

[字符串转换.pdf](#)

[内存控制.pdf](#)

[日期时间.pdf](#)

[内存及字符操作.pdf](#)

[常用数学函数.pdf](#)

[用户组.pdf](#)

[数据结构及算法.pdf](#)

[文件操作.pdf](#)

[文件内容操作.pdf](#)

[进程操作.pdf](#)

[文件权限操作.pdf](#)

[信号处理.pdf](#)

[借口处理.pdf](#)

[环境变量.pdf](#)

[终端操作.pdf](#)

atof (将字符串转换成浮点型数)

相关函数 atoi, atol, strtod, strtol, strtoul

表头文件 #include <stdlib.h>

定义函数 double atof(const char *nptr);

函数说明 atof()会扫描参数nptr字符串，跳过前面的空格字符，直到遇上数字或正负符号才开始做转换，而再遇到非数字或字符串结束时('\0')才结束转换，并将结果返回。参数nptr字符串可包含正负号、小数点或E(e)来表示指数部分，如123.456或123e-2。

返回值 返回转换后的浮点型数。

附加说明 atof()与使用strtod(nptr,(char**)NULL)结果相同。

```
范例 /* 将字符串a 与字符串b转换成数字后相加*/
#include<stdlib.h>
main()
{
    char *a="-100.23";
    char *b="200e-2";
    float c;
    c=atof(a)+atof(b);
    printf("c=%.2f\n",c);
}
```

执行 c=-98.23

atoi (将字符串转换成整型数)

相关函数 atof, atol, atrtod, strtol, strtoul

表头文件 #include<stdlib.h>

定义函数 int atoi(const char *nptr);

函数说明 atoi()会扫描参数nptr字符串，跳过前面的空格字符，直到遇上数字或正负符号才开始做转换，而再遇到非数字或字符串结束时('\0')才结束转换，并将结果返回。

返回值 返回转换后的整型数。

附加说明 atoi()与使用strtol(nptr, (char**)NULL, 10); 结果相同。

```
范例 /* 将字符串a 与字符串b转换成数字后相加*/
#include<stdlib.h>
mian()
{
    char a[]="-100";
    char b[]="456";
    int c;
    c=atoi(a)+atoi(b);
    printf(c=%d\n",c);
}
```

执行 c=356

atol (将字符串转换成长整型数)

相关函数 `atof`, `atoi`, `strtod`, `strtol`, `strtoul`

表头文件 `#include<stdlib.h>`

定义函数 `long atol(const char *nptr);`

函数说明 `atol()`会扫描参数`nptr`字符串，跳过前面的空格字符，直到遇上数字或正负符号才开始做转换，而再遇到非数字或字符串结束时('\0')才结束转换，并将结果返回。

返回值 返回转换后的长整型数。

附加说明 `atol()`与使用`strtol(nptr,(char**)NULL,10)`；结果相同。

范例 /*将字符串a与字符串b转换成数字后相加*/

```
#include<stdlib.h>
main()
{
    char a[]="1000000000";
    char b[]=" 234567890";
    long c;
    c=atol(a)+atol(b);
    printf("c=%d\n",c);
}
```

执行 `c=1234567890`

`gcvt`（将浮点型数转换为字符串，取四舍五入）

相关函数 `ecvt`, `fcvt`, `sprintf`

表头文件 `#include<stdlib.h>`

定义函数 `char *gcvt(double number, size_t ndigits, char *buf);`

函数说明 `gcvt()`用来将参数`number`转换成ASCII码字符串，参数`ndigits`表示显示的位数。`gcvt()`与`ecvt()`和`fcvt()`不同的地方在于，`gcvt()`所转换后的字符串包含小数点或正负符号。若转换成功，转换后的字符串会放在参数`buf`指针所指的空间。

返回值 返回一字符串指针，此地址即为`buf`指针。

附加说明

范例 `#include<stdlib.h>`

```
main()
{
    double a=123.45;
    double b=-1234.56;
    char *ptr;
    int decpt,sign;
    gcvt(a,5,ptr);
    printf("a value=%s\n",ptr);
    ptr=gcvt(b,6,ptr);
    printf("b value=%s\n",ptr);
}
```

执行 `a value=123.45`

`b value=-1234.56`

`strtod`（将字符串转换成浮点数）

相关函数 `atoi`, `atol`, `strtod`, `strtol`, `strtoul`

表头文件 `#include<stdlib.h>`

定义函数 `double strtod(const char *nptr,char **endptr);`

函数说明 strtod()会扫描参数nptr字符串，跳过前面的空格字符，直到遇上数字或正负符号才开始做转换，到出现非数字或字符串结束时('\0')才结束转换，并将结果返回。若endptr不为NULL，则会将遇到不合条件而终止的nptr中的字符指针由endptr传回。参数nptr字符串可包含正负号、小数点或E(e)来表示指数部分。如123.456或123e-2。

返回值 返回转换后的浮点型数。

附加说明 参考atof()。

范例 /*将字符串a, b, c 分别采用10, 2, 16 进制转换成数字*/

```
#include<stdlib.h>
main()
{
    char a[]="10000000000";
    char b[]="10000000000";
    char c[]="ffff";
    printf("a=%d\n",strtod(a,NULL,10));
    printf("b=%d\n",strtod(b,NULL,2));
    printf("c=%d\n",strtod(c,NULL,16));
}
```

执行 a=10000000000
b=512
c=65535

strtol（将字符串转换成长整型数）

相关函数 atof, atoi, atol, strtod, strtoul

表头文件 #include<stdlib.h>

定义函数 long int strtol(const char *nptr,char **endptr,int base);

函数说明 strtol()会将参数nptr字符串根据参数base来转换成长整型数。参数base范围从2至36，或0。参数base代表采用的进制方式，如base值为10则采用10进制，若base值为16则采用16进制等。当base值为0时则是采用10进制做转换，但遇到如'0x'前置字符则会使用16进制做转换。一开始strtol()会扫描参数nptr字符串，跳过前面的空格字符，直到遇上数字或正负符号才开始做转换，再遇到非数字或字符串结束时('\0')结束转换，并将结果返回。若参数endptr不为NULL，则会将遇到不合条件而终止的nptr中的字符指针由endptr返回。

返回值 返回转换后的长整型数，否则返回ERANGE并将错误代码存入errno中。

附加说明 ERANGE指定的转换字符串超出合法范围。

范例 /* 将字符串a, b, c 分别采用10, 2, 16进制转换成数字*/

```
#include<stdlib.h>
main()
{
    char a[]="10000000000";
    char b[]="10000000000";
    char c[]="ffff";
    printf("a=%d\n",strtol(a,NULL,10));
    printf("b=%d\n",strtol(b,NULL,2));
    printf("c=%d\n",strtol(c,NULL,16));
}
```

执行 a=10000000000
b=512
c=65535

strtoul（将字符串转换成无符号长整型数）

相关函数 `atof`, `atoi`, `atol`, `strtod`, `strtol`

表头文件 `#include <stdlib.h>`

定义函数 `unsigned long int strtoul(const char *nptr, char **endptr, int base);`

函数说明 `strtoul()`会将参数`nptr`字符串根据参数`base`来转换成无符号的长整型数。参数`base`范围从2至36，或0。参数`base`代表采用的进制方式，如`base`值为10则采用10进制，若`base`值为16则采用16进制数等。当`base`值为0时则是采用10进制做转换，但遇到如'0x'前置字符则会使用16进制做转换。一开始`strtoul()`会扫描参数`nptr`字符串，跳过前面的空格字符串，直到遇上数字或正负符号才开始做转换，再遇到非数字或字符串结束时('\0')结束转换，并将结果返回。若参数`endptr`不为NULL，则会将遇到不合条件而终止的`nptr`中的字符指针由`endptr`返回。

返回值 返回转换后的长整型数，否则返回ERANGE并将错误代码存入`errno`中。

附加说明 ERANGE指定的转换字符串超出合法范围。

范例 参考`strtol()`

`toascii`（将整型数转换成合法的ASCII 码字符）

相关函数 `isascii`, `toupper`, `tolower`

表头文件 `#include <ctype.h>`

定义函数 `int toascii(int c)`

函数说明 `toascii()`会将参数`c`转换成7位的`unsigned char`值，第八位则会被清除，此字符即会被转成ASCII码字符。

返回值 将转换成功的ASCII码字符值返回。

范例 `#include <stdlib.h>`

```
main()
{
    int a=217;
    char b;
    printf("before toascii () : a value =%d(%c)\n",a,a);
    b=toascii(a);
    printf("after toascii() : a value =%d(%c)\n",b,b);
}
```

执行 before toascii() : a value =217()
after toascii() : a value =89(Y)

