlink&tn=SE_hldp01350_v6v6zkg6" \t "_blank)(Non-automatic variables)  
  　　3).** [**多线程**](http://www.baidu.com/s?wd=%E5%A4%9A%E7%BA%BF%E7%A8%8B&hl_tag=textlink&tn=SE_hldp01350_v6v6zkg6)**应用中被几个任务共享的变量**

# 1-18※unsigned int a=1; int b=-20 (a+b)>0

#include <stdio.h>

int main(void)

{

unsigned int a=2;

int b=-20;

if((a+b)>10000000)**//如果无符号的变量和有符号的变量相运算时会是一个非常大的数 [除了-运算]**

{

printf("a+b=%u\n",a+b);

printf("a-b=%u\n",a-b);

printf("a\*b=%u\n",a\*b);

printf("b/a=%u\n",b/a);

}else

{

printf("a+b<0\n");

}

return 0;

}

a+b=4294967278

a-b=22

a\*b=4294967256

b/a=2147483638

%d 有符号10进制整数 %u 无符号10进制整数

# 1-19※Linux I/O操作 open read write open（系统）

flags：（falgs参数可通过“|”组合构成，只读、只写、读写 这三种方式是互斥的，不可同时使用）  
        O\_RDONLY（只读方式打开），  
        O\_WRONLY（只写方式打开），  
        O\_RDWR（读/写方式打开），  
        O\_CREAT（如果文件不存在，就创建新的文件），  
        O\_EXCL（如果使用O\_CREAT时，文件存在，则可返回错误信息），  
        O\_TRUNC（如果文件已存在，且以只读或只写方式打开，则先删除文件中的原有数据），  
        O\_APPEND（以添加方式打开文件，在打开文件的同时，文件指针指向文件末尾）

 为了防止对文件的意外操作，往往要以合适的方式打开文件（只读，只写），每个文件只负责一个特定的用  
    途，有利于提高这些文件的重复利用

#include <unistd.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <fcntl.h>

#include <stdio.h>

#define NAME "file.log"

int main(void){

int fd,size,w;//fd：文件描述符，用于判断打开|新建文件是否成功，size：读取到的字节数

char s[] = "hellossssssssss\n----->\n52661314ll";//需要写入的字符串

char **buffer**[80];//存储读出数据的缓冲区

fd = **open**(NAME, **O\_WRONLY | O\_CREAT**);//以只写方式打开文件，若不存在则创建文件

if (**fd == -1**){

printf("Open or create file failed.\n");

return -1;

}

w=**write**(fd, s, sizeof(s));//向该文件写入字符串

printf("w==%d\n",w);

close(fd);

fd = **open**(NAME, O\_RDONLY);//以只读方式打开文件

if (fd == -1){

printf("Open file failed.\n");

return -1;

}

size = **read**(fd, buffer, sizeof(buffer));//读取文件内容到buffer并返回读取的字符个数

printf("size==%d\n",size);/**/w=size 字符长度**

**close**(fd);

printf("%s", buffer);

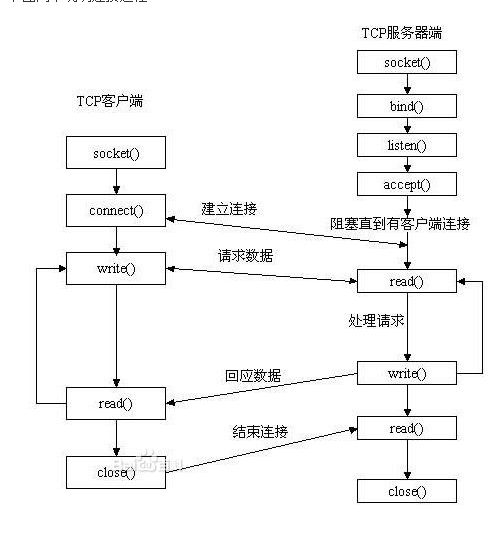
return 0;