`tolower`（将大写字母转换成小写字母）

相关函数 `isalpha`, `toupper`

表头文件 `#include <stdlib.h>`

定义函数 `int tolower(int c);`

函数说明 若参数`c`为大写字母则将该对应的小写字母返回。

返回值 返回转换后的小写字母，若不须转换则将参数`c`值返回。

附加说明

范例 `/* 将s字符串内的大写字母转换成小写字母*/`

```
#include <ctype.h>
main()
{
    char s[]="aBcDeFgH12345;!#$";
    int i;
    printf("before tolower() : %s\n",s);
    for(i=0;I<sizeof(s);i++)
```

```
s[i]=tolower(s[i]);  
printf("after tolower() : %s\n",s);  
}
```

执行 before tolower() : aBcDeFgH12345;!#\$
after tolower() : abcdefgh12345;!#\$

toupper（将小写字母转换成大写字母）

相关函数 isalpha, tolower

表头文件 #include <ctype.h>

定义函数 int toupper(int c);

函数说明 若参数c为小写字母则将该对映的大写字母返回。

返回值 返回转换后的大写字母，若不须转换则将参数c值返回。

附加说明

范例 /* 将s字符串内的小写字母转换成大写字母*/

```
#include <ctype.h>  
main()  
{  
char s[]="aBcDeFgH12345;!#$";  
int i;  
printf("before toupper() : %s\n",s);  
for(i=0;i<sizeof(s);i++)  
s[i]=toupper(s[i]);  
printf("after toupper() : %s\n",s);  
}
```

执行 before toupper() : aBcDeFgH12345;!#\$
after toupper() : ABCDEFGH12345;!#\$

calloc（配置内存空间）

相关函数 malloc, free, realloc, brk

表头文件 #include <stdlib.h>

定义函数 void *calloc(size_t nmemb, size_t size);

函数说明 calloc()用来配置nmemb个相邻的内存单位，每一单位的大小为size，并返回指向第一个元素的指针。这和使用下列的方式效果相同:malloc(nmemb*size);不过，在利用calloc()配置内存时会先将内存内容初始化为0。

返回值 若配置成功则返回一指针，失败则返回NULL。

```
范例 /* 动态配置10个struct test 空间*/
#include<stdlib.h>
struct test
{
    int a[10];
    char b[20];
}
main()
{
    struct test *ptr=calloc(sizeof(struct test),10);
}
```

free（释放原先配置的内存）

相关函数 malloc, calloc, realloc, brk

表头文件 #include<stdlib.h>

定义函数 void free(void *ptr);

函数说明 参数ptr为指向先前由malloc()、calloc()或realloc()所返回的内存指针。调用free()后ptr所指的内存空间便会被收回。假若参数ptr所指的内存空间已被收回或是未知的内存地址，则调用free()可能会有无法预期的情况发生。若参数ptr为NULL，则free()不会有任何作用。

getpagesize（取得内存分页大小）

相关函数 sbrk

表头文件 #include<unistd.h>

定义函数 size_t getpagesize(void);

函数说明 返回一分页的大小，单位为字节（byte）。此为系统的分页大小，不一定会和硬件分页大小相同。

返回值 内存分页大小。附加说明在Intel x86 上其返回值应为4096bytes。

```
范例 #include <unistd.h>
main()
{
    printf("page size = %d\n",getpagesize());
}
```

malloc（配置内存空间）

相关函数 calloc, free, realloc, brk

表头文件 #include<stdlib.h>

定义函数 void * malloc(size_t size);

函数说明 malloc()用来配置内存空间，其大小由指定的size决定。

返回值 若配置成功则返回一指针，失败则返回NULL。

范例 void p = malloc(1024); /*配置1k的内存*/

mmap（建立内存映射）

相关函数 munmap, open

表头文件 #include <unistd.h>

#include <sys/mman.h>

定义函数 void *mmap(void *start,size_t length,int prot,int flags,int fd,off_t offsize);

函数说明 mmap()用来将某个文件内容映射到内存中，对该内存区域的存取即是直接对该文件内容的读写。参数start指向欲对应的内存起始地址，通常设为NULL，代表让系统自动选定地址，对应成功后该地址会返回。参数length代表将文件中多大的部分对应到内存。

参数 prot代表映射区域的保护方式有下列组合

PROT_EXEC 映射区域可被执行

PROT_READ 映射区域可被读取

PROT_WRITE 映射区域可被写入

PROT_NONE 映射区域不能存取

参数 flags会影响映射区域的各种特性

MAP_FIXED 如果参数start所指的地址无法成功建立映射时，则放弃映射，不对地址做修正。通常不鼓励用此旗标。

MAP_SHARED对映射区域的写入数据会复制回文件内，而且允许其他映射该文件的进程共享。

MAP_PRIVATE 对映射区域的写入操作会产生一个映射文件的复制，即私人的“写入时复制”（copy on write）对此区域作的任何修改都不会写回原来的文件内容。

MAP_ANONYMOUS建立匿名映射。此时会忽略参数fd，不涉及文件，而且映射区域无法和其他进程共享。

MAP_DENYWRITE只允许对映射区域的写入操作，其他对文件直接写入的操作将会被拒绝。

MAP_LOCKED 将映射区域锁定住，这表示该区域不会被置换（swap）。

在调用mmap()时必须指定MAP_SHARED 或MAP_PRIVATE。参数fd为open()返回的文件描述词，代表欲映射到内存的文件。参数offset为文件映射的偏移量，通常设置为0，代表从文件最前方开始对应，offset必须是分页大小的整数倍。

返回值 若映射成功则返回映射区的内存起始地址，否则返回MAP_FAILED(-1)，错误原因存于errno 中。

错误代码 EBADF 参数fd 不是有效的文件描述词

EACCES 存取权限有误。如果是MAP_PRIVATE 情况下文件必须可读，使用MAP_SHARED则要有PROT_WRITE以及该文件要能写入。

EINVAL 参数start、length 或offset有一个不合法。

EAGAIN 文件被锁住，或是有太多内存被锁住。

ENOMEM 内存不足。

范例 /* 利用mmap()来读取/etc/passwd 文件内容*/

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

#include<unistd.h>

#include<sys/mman.h>

main()

{

int fd;


```

void *start;
struct stat sb;
fd=open("/etc/passwd",O_RDONLY); /*打开/etc/passwd*/
fstat(fd,&sb); /*取得文件大小*/
start=mmap(NULL,sb.st_size,PROT_READ,MAP_PRIVATE,fd,0);
if(start== MAP_FAILED) /*判断是否映射成功*/
return;
printf("%s",start);
munmap(start,sb.st_size); /*解除映射*/
closed(fd);
}

```

执行 root : x : 0 : root : /root : /bin/bash

bin : x : 1 : 1 : bin : /bin :

daemon : x : 2 : 2 : daemon : /sbin

adm : x : 3 : 4 : adm : /var/adm :

lp : x : 4 : 7 : lp : /var/spool/lpd :

sync : x : 5 : 0 : sync : /sbin : bin/sync :

shutdown : x : 6 : 0 : shutdown : /sbin : /sbin/shutdown

halt : x : 7 : 0 : halt : /sbin : /sbin/halt

mail : x : 8 : 12 : mail : /var/spool/mail :

news : x : 9 : 13 : news : /var/spool/news :

uucp : x : 10 : 14 : uucp : /var/spool/uucp :

operator : x : 11 : 0 : operator : /root:

games : x : 12 : 100 : games : /usr/games:

gopher : x : 13 : 30 : gopher : /usr/lib/gopher-data:

ftp : x : 14 : 50 : FTP User : /home/ftp:

nobody : x : 99 : 99 : Nobody : /:

xfx : x : 100 : 101 : X Font Server : /etc/x11/fs : /bin/false

gdm : x : 42 : 42 : : /home/gdm : /bin/bash

kids : x : 500 : 500 : /home/kids : /bin/bash

munmap (解除内存映射)

相关函数 mmap

表头文件 #include <unistd.h>

#include <sys/mman.h>

定义函数 int munmap(void *start,size_t length);

函数说明 munmap()用来取消参数start所指的映射内存起始地址, 参数length则是欲取消的内存大小。当进程结束或利用exec相关函数来执行其他程序时, 映射内存会自动解除, 但关闭对应的文件描述词时不会解除映射。

返回值 如果解除映射成功则返回0, 否则返回-1, 错误原因存于errno中错误代码EINVAL

参数 start或length 不合法。

范例 参考mmap ()

asctime (将时间和日期以字符串格式表示)

相关函数 time, ctime, gmtime, localtime

表头文件 #include<time.h>

定义函数 char * asctime(const struct tm * timeptr);

函数说明 asctime()将参数timeptr所指的tm结构中的信息转换成真实世界所使用的时间日期表示方法，然后将结果以字符串形态返回。此函数已经由时区转换成当地时间，字符串格式为:"Wed Jun 30 21:49:08 1993\n"