}

# 1-20※Linux socket 本地通信

* **理论篇**

两个进程有唯一的PID，在本地进程通讯中我们可以使用PID来唯一标示一个本地进程，但PID只在本地唯一，网络中的两个进程PID冲突几率很大。IP层的ip地址可以唯一标示主机，而TCP层协议和端口号可以唯一标示主机的一个进程，这样我们可以利用ip地址＋协议＋端口号唯一标示网络中的一个进程。socket起源于Unix，而Unix/[Linux](http://lib.csdn.net/base/linux)基本哲学之一就是“一切皆文件”，都可以用“打开open –> 读写write/read –> 关闭close”模式来操作。Socket文件。(http://blog.csdn.net/sight\_/article/details/8138802)



* **Socket()函数**

**int socket(int domain, int type, int protocol)**

eg:int socket(AF\_INET,SCOK\_STREAM,0) 【常用】

 -**domain**：即协议域，又称为协议族（family）。常用的协议族有，**AF\_INET（PF\_INET没有区别）**、AF\_INET6、AF\_LOCAL（或称AF\_UNIX，Unix域socket）、AF\_ROUTE等等。协议族决定了socket的地址类型，在通信中必须采用对应的地址，如AF\_INET决定了要用ipv4地址（32位的）与端口号（16位的）的组合、**AF\_UNIX决定了要用一个绝对路径名作为地址**。

  **-type：**指定socket类型。常用的socket类型有，SOCK\_STREAM、SOCK\_DGRAM、SOCK\_RAW、SOCK\_PACKET、SOCK\_SEQPACKET等等（socket的类型有哪些？）。

 -**protocol**：就是指定协议。常用的协议有，IPPROTO\_TCP、IPPTOTO\_UDP、IPPROTO\_SCTP、IPPROTO\_TIPC等，它们分别对应TCP传输协议、UDP传输协议、STCP传输协议、TIPC传输协议（这个协议我将会单独开篇讨论！）。

注意：并不是上面的type和protocol可以随意组合的，如SOCK\_STREAM不可以跟IPPROTO\_UDP组合。**当protocol为0时（常用），会自动选择type类型对应的默认协议**。

当我们调用**socket**创建一个socket时，返回的socket描述字它存在于协议族（address family，AF\_XXX）空间中，但没有一个具体的地址。如果想要给它赋值一个地址，就必须调用**bind()函数（绑定一个具体的地址）**，否则就当调用connect()、listen()时系统会自动随机分配一个端口。

返回值：

success :返回一个新的socket fail: 返回 - 1

* **bind()函数**

int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

**- sockfd**：即socket描述字，它是通过fd=socket()函数

**- addr**：一个const struct sockaddr \*指针，**指向要绑定给sockfd的协议地址**。这个地址结构根据地址创建socket时的地址协议族的不同而不同.

**Unix域对应的是**：

**#define UNIX\_PATH\_MAX 108**

struct sockaddr\_un {

sa\_family\_t sun\_family; /\* AF\_UNIX \*/

char sun\_path[UNIX\_PATH\_MAX]; /\* pathname \*/

};

**ipv4对应的是：**

struct sockaddr\_in {

sa\_family\_t sin\_family; /\* address family: AF\_INET \*/

in\_port\_t sin\_port; /\* port in network byte order \*/

struct in\_addr sin\_addr; /\* internet address \*/

};

/\* Internet address. \*/

struct in\_addr {

uint32\_t s\_addr; /\* address in network byte order \*/

};

ipv6对应的是：

struct sockaddr\_in6 {

sa\_family\_t sin6\_family; /\* AF\_INET6 \*/

in\_port\_t sin6\_port; /\* port number \*/

uint32\_t sin6\_flowinfo; /\* IPv6 flow information \*/

struct in6\_addr sin6\_addr; /\* IPv6 address \*/

uint32\_t sin6\_scope\_id; /\* Scope ID (new in 2.4) \*/

};