返回值 若再调用相关的时间日期函数，此字符串可能会被破坏。此函数与ctime不同处在于传入的参数是不同的结构。

附加说明 返回一字符串表示目前当地的时间日期。

范例 #include <time.h>

```
main()
{
    time_t timep;
    time (&timep);
    printf("%s",asctime(gmtime(&timep)));
}
```

执行 Sat Oct 28 02:10:06 2000

ctime (将时间和日期以字符串格式表示)

相关函数 time, asctime, gmtime, localtime

表头文件 #include<time.h>

定义函数 char *ctime(const time_t *timep);

函数说明 ctime()将参数timep所指的time_t结构中的信息转换成真实世界所使用的时间日期表示方法，然后将结果以字符串形态返回。此函数已经由时区转换成当地时间，字符串格式为:"Wed Jun 30 21 :49 :08 1993\n"。若再调用相关的时间日期函数，此字符串可能会被破坏。

返回值 返回一字符串表示目前当地的时间日期。

范例 #include<time.h>

```
main()
{
    time_t timep;
    time (&timep);
    printf("%s",ctime(&timep));
}
```

执行 Sat Oct 28 10 : 12 : 05 2000

gettimeofday (取得目前的时间)

相关函数 time, ctime, ftime, settimeofday

表头文件 #include <sys/time.h>

#include <unistd.h>

定义函数 int gettimeofday (struct timeval * tv , struct timezone * tz)

函数说明 `gettimeofday()` 会把目前的时间有 `tv` 所指的结构返回，当地时区的信息则放到 `tz` 所指的结构中。

`timeval` 结构定义为:

```
struct timeval{
    long tv_sec; /*秒*/
    long tv_usec; /*微秒*/
};
```

`timezone` 结构定义为:

```
struct timezone{
    int tz_minuteswest; /*和Greenwich 时间差了多少分钟*/
    int tz_dsttime; /*日光节约时间的状态*/
};
```

上述两个结构都定义在 `/usr/include/sys/time.h`。 `tz_dsttime` 所代表的状态如下

```
DST_NONE /*不使用*/
DST_USA /*美国*/
DST_AUST /*澳洲*/
DST_WET /*西欧*/
DST_MET /*中欧*/
DST_EET /*东欧*/
DST_CAN /*加拿大*/
DST_GB /*大不列颠*/
DST_RUM /*罗马尼亚*/
DST_TUR /*土耳其*/
DST_AUSTALT /*澳洲（1986年以后）*/
```

返回值 成功则返回0，失败返回-1，错误代码存于 `errno`。附加说明 `EFAULT` 指针 `tv` 和 `tz` 所指的内存空间超出存取权限。

```
范例 #include<sys/time.h>
#include<unistd.h>
main(){
    struct timeval tv;
    struct timezone tz;
    gettimeofday (&tv , &tz);
    printf("tv_sec; %d\n", tv.tv_sec);
    printf("tv_usec; %d\n",tv.tv_usec);
    printf("tz_minuteswest; %d\n", tz.tz_minuteswest);
    printf("tz_dsttime, %d\n",tz.tz_dsttime);
}
```

```
执行 tv_sec: 974857339
tv_usec:136996
tz_minuteswest:-540
tz_dsttime:0
```

`gmtime`（取得目前时间和日期）

相关函数 `time`, `asctime`, `ctime`, `localtime`

表头文件 `#include<time.h>`

定义函数 `struct tm*gmtime(const time_t*timep);`

函数说明 `gmtime()` 将参数 `timep` 所指的 `time_t` 结构中的信息转换成真实世界所使用的时间日期表示方法，然后将结果由结构 `tm` 返回。

结构 `tm` 的定义为

```
struct tm
{
    int tm_sec;
    int tm_min;
    int tm_hour;
```

```

int tm_mday;
int tm_mon;
int tm_year;
int tm_wday;
int tm_yday;
int tm_isdst;
};
int tm_sec 代表目前秒数，正常范围为0-59，但允许至61秒
int tm_min 代表目前分数，范围0-59
int tm_hour 从午夜算起的时数，范围为0-23
int tm_mday 目前月份的日数，范围01-31
int tm_mon 代表目前月份，从一月算起，范围从0-11
int tm_year 从1900 年算起至今的年数
int tm_wday 一周的日数，从星期一算起，范围为0-6
int tm_yday 从今年1月1日算起至今的天数，范围为0-365
int tm_isdst 日光节约时间的旗标
此函数返回的时间日期未经时区转换，而是UTC时间。

```

返回值 返回结构tm代表目前UTC 时间

```

范例 #include <time.h>
main(){
char *wday[]={"Sun","Mon","Tue","Wed","Thu","Fri","Sat"};
time_t timep;
struct tm *p;
time(&timep);
p=gmtime(&timep);
printf("%d%d%d",(1900+p->tm_year), (1+p->tm_mon),p->tm_mday);
printf("%s%d;%d;%d\n", wday[p->tm_wday], p->tm_hour, p->tm_min, p->tm_sec);
}

```

执行 2000/10/28 Sat 8:15:38

localtime（取得当地目前时间和日期）

相关函数 time, asctime, ctime, gmtime

表头文件 #include<time.h>

定义函数 struct tm *localtime(const time_t * timep);

函数说明 localtime()将参数timep所指的time_t结构中的信息转换成真实世界所使用的时间日期表示方法，然后将结果由结构tm返回。结构tm的定义请参考gmtime()。此函数返回的时间日期已经转换成当地时间。

返回值 返回结构tm代表目前的当地时间。

```

范例 #include<time.h>
main(){
char *wday[]={"Sun","Mon","Tue","Wed","Thu","Fri","Sat"};
time_t timep;
struct tm *p;
time(&timep);
p=localtime(&timep); /*取得当地时间*/
printf ("%d%d%d ", (1900+p->tm_year),( 1+p->tm_mon), p->tm_mday);
printf ("%s%d:%d:%d\n", wday[p->tm_wday],p->tm_hour, p->tm_min, p->tm_sec);
}

```

执行 2000/10/28 Sat 11:12:22

mktime（将时间结构数据转换成经过的秒数）

相关函数 time, asctime, gmtime, localtime

表头文件 #include<time.h>

定义函数 time_t mktime(struct tm * timeptr);

函数说明 mktime()用来将参数timeptr所指的tm结构数据转换成从公元1970年1月1日0时0分0 秒算起至今的UTC时间所经过的秒数。

返回值 返回经过的秒数。

```
范例 /* 用time()取得时间（秒数），利用localtime()
      转换成struct tm 再利用mktime（）将struct tm转换成原来的秒数*/
#include<time.h>
main()
{
    time_t timep;
    struct tm *p;
    time(&timep);
    printf("time() : %d \n",timep);
    p=localtime(&timep);
    timep = mktime(p);
    printf("time()->localtime()->mktime():%d\n",timep);
}
```

执行 time():974943297
time()->localtime()->mktime():974943297

settimeofday（设置目前时间）

相关函数 time, ctime, ftime, gettimeofday

表头文件 #include<sys/time.h>
#include<unistd.h>

定义函数 int settimeofday (const struct timeval *tv,const struct timezone *tz);

函数说明 settimeofday()会把目前时间设成由tv所指的结构信息，当地时区信息则设成tz所指的结构。详细的说明请参考gettimeofday()。注意，只有root权限才能使用此函数修改时间。

返回值 成功则返回0，失败返回-1，错误代码存于errno。

错误代码 EPERM 并非由root权限调用settimeofday（），权限不够。
EINVAL 时区或某个数据是不正确的，无法正确设置时间。

time（取得目前的时间）

相关函数 ctime, ftime, gettimeofday

表头文件 #include<time.h>

定义函数 time_t time(time_t *t);

函数说明 此函数会返回从公元1970年1月1日的UTC时间从0时0分0秒算起到现在所经过的秒数。如果t 并非空指针的话，此函数也会将返回值存到t指针所指的内存。

返回值 成功则返回秒数，失败则返回((time_t)-1)值，错误原因存于errno中。

```
范例 #include<time.h>
main()
{
    int seconds= time((time_t*)NULL);
    printf("%d\n",seconds);
}
```

}

执行 9.73E+08

bcmp (比较内存内容)

相关函数 bcmp, strcasecmp, strcmp, strcoll, strncmp, strncasecmp

表头文件 #include<string.h>

定义函数 int bcmp (const void *s1,const void * s2,int n);

函数说明 bcmp()用来比较s1和s2所指的内存区间前n个字节，若参数n为0，则返回0。

返回值 若参数s1 和s2 所指的内存内容都完全相同则返回0 值，否则返回非零值。

附加说明 建议使用memcmp()取代。

范例 参考memcmp()。

bcopy (拷贝内存内容)

相关函数 memcpy, memmove, strcpy, strncpy

表头文件 #include <string.h>

定义函数 void bcopy (const void *src,void *dest ,int n);

函数说明 bcopy()与memcpy()一样都是用来拷贝src所指的内存内容前n个字节到dest所指的地址，不过参数src与dest在传给函数时是相反的位置。

返回值

附加说明 建议使用memcpy()取代

范例 #include<string.h>

```
main()
{
    char dest[30]="string(a)";
    char src[30]="string\0string";
    int i;
    bcopy(src,dest,30);/* src指针放在前*/
    printf(bcopy(): ")
    for(i=0;i<30;i++)
        printf("%c",dest[i]);
    memcpy(dest src,30); /*dest指针放在钱*/
    printf("\nmemcpy() : ");
    for(i=0;i<30;i++)
        printf("%c",dest[i]);
```

执行 bcopy() : string string
memcpy() :string sring

bzero (将一段内存内容全清为零)

相关函数 memset, swab

表头文件 #include<string.h>

定义函数 void bzero(void *s,int n);

函数说明 bzero()会将参数s所指的内存区域前n个字节，全部设为零值。相当于调用memset((void*)s,0,size_tn);

返回值

附加说明 建议使用memset取代

范例 参考memset()。

index（查找字符串中第一个出现的指定字符）

相关函数 rindex, srechr, strchr

表头文件 #include<string.h>

定义函数 char * index(const char *s, int c);

函数说明 index()用来找出参数s字符串中第一个出现的参数c地址，然后将该字符出现的地址返回。字符串结束字符(NULL)也视为字符串一部分。

返回值 如果找到指定的字符则返回该字符所在地址，否则返回0。

```
范例 #include<string.h>
main()
{
    char *s = "0123456789012345678901234567890";
    char *p;
    p = index(s, '5');
    printf("%s\n", p);
}
```

执行 5.68E+25

memccpy（拷贝内存内容）

相关函数 bcopy, memcpy, memmove, strcpy, strncpy

表头文件 #include<string.h>

定义函数 void * memccpy(void *dest, const void * src, int c, size_t n);

函数说明 memccpy()用来拷贝src所指的内存内容前n个字节到dest所指的地址上。与memcpy()不同的是，memccpy()会在复制时检查参数c是否出现，若是则返回dest中值为c的下一个字节地址。

返回值 返回指向dest中值为c的下一个字节指针。返回值为0表示在src所指内存前n个字节中没有值为c的字节。

```
范例 #include<string.h>
main()
{
    char a[] = "string[a]";
    char b[] = "string(b)";
    memccpy(a, b, 'B', sizeof(b));
    printf("memccpy():%s\n", a);
}
```

执行 memccpy():string(b)

memchr（在某一内存范围中查找一特定字符）

相关函数 index, rindex, strchr, strpbrk, strrchr, strsep, strspn, strstr

表头文件 #include<string.h>

定义函数 void * memchr(const void *s, int c, size_t n);

函数说明 memchr()从头开始搜寻s所指的内存内容前n个字节，直到发现第一个值为c的字节，则返回指向该字节的指针。

返回值 如果找到指定的字节则返回该字节的指针，否则返回0。

```
范例 #include <string.h>
main()
{
    char *s="0123456789012345678901234567890";
    char *p;
    p=memchr(s,'5',10);
    printf("%s\n",p);
}
```

执行 5.68E+25

memcmp（比较内存内容）

相关函数 bcmp, strcasecmp, strcmp, strcoll, strncmp, strncasecmp

表头文件 #include<string.h>

定义函数 int memcmp (const void *s1,const void *s2,size_t n);

函数说明 memcmp()用来比较s1和s2所指的内存区间前n个字符。字符串大小的比较是以ASCII码表上的顺序来决定，次顺序亦为字符的值。memcmp()首先将s1第一个字符值减去s2第一个字符的值，若差为0则再继续比较下个字符，若差值不为0则将差值返回。例如，字符串"Ac"和"ba"比较则会返回字符'A'(65)和'b'(98)的差值(-33)。

返回值 若参数s1和s2所指的内存内容都完全相同则返回0值。s1若大于s2则返回大于0的值。s1若小于s2则返回小于0的值。

```
范例 #include<string.h>
main()
{
    char *a ="aBcDeF";
    char *b="AbCdEf";
    char *c="aacdef";
    char *d="aBcDeF";
    printf("memcmp(a,b):%d\n",memcmp((void*)a,(void*) b,6));
    printf("memcmp(a,c):%d\n",memcmp((void*)a,(void*) c,6));
    printf("memcmp(a,d):%d\n",memcmp((void*)a,(void*) d,6));
}
```

执行 memcmp(a,b):1 /*字符串a>字符串b，返回1*/
memcmp(a,c):-1 /* 字符串a<字符串c,返回-1*/
memcmp(a,d):0 /*字符串a=字符串d，返回0*/

memcpy（拷贝内存内容）

相关函数 bcopy, memcpy, memmove, strcpy, strncpy

表头文件 #include<string.h>

定义函数 void * memcpy (void * dest ,const void *src, size_t n);

函数说明 memcpy()用来拷贝src所指的内存内容前n个字节到dest所指的内存地址上。与strcpy()不同的是，memcpy()会完整的复制n个字节，不会因为遇到字符串结束'\0'而结束。

返回值 返回指向dest的指针。

附加说明 指针src和dest所指的内存区域不可重叠。

```
范例 #include<string.h>
main()
{
    char a[30]="string (a)";
}
```

```

char b[30]="string\0string";
int i;
strcpy(a,b);
printf("strcpy():");
for(i=0;i<30;i++)
printf("%c",a[i]);
memcpy(a,b,30);
printf("\nmemcpy() :");
for(i=0;i<30;i++)
printf("%c",a[i]);
}

```

执行 strcpy() : string a)
 memcpy() : string string

memmove（拷贝内存内容）

相关函数 bcopy, memccpy, memcpy, strcpy, strncpy

表头文件 #include<string.h>

定义函数 void * memmove(void *dest,const void *src,size_t n);

函数说明 memmove()与memcpy()一样都是用来拷贝src所指的内存内容前n个字节到dest所指的地址上。不同的是，当src和dest所指的内存区域重叠时，memmove()仍然可以正确的处理，不过执行效率上会比使用memcpy()略慢些。

返回值 返回指向dest的指针。

附加说明 指针src和dest所指的内存区域可以重叠。

范例 参考memcpy()。

memset（将一段内存空间填入某值）

相关函数 bzero, swab

表头文件 #include<string.h>

定义函数 void * memset (void *s ,int c, size_t n);

函数说明 memset()会将参数s所指的内存区域前n个字节以参数c填入，然后返回指向s的指针。在编写程序时，若需要将某一数组作初始化，memset()会相当方便。

返回值 返回指向s的指针。

附加说明 参数c虽声明为int，但必须是unsigned char，所以范围在0到255之间。

范例 #include <string.h>

```

main()
{
char s[30];
memset (s,'A',sizeof(s));
s[30]='\0';
printf("%s\n",s);
}

```

执行 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

rindex（查找字符串中最后一个出现的指定字符）

相关函数 index, memchr, strchr, strrchr

表头文件 `#include <string.h>`

定义函数 `char * rindex(const char *s,int c);`

函数说明 `rindex()`用来找出参数s字符串中最后一个出现的参数c地址，然后将该字符出现的地址返回。字符串结束字符(NULL)也视为字符串一部分。

返回值 如果找到指定的字符则返回该字符所在的地址，否则返回0。

```
范例 #include <string.h>
      main()
      {
        char *s ="0123456789012345678901234567890";
        char *p;
        p=rindex(s,'5');
        printf("%s\n",p);
      }
```

执行 567890

`strcasecmp`（忽略大小写比较字符串）

相关函数 `bcmp`，`memcmp`，`strcmp`，`strcoll`，`strncmp`

表头文件 `#include <string.h>`

定义函数 `int strcasecmp (const char *s1, const char *s2);`

函数说明 `strcasecmp()`用来比较参数s1和s2字符串，比较时会自动忽略大小写的差异。

返回值 若参数s1和s2字符串相同则返回0。s1长度大于s2长度则返回大于0 的值，s1 长度若小于s2 长度则返回小于0的值。

```
范例 #include <string.h>
      main()
      {
        char *a="aBcDeF";
        char *b="AbCdEf";
        if(!strcasecmp(a,b))
        printf("%s=%s\n",a,b);
      }
```

执行 aBcDeF=AbCdEf

`strcat`（连接两字符串）

相关函数 `bcopy`，`memccpy`，`memcpy`，`strcpy`，`strncpy`

表头文件 `#include <string.h>`

定义函数 `char *strcat (char *dest,const char *src);`

函数说明 `strcat()`会将参数src字符串拷贝到参数dest所指的字符串尾。第一个参数dest要有足够的空间来容纳要拷贝的字符串。

返回值 返回参数dest的字符串起始地址

```
范例 #include <string.h>
      main()
      {
        char a[30]="string(1)";
        char b[]="string(2)";
        printf("before strcat() : %s\n",a);
        printf("after strcat() : %s\n",strcat(a,b));
      }
```