**主要事项：网络字节序与主机字节序**

**主机字节序**就是我们平常说的大端和小端模式：不同的CPU有不同的字节序类型，这些字节序是指整数在内存中保存的顺序，这个叫做主机序。引用标准的Big-Endian和Little-Endian的定义如下：

　　a) Little-Endian就是低位字节排放在内存的低地址端，高位字节排放在内存的高地址端。

　　b) Big-Endian就是高位字节排放在内存的低地址端，低位字节排放在内存的高地址端。

**网络字节序**：4个字节的32 bit值以下面的次序传输：首先是0～7bit，其次8～15bit，然后16～23bit，最后是24~31bit。这种传输次序称作大端字节序。**由于TCP/IP首部中所有的二进制整数在网络中传输时都要求以这种次序，因此它又称作网络字节序。**字节序，顾名思义字节的顺序，就是大于一个字节类型的数据在内存中的存放顺序，一个字节的数据没有顺序的问题了。

所以：在将一个地址绑定到socket的时候，**请先将主机字节序转换成为网络字节序，而不要假定主机字节序跟网络字节序一样使用的是Big-Endian。**由于这个问题曾引发过血案！公司项目代码中由于存在这个问题，导致了很多莫名其妙的问题，所以请谨记对**主机字节序不要做任何假定，务必将其转化为网络字节序再赋给socket。**

返回值：

success :返回一个新的socket fail: 返回 - 1

* **addrlen**：对应的是地址的长度。
* **listen（）/ connect（）函数**

int listen(int sockfd, int backlog);

int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

listen函数的第一个参数即为要监听的socket描述字，第二个参数为相应***socket可以排队的最大连接个数***。**socket()函数创建的socket默认是一个主动类型的，listen函数将socket变为被动类型的，等待客户的连接请求**。

connect函数的第一个参数即为客户端的socket描述字，第二参数为服务器的socket地址，**第三个参数为socket地址的长度。客户端通过调用connect函数来建立与TCP服务器的连接。**

返回值：

success :返回一个新的socket fail: 返回 - 1

* **accept（）函数（while（1）{ accept（）阻塞在这里}）**

int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);

TCP服务器端依次调用socket()、bind()、listen()之后，**就会监听指定的socket地址了**。TCP客户端依次调用socket()、connect()之后就想TCP服务器发送了一个连接请求。TCP服务器监听到这个请求之后，**就会调用accept()函数取接收请求，这样连接就建立好了。之后就可以开始网络I/O操作了，即类同于普通文件的读写I/O操作**。

accept函数的第一个参数为服务器的socket描述字，第二个参数为指向struct sockaddr \*的指针，**用于返回客户端的协议地址**，第三个参数为协议地址的长度。如果accpet成功，那么其返回值是由内核自动生成的一个全新的描述字，**代表与返回客户的TCP连接**。

注意：accept的第一个参数为服务器的socket描述字，是服务器开始调用socket()函数生成的，称为监听socket描述字；**而accept函数返回的是已连接的socket描述字**。一个服务器通常仅只创建一个监听socket描述字，它在该服务器的生命周期内一直存在。内核为每个由服务器进程接受的客户连接创建了一个已连接socket描述字，当服务器完成了对某个客户的服务，相应的已连接socket描述字就被关闭。

返回值：

success :返回一个新的socket fail: 返回 - 1

# 1-21※同步和异步使用

同步：（用于不耗时的操作或者多线程用同一个资源时只能有一个进程访问资源，其他线程必须等待。同步的机制: 临界区、互斥、信号量和事件）

A

C

B

异步：（有耗时的操作时）

C

B

A

或者不用返回

（正常来说是不用回的）

# 1-22※select（）轮询函数

int select(int nfds, fd\_set \*readfds, fd\_set \*writefds,

fd\_set \*exceptfds, struct timeval \*timeout);

# 1-23※select（）轮询函数