执行 before strcat () : string(1)
after strcat () : string(1)string(2)

strchr (查找字符串中第一个出现的指定字符)

相关函数 index, memchr, rindex, strbrk, strsep, strspn, strstr, strtok

表头文件 #include <string.h>

定义函数 char * strchr (const char *s, int c);

函数说明 strchr()用来找出参数s字符串中第一个出现的参数c地址，然后将该字符出现的地址返回。

返回值 如果找到指定的字符则返回该字符所在地址，否则返回0。

```
范例 #include <string.h>
main()
{
    char *s="0123456789012345678901234567890";
    char *p;
    p=strchr(s,'5');
    printf("%s\n",p);
}
```

执行 5.68E+25

strcmp (比较字符串)

相关函数 bcmp, memcmp, strcasecmp, strncasecmp, strcoll

表头文件 #include <string.h>

定义函数 int strcmp(const char *s1, const char *s2);

函数说明 strcmp()用来比较参数s1和s2字符串。字符串大小的比较是以ASCII码表上的顺序来决定，此顺序亦为字符的值。strcmp()首先将s1第一个字符值减去s2第一个字符值，若差值为0则再继续比较下个字符，若差值不为0则将差值返回。例如字符串"Ac"和"ba"比较则会返回字符"A"(65)和'b'(98)的差值(-33)。

返回值 若参数s1和s2字符串相同则返回0。s1若大于s2则返回大于0的值。s1若小于s2则返回小于0的值。

```
范例 #include <string.h>
main()
{
    char *a="aBcDeF";
    char *b="AbCdEf";
    char *c="aacdef";
    char *d="aBcDeF";
    printf("strcmp(a,b) : %d\n",strcmp(a,b));
    printf("strcmp(a,c) : %d\n",strcmp(a,c));
    printf("strcmp(a,d) : %d\n",strcmp(a,d));
}
```

执行 strcmp(a,b) : 32
strcmp(a,c) : -31
strcmp(a,d) : 0

strcoll (采用目前区域的字符排列次序来比较字符串)

相关函数 strcmp, bcmp, memcmp, strcasecmp, strncasecmp

表头文件 `#include<string.h>`

定义函数 `int strcoll(const char *s1, const char *s2);`

函数说明 `strcoll()`会依环境变量`LC_COLLATE`所指定的文字排列次序来比较`s1`和`s2` 字符串。

返回值 若参数`s1`和`s2`字符串相同则返回0。`s1`若大于`s2`则返回大于0的值。`s1`若小于`s2`则返回小于0 的值。

附加说明 若`LC_COLLATE`为"POSIX"或"C"，则`strcoll()`与`strcmp()`作用完全相同。

范例 参考`strcmp()`。

`strcpy`（拷贝字符串）

相关函数 `bcopy`，`memcpy`，`memccpy`，`memmove`

表头文件 `#include<string.h>`

定义函数 `char *strcpy(char *dest,const char *src);`

函数说明 `strcpy()`会将参数`src`字符串拷贝至参数`dest`所指的地址。

返回值 返回参数`dest`的字符串起始地址。

附加说明 如果参数`dest`所指的内存空间不够大，可能会造成缓冲溢出(buffer Overflow)的错误情况，在编写程序时请特别留意，或者用`strncpy()`来取代。

```
范例 #include<string.h>
main()
{
    char a[30]="string(1)";
    char b[]="string(2)";
    printf("before strcpy() :%s\n",a);
    printf("after strcpy() :%s\n",strcpy(a,b));
}
```

执行 before strcpy() :string(1)
after strcpy() :string(2)

`strcspn`（返回字符串中连续不含指定字符串内容的字符数）

相关函数 `strspn`

表头文件 `#include<string.h>`

定义函数 `size_t strcspn (const char *s,const char * reject);`

函数说明 `strcspn()`从参数`s`字符串的开头计算连续的字符，而这些字符都完全不在参数`reject` 所指的字符串中。简单地说，若`strcspn()`返回的数值为`n`，则代表字符串`s`开头连续有`n`个字符都不含字符串`reject` 内的字符。

返回值 返回字符串`s`开头连续不含字符串`reject`内的字符数目。

```
范例 #include <string.h>
main()
{
    char *str="Linux was first developed for 386/486-based pcs.";
    printf("%d\n",strcspn(str," "));
    printf("%d\n",strcspn(str,"/-"));
    printf("%d\n",strcspn(str,"1234567890"));
}
```

执行 5 /*只计算到" "的出现，所以返回"Linux"的长度*/
33 /*计算到出现"/"或"-"，所以返回到"6"的长度*/
30 /* 计算到出现数字字符为止，所以返回"3"出现前的长度*/

strdup（复制字符串）

相关函数 calloc, malloc, realloc, free

表头文件 #include<string.h>

定义函数 char * strdup(const char *s);

函数说明 strdup()会先用malloc()配置与参数s字符串相同的空间大小，然后将参数s字符串的内容复制到该内存地址，然后把该地址返回。该地址最后可以利用free()来释放。

返回值 返回一字符串指针，该指针指向复制后的新字符串地址。若返回NULL表示内存不足。

```
范例 #include<string.h>
main()
{
    char a[]="strdup";
    char *b;
    b=strdup(a);
    printf("b[ ]=\"%s\\n",b);
}
```

执行 b[]="strdup"

strlen（返回字符串长度）

相关函数

表头文件 #include<string.h>

定义函数 size_t strlen (const char *s);

函数说明 strlen()用来计算指定的字符串s的长度，不包括结束字符"\0"。

返回值 返回字符串s的字符数。

```
范例 /*取得字符串str的长度*/
#include<string.h>
main()
{
    char *str = "12345678";
    printf("str length = %d\\n", strlen(str));
}
```

执行 str length = 8

strncasecmp（忽略大小写比较字符串）

相关函数 bcmp, memcmp, strcmp, strcoll, strncmp

表头文件 #include<string.h>

定义函数 int strncasecmp(const char *s1,const char *s2,size_t n);

函数说明 strncasecmp()用来比较参数s1和s2字符串前n个字符，比较时会自动忽略大小写的差异。

返回值 若参数s1和s2 字符串相同则返回0。s1 若大于s2则返回大于0的值，s1若小于s2则返回小于0 的值。

```
范例 #include<string.h>
main()
{
    char *a="aBcDeF";
    char *b="AbCdEf";
    if(!strncasecmp(a,b))
        printf("%s =%s\\n",a,b);
}
```

执行 aBcDef=AbCdEf

strncat（连接两字符串）

相关函数 bcopy, memccpy, memecpy, strcpy, strncpy

表头文件 #include <string.h>

定义函数 char * strncat(char *dest,const char *src,size_t n);

函数说明 strncat()会将参数src字符串拷贝n个字符到参数dest所指的字符串尾。第一个参数dest要有足够的空间来容纳要拷贝的字符串。

返回值 返回参数dest的字符串起始地址。

```
范例 #include <string.h>
main()
{
    char a[30]="string(1)";
    char b[]="string(2)";
    printf("before strncat() :%s\n", a);
    printf("after strncat() :%s\n", strncat(a,b,6));
}
```

执行 before strncat() : string(1)
after strncat() : string(1) string

strncpy（拷贝字符串）

相关函数 bcopy, memccpy, memcpy, memmove

表头文件 #include<string.h>

定义函数 char * strncpy(char *dest,const char *src,size_t n);

函数说明 strncpy()会将参数src字符串拷贝前n个字符至参数dest所指的地址。

返回值 返回参数dest的字符串起始地址。

```
范例 #include <string.h>
main()
{
    char a[30]="string(1)";
    char b[]="string(2)";
    printf("before strncpy() : %s\n",a);
    printf("after strncpy() : %s\n",strncpy(a,b,6));
}
```

执行 before strncpy() : string(1)
after strncpy() : string(1)

strpbrk（查找字符串中第一个出现的指定字符）

相关函数 index, memchr, rindex, strpbrk, strsep, strspn, strstr, strtok

表头文件 #include <string.h>

定义函数 char *strpbrk(const char *s,const char *accept);

函数说明 strpbrk()用来找出参数s字符串中最先出现存在参数accept字符串中的任意字符。

返回值 如果找到指定的字符则返回该字符所在地址，否则返回0。

```
范例 #include <string.h>
```

```

main()
{
char *s="0123456789012345678901234567890";
char *p;
p=strupbrk(s,"a1 839"); /*1会最先在s字符串中找到*/
printf("%s\n",p);
p=struprk(s,"4398"); /*3 会最先在s 字符串中找到*/
printf("%s\n",p);

```

执行 1.23E+29

strrchr（查找字符串中最后出现的指定字符）

相关函数 **index**, **memchr**, **rindex**, **strupbrk**, **strsep**, **strspn**, **strstr**, **strtok**

表头文件 **#include<string.h>**

定义函数 **char * strrchr(const char *s, int c);**

函数说明 **strrchr()**用来找出参数s字符串中最后一个出现的参数c地址，然后将该字符出现的地址返回。

返回值 如果找到指定的字符则返回该字符所在地址，否则返回0。

```

范例 #include<string.h>
main()
{
char *s="0123456789012345678901234567890";
char *p;
p=strrchr(s,'5');
printf("%s\n",p);
}

```

执行 567890

strspn（返回字符串中连续不含指定字符串内容的字符数）

相关函数 **strcspn**, **strchr**, **strupbrk**, **strsep**, **strstr**

表头文件 **#include<string.h>**

定义函数 **size_t strspn (const char *s,const char * accept);**

函数说明 **strspn()**从参数s 字符串的开头计算连续的字符，而这些字符都完全是**accept** 所指字符串中的字符。简单的说，若**strspn()**返回的数值为n，则代表字符串s 开头连续有n 个字符都是属于字符串**accept**内的字符。

返回值 返回字符串s开头连续包含字符串**accept**内的字符数目。

```

范例 #include<string.h>
main()
{
char *str="Linux was first developed for 386/486-based PCs.";
char *t1="abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
printf("%d\n",strspn(str,t1));
}

```

执行 5 /*计算大小写字母。不包含" "，所以返回Linux的长度。*/

strstr（在一字符串中查找指定的字符串）

相关函数 **index**, **memchr**, **rindex**, **strchr**, **strupbrk**, **strsep**, **strspn**, **strtok**

表头文件 `#include<string.h>`

定义函数 `char *strstr(const char *haystack,const char *needle);`

函数说明 `strstr()`会从字符串haystack 中搜寻字符串needle，并将第一次出现的地址返回。

返回值 返回指定字符串第一次出现的地址，否则返回0。

```
范例 #include<string.h>
main()
{
    char * s="012345678901234567890123456789";
    char *p;
    p= strstr(s,"901");
    printf("%s\n",p);
}
```

执行 9.01E+21

`strtok`（分割字符串）

相关函数 `index`，`memchr`，`rindex`，`strpbrk`，`strsep`，`strspn`，`strstr`

表头文件 `#include<string.h>`

定义函数 `char * strtok(char *s,const char *delim);`

函数说明 `strtok()`用来将字符串分割成一个个片段。参数s指向欲分割的字符串，参数delim则为分割字符串，当`strtok()`在参数s的字符串中发现到参数delim的分割字符时则会将该字符改为\0 字符。在第一次调用时，`strtok()`必需给予参数s字符串，往后的调用则将参数s设置成NULL。每次调用成功则返回下一个分割后的字符串指针。

返回值 返回下一个分割后的字符串指针，如果已无从分割则返回NULL。

```
范例 #include<string.h>
main()
{
    char s[]="ab-cd : ef;gh :i-jkl;mnop;qrs-tu: vwx-y;z";
    char *delim="-.: ";
    char *p;
    printf("%s ",strtok(s,delim));
    while((p=strtok(NULL,delim)))printf("%s ",p);
    printf("\n");
}
```

执行 ab cd ef;gh i jkl;mnop;qrs tu vwx y;z /*—与:字符已经被\0 字符取代*/

abs（计算整型数的绝对值）

相关函数 **labs**, **fabs**

表头文件 **#include <stdlib.h>**

定义函数 **int abs (int j)**

函数说明 **abs()**用来计算参数**j**的绝对值，然后将结果返回。

返回值 返回参数**j**的绝对值结果。

```
范例 #include <stdlib.h>
main(){
    int anser;
    answer = abs(-12);
    printf("|-12| = %d\n", answer);
}
```

执行 **|-12| = 12**

acos（取反余弦函数数值）

相关函数 **asin** , **atan** , **atan2** , **cos** , **sin** , **tan**

表头文件 **#include <math.h>**

定义函数 **double acos (double x);**

函数说明 **acos()**用来计算参数**x**的反余弦值，然后将结果返回。参数**x**范围为-1至1之间，超过此范围则会失败。

返回值 返回0至PI之间的计算结果，单位为弧度，在函数库中角度均以弧度来表示。

错误代码 **EDOM**参数**x**超出范围。

附加说明 使用GCC编译时请加入-lm。

```
范例 #include <math.h>
main (){
    double angle;
    angle = acos(0.5);
    printf("angle = %f\n", angle);
}
```

执行 **angle = 1.047198**

asin（取反正弦函数值）

相关函数 **acos** , **atan** , **atan2** , **cos** , **sin** , **tan**

表头文件 **#include <math.h>**

定义函数 **double asin (double x)**

函数说明 **asin()**用来计算参数**x**的反正弦值，然后将结果返回。参数**x**范围为-1至1之间，超过此范围则会失败。

返回值 返回-PI/2至PI/2之间的计算结果。

错误代码 **EDOM**参数**x**超出范围

附加说明 使用GCC编译时请加入-lm

范例 `#include<math.h>`
`main()`
`{`
`double angle;`
`angle = asin (0.5);`
`printf("angle = %f\n",angle);`
`}`

执行 `angle = 0.523599`

`atan`（取反正切函数值）

相关函数 `acos`, `asin`, `atan2`, `cos`, `sin`, `tan`

表头文件 `#include<math.h>`

定义函数 `double atan(double x);`

函数说明 `atan()`用来计算参数`x`的反正切值，然后将结果返回。

返回值 返回 $-\pi/2$ 至 $\pi/2$ 之间的计算结果。

附加说明 使用GCC编译时请加入`-lm`

范例 `#include<math.h>`
`main()`
`{`
`double angle;`
`angle =atan(1);`
`printf("angle = %f\n",angle);`
`}`

执行 `angle = 1.570796`

`atan2`（取得反正切函数值）

相关函数 `acos`, `asin`, `atan`, `cos`, `sin`, `tan`

表头文件 `#include<math.h>`

定义函数 `double atan2(double y,double x);`

函数说明 `atan2()`用来计算参数`y/x`的反正切值，然后将结果返回。

返回值 返回 $-\pi/2$ 至 $\pi/2$ 之间的计算结果。

附加说明 使用GCC编译时请加入`-lm`。

范例 `#include<math.h>`
`main()`
`{`
`double angle;`
`angle = atan2(1,2);`
`printf("angle = %f\n", angle);`
`}`

执行 `angle = 0.463648`

`ceil`（取不小于参数的最小整型数）

相关函数 `fabs`

表头文件 `#include <math.h>`

定义函数 `double ceil (double x);`

函数说明 `ceil()`会返回不小于参数x的最小整数值，结果以double形态返回。

返回值 返回不小于参数x的最小整数值。

附加说明 使用GCC编译时请加入-lm。

```
范例 #include<math.h>
main()
{
    double value[ ]={4.8,1.12,-2.2,0};
    int i;
    for (i=0;value[i]!=0;i++)
        printf("%f=>%f\n",value[i],ceil(value[i]));
}
```

```
执行 4.800000=>5.000000
      1.120000=>2.000000
      -2.200000=>-2.000000
```

`cos`（取余玄函数值）

相关函数 `acos`, `asin`, `atan`, `atan2`, `sin`, `tan`

表头文件 `#include<math.h>`

定义函数 `double cos(double x);`

函数说明 `cos()`用来计算参数x 的余玄值，然后将结果返回。

返回值 返回-1至1之间的计算结果。

附加说明 使用GCC编译时请加入-lm。

```
范例 #include<math.h>
main()
{
    double answer = cos(0.5);
    printf("cos (0.5) = %f\n",answer);
}
```

```
执行 cos(0.5) = 0.877583
```

`cosh`（取双曲线余玄函数值）

相关函数 `sinh`, `tanh`

表头文件 `#include<math.h>`

定义函数 `double cosh(double x);`

函数说明 `cosh()`用来计算参数x的双曲线余玄值，然后将结果返回。数学定义式为: $(\exp(x)+\exp(-x))/2$ 。

返回值 返回参数x的双曲线余玄值。

附加说明 使用GCC编译时请加入-lm。

```
范例 #include<math.h>
main()
{
    double answer = cosh(0.5);
    printf("cosh(0.5) = %f\n",answer);
}
```

```
执行 cosh(0.5) = 1.127626
```

exp (计算指数)

相关函数 log, log10, pow

表头文件 #include<math.h>

定义函数 double exp(double x);

函数说明 exp()用来计算以e为底的x次方值，即 e^x 值，然后将结果返回。

返回值 返回e的x次方计算结果。

附加说明 使用GCC编译时请加入-lm。

```
范例 #include<math.h>
main()
{
    double answer;
    answer = exp (10);
    printf("e^10 = %f\n", answer);
}
```

执行 $e^{10} = 22026.465795$

frexp (将浮点型数分为底数与指数)

相关函数 ldexp, modf

表头文件 #include<math.h>

定义函数 double frexp(double x, int *exp);

函数说明 frexp()用来将参数x 的浮点型数切割成底数和指数。底数部分直接返回，指数部分则借参数exp 指针返回，将返回值乘以2 的exp次方即为x的值。

返回值 返回参数x的底数部分，指数部分则存于exp指针所指的地址。

附加说明 使用GCC编译时请加入-lm。

```
范例 #include <math.h>
main()
{
    int exp;
    double fraction;
    fraction = frexp (1024,&exp);
    printf("exp = %d\n",exp);
    printf("fraction = %f\n", fraction);
}
```

执行 $exp = 11$
 $fraction = 0.500000 /* 0.5*(2^{11})=1024*/$

ldexp (计算2的次方值)

相关函数 frexp

表头文件 #include<math.h>

定义函数 double ldexp(double x,int exp);

函数说明 ldexp()用来将参数x乘以2的exp次方值，即 $x*2^{exp}$ 。

返回值 返回计算结果。

附加说明 使用GCC编译时请加入-lm。

```
范例: /* 计算3*(2^2)=12 */
#include<math.h>
```

```

main()
{
int exp;
double x,answer;
answer = ldexp(3,2);
printf("3*2^(2) = %f\n",answer);
}

```

执行 $3 \times 2^2 = 12.000000$

log（计算以e 为底的对数值）

相关函数 **exp**, **log10**, **pow**

表头文件 **#include <math.h>**

定义函数 **double log (double x);**

函数说明 **log ()** 用来计算以e为底的x 对数值，然后将结果返回。

返回值 返回参数x的自然对数值。

错误代码 **EDOM** 参数x为负数，**ERANGE** 参数x为零值，零的对数值无定义。

附加说明 使用GCC编译时请加入-lm。

```

范例 #include<math.h>
main()
{
double answer;
answer = log (100);
printf("log(100) = %f\n",answer);
}

```

执行 $\log(100) = 4.605170$

log10（计算以10 为底的对数值）

相关函数 **exp**, **log**, **pow**

表头文件 **#include<math.h>**

定义函数 **double log10(double x);**

函数说明 **log10()**用来计算以10为底的x对数值，然后将结果返回。

返回值 返回参数x以10为底的对数值。

错误代码 **EDOM**参数x为负数。**RANGE**参数x为零值，零的对数值无定义。

附加说明 使用GCC编译时请加入-lm。

```

范例 #include<math.h>
main()
{
double answer;
answer = log10(100);
printf("log10(100) = %f\n",answer);
}

```

执行 $\log_{10}(100) = 2.000000$

pow（计算次方值）

相关函数 `exp`, `log`, `log10`

表头文件 `#include <math.h>`

定义函数 `double pow(double x, double y);`

函数说明 `pow()` 用来计算以 `x` 为底的 `y` 次方值，即 x^y 值，然后将结果返回。

返回值 返回 `x` 的 `y` 次方计算结果。

错误代码 `EDOM` 参数 `x` 为负数且参数 `y` 不是整数。

附加说明 使用 GCC 编译时请加入 `-lm`。

```
范例 #include <math.h>
      main()
      {
        double answer;
        answer = pow(2,10);
        printf("2^10 = %f\n", answer);
      }
```

执行 `2^10 = 1024.000000`

`sin`（取正弦函数值）

相关函数 `acos`, `asin`, `atan`, `atan2`, `cos`, `tan`

表头文件 `#include <math.h>`

定义函数 `double sin(double x);`

函数说明 `sin()` 用来计算参数 `x` 的正弦值，然后将结果返回。

返回值 返回 -1 至 1 之间的计算结果。

附加说明 使用 GCC 编译时请加入 `-lm`。

```
范例 #include <math.h>
      main()
      {
        double answer = sin (0.5);
        printf("sin(0.5) = %f\n", answer);
      }
```

执行 `sin(0.5) = 0.479426`

`sinh`（取双曲线正弦函数值）

相关函数 `cosh`, `tanh`

表头文件 `#include <math.h>`

定义函数 `double sinh(double x);`

函数说明 `sinh()` 用来计算参数 `x` 的双曲线正弦值，然后将结果返回。数学定义式为： $(\exp(x) - \exp(-x))/2$ 。

返回值 返回参数 `x` 的双曲线正弦值。

附加说明 使用 GCC 编译时请加入 `-lm`。

```
范例 #include <math.h>
      main()
      {
        double answer = sinh (0.5);
        printf("sinh(0.5) = %f\n", answer);
      }
```

执行 `sinh(0.5) = 0.521095`

sqrt（计算平方根值）

相关函数 hypotq

表头文件 #include<math.h>

定义函数 double sqrt(double x);

函数说明 sqrt()用来计算参数x的平方根，然后将结果返回。参数x必须为正数。

返回值 返回参数x的平方根值。

错误代码 EDOM 参数x为负数。

附加说明 使用GCC编译时请加入-lm。

```
范例 /* 计算200的平方根值*/
#include<math.h>
main()
{
    double root;
    root = sqrt (200);
    printf("answer is %f\n",root);
}
```

执行 answer is 14.142136

tan（取正切函数值）

相关函数 atan, atan2, cos, sin

表头文件 #include <math.h>

定义函数 double tan(double x);

函数说明 tan()用来计算参数x的正切值，然后将结果返回。

返回值 返回参数x的正切值。

附加说明 使用GCC编译时请加入-lm。

```
范例 #include<math.h>
main()
{
    double answer = tan(0.5);
    printf("tan (0.5) = %f\n",answer);
}
```

执行 tan(0.5) = 0.546302

tanh（取双曲线正切函数值）

相关函数 cosh, sinh

表头文件 #include<math.h>

定义函数 double tanh(double x);

函数说明 tanh()用来计算参数x的双曲线正切值，然后将结果返回。数学定义式为:sinh(x)/cosh(x)。

返回值 返回参数x的双曲线正切值。

附加说明 使用GCC编译时请加入-lm。

```
范例 #include<math.h>
main()
{
    double answer = tanh(0.5);
```



```
printf("tanh(0.5) = %f\n",answer);  
}
```

执行 $\tanh(0.5) = 0.462117$

endgrent（关闭组文件）

相关函数 **getgrent**, **setgrent**

表头文件 `#include <grp.h>`
`#include <sys/types.h>`

定义函数 `void endgrent(void);`

函数说明 **endgrent()**用来关闭由**getgrent()**所打开的密码文件。

返回值

附加说明

范例 请参考**getgrent()**与**setgrent()**。

endpwent（关闭密码文件）

相关函数 **getpwent**, **setpwent**

表头文件 `#include <pwd.h>`
`#include <sys/types.h>`

定义函数 `void endpwent(void);`

函数说明 **endpwent()**用来关闭由**getpwent()**所打开的密码文件。

返回值

附加说明

范例 请参考**getpwent()**与**setpwent()**。

endutent（关闭utmp 文件）

相关函数 **getutent**, **setutent**

表头文件 `#include <utmp.h>`

定义函数 `void endutent(void);`

函数说明 **endutent()**用来关闭由**getutent**所打开的utmp文件。

返回值

附加说明

范例 请参考**getutent()**。

fgetgrent（从指定的文件来读取组格式）

相关函数 **fgetpwent**

表头文件 `#include <grp.h>`
`#include <stdio.h>`
`#include <sys/types.h>`

定义函数 `struct group * getgrent(FILE * stream);`

函数说明 **fgetgrent()**会从参数stream指定的文件读取一行数据，然后以group结构将该数据返回。参数stream所指定的文件必须和、etc/group相同的格式。group结构定义请参考**getgrent()**。

返回值 返回group结构数据，如果返回NULL则表示已无数据，或有错误发生。

```
范例 #include <grp.h>
#include<sys/types.h>
#include<stdio.h>
main()
{
    struct group *data;
    FILE *stream;
    int i;
    stream = fopen("/etc/group", "r");
    while((data = fgetgrent(stream))!=0){
        i=0;
        printf("%s :%s:%d :", data->gr_name,data->gr_passwd,data->gr_gid);
        while (data->gr_mem[i])printf("%s,",data->gr_mem[i++]);
        printf("\n");
    }
    fclose(stream);
}
```

执行 root:x:0:root,
bin:x:1:root,bin,daemon
daemon:x:2:root,bin,daemon
sys:x:3:root,bin,adm
adm:x:4:root,adm,daemon
tty:x:5
disk:x:6:root
lp:x:7:daemon,lp
mem:x:8
kmem:x:9
wheel:x:10:root
mail:x:12:mail
news:x:13:news
uucp:x:14:uucp
man:x:15
games:x:20
gopher:x:30
dip:x:40:
ftp:x:50
nobody:x:99:

fgetpwent（从指定的文件来读取密码格式）

相关函数 fgetgrent

表头文件 #include<pwd.h>
#include<stdio.h>
#include<sys/types.h>

定义函数 struct passwd * fgetpwent(FILE *stream);

函数说明 fgetpwent()会从参数stream指定的文件读取一行数据，然后以passwd结构将该数据返回。参数stream所指定的文件必须和/etc/passwd相同的格式。passwd结构定义请参考getpwent()。

返回值 返回passwd结构数据，如果返回NULL则表示已无数据，或有错误发生。

```
范例 #include<pwd.h>
#include<sys/types.h>
main()
{
    struct passwd *user;
```

```
FILE *stream;
stream = fopen("/etc/passwd", "r");
while((user = fgetpwent(stream))!=0){
printf("%s:%d:%d:%s:%s:%s\n",user->pw_name,user->pw_uid,user->pw_gid,user->pw_gecos,user->pw_dir,user->pw_shell);
}
}
```

执行 root:0:0:root:/root:/bin/bash
bin:1:1:bin:/bin:
daemon:2:2:daemon:/sbin:
adm:3:4:adm:/var/adm:
lp:4:7:lp:/var/spool/lpd:
sync:5:0:sync:/sbin:/bin/sync
shutdown:6:0:shutdown:/sbin:/sbin/shutdown
halt:7:0:halt:/sbin:/sbin/halt
mail:8:12:mail:/var/spool/mail:
news:9:13:news:/var/spool/news
uucp:10:14:uucp:/var/spool/uucp:
operator:11:0:operator :/root:
games:12:100:games:/usr/games:
gopher:13:30:gopher:/usr/lib/gopher-data:
ftp:14:50:FTP User:/home/ftp:
nobody:99:99:Nobody:/:
xfs:100:101:X Font Server: /etc/X11/fs:/bin/false
gdm:42:42:/home/gdm:/bin/bash
kids:500:500: : /home/kids:/bin/bash

getgid（取得有效的组织识别码）

相关函数 **getgid**, **setgid**, **setregid**

表头文件 `#include <unistd.h>`
`#include <sys/types.h>`

定义函数 `gid_t getgid(void);`

函数说明 **getgid()**用来取得执行目前进程有效组织识别码。有效的组织识别码用来决定进程执行时组的权限。返回值返回有效的组织识别码。

范例 `main()`

```
{
printf("egid is %d\n",getgid());
}
```

执行 `egid is 0` /*当使用root身份执行范例程序时*/

geteuid（取得有效的用户识别码）

相关函数 **getuid**, **setreuid**, **setuid**

表头文件 `#include <unistd.h>`
`#include <sys/types.h>`

定义函数 `uid_t geteuid(void)`

函数说明 **geteuid()**用来取得执行目前进程有效的用户识别码。有效的用户识别码用来决定进程执行的权限，借由此改变此值，进程可以获得额外的权限。倘若执行文件的**setID**位已被设置，该文件执行时，其进程的**euid**值便会设成该文件所有者的**uid**。例如，执行文件**/usr/bin/passwd**的权限为**-r-s--x--x**，其**s**位即为**setID(SUID)**位，而当任何用户在执行**passwd**时其有效的用户识别码会被设成**passwd**所有

者的uid 值，即root的uid 值(0)。

返回值 返回有效的用户识别码。

```
范例 main()
{
    printf ("euid is %d \n",geteuid());
}
```

执行 euid is 0 /*当使用root身份执行范例程序时*/

getgid（取得真实的组织识别码）

相关函数 getegid, setregid, setgid

表头文件 #include<unistd.h>
#include<sys/types.h>

定义函数 gid_t getgid(void);

函数说明 getgid()用来取得执行目前进程的组识别码。

返回值 返回组识别码

```
范例 main()
{
    printf("gid is %d\n",getgid());
}
```

执行 gid is 0 /*当使用root身份执行范例程序时*/

getgrent（从组文件中取得账号的数据）

相关函数 setgrent, endgrent

表头文件 #include<grp.h>
#include <sys/types.h>

定义函数 struct group *getgrent(void);

函数说明 getgrent()用来从组文件(/etc/group)中读取一项组数据，该数据以group 结构返回。第一次调用时会取得第一项组数据，之后每调用一次就会返回下一项数据，直到已无任何数据时返回NULL。

```
struct group{
    char *gr_name; /*组名称*/
    char *gr_passwd; /* 组密码*/
    gid_t gr_gid; /*组识别码*/
    char **gr_mem; /*组成员账号*/
}
```

返回值 返回group结构数据，如果返回NULL则表示已无数据，或有错误发生。

附加说明 getgrent()在第一次调用时会打开组文件，读取数据完毕后可使用endgrent()来关闭该组文件。

错误代码 ENOMEM 内存不足，无法配置group结构。

```
范例 #include<grp.h>
#include<sys/types.h>
main()
{
    struct group *data;
    int i;
    while((data= getgrent())!=0){
        i=0;
        printf("%s:%s:%d:",data->gr_name,data->gr_passwd,data->gr_gid);
        while(data->gr_mem[i])printf("%s,",data->gr_mem[i++]);
        printf("\n");
    }
```

```

}
endgrent();
}

```

执行 root:x:0:root,
 bin:x:1:root,bin,daemon,
 daemon:x:2:root,bin,daemon,
 sys:x:3:root,bin,adm,
 adm:x:4:root,adm,daemon
 tty:x:5
 disk:x:6:root
 lp:x:7:daemon,lp
 mem:x:8
 kmem:x:9:
 wheel:x:10:root
 mail:x:12:mail
 news:x:13:news
 uucp:x:14:uucp
 man:x:15:
 games:x:20
 gopher:x:30
 dip:x:40
 ftp:x:50
 nobody:x:99

getgrgid（从组文件中取得指定gid 的数据）

相关函数 fgetgrent， getgrent， getgrnam

表头文件 #include<grp.h>
 #include<sys/types.h>

定义函数 struct group * getgrgid(gid_t gid);

函数说明 getgrgid（）用来依参数gid指定的组织别码逐一搜索组文件，找到时便将该组的数据以group结构返回。group结构请参考getgrent（）。

返回值 返回group结构数据，如果返回NULL则表示已无数据，或有错误发生。

范例 /* 取得gid=3的组数据*/

```

#include<grp.h>
#include<sys/types.h>
main()
{
    struct group *data;
    int i=0;
    data = getgrgid(3);
    printf("%s:%s:%d:",data->gr_name,data->gr_passwd,data->gr_gid);
    while(data->gr_mem[i])printf("%s,",data->mem[i++]);
    printf("\n");
}

```

执行 sys:x:3:root,bin,adm

getgrnam（从组文件中取得指定组的数据）

相关函数 fgetgrent， getrent， getgruid

表头文件 `#include <grp.h>`

`#include <sys/types.h>`

定义函数 `strcut group * getgrnam(const char * name);`

函数说明 `getgrnam()` 用来逐一搜索参数那么指定的组名称，找到时便将该组的数据以 `group` 结构返回。
`group` 结构请参考 `getgrent()`。

返回值 返回 `group` 结构数据，如果返回 `NULL` 则表示已无数据，或有错误发生。

范例 /* 取得adm的组数据*/

```
#include <grp.h>
```

```
#include <sys/types.h>
```

```
main()
```

```
{
```

```
strcut group * data;
```

```
int i=0;
```

```
data = getgrnam("adm");
```

```
printf("%s:%s:%d:",data->gr_name,data->gr_passwd,data->gr_gid);
```

```
while(data->gr_mem[i])printf("%s,",data->gr_mem[i++]);
```

```
printf("\n");
```

```
}
```

执行 `adm:x:4:root,adm,daemon`

`getgroups`（取得组代码）

相关函数 `initgroups`, `setgroup`, `getgid`, `setgid`

表头文件 `#include <unistd.h>`

`#include <sys/types.h>`

定义函数 `int getgroups(int size,gid_t list[]);`

函数说明 `getgroup()` 用来取得目前用户所属的组代码。参数 `size` 为 `list()` 所能容纳的 `gid_t` 数目。如果参数 `size` 值为零，此函数仅会返回用户所属的组数。

返回值 返回组识别码，如有错误则返回 -1。

错误代码 `EFAULT` 参数 `list` 数组地址不合法。`EINVAL` 参数 `size` 值不足以容纳所有的组。

范例 `#include <unistd.h>`

```
#include <sys/types.h>
```

```
main()
```

```
{
```

```
gid_t list[500];
```

```
int x,i;
```

```
x = getgroups(0,list);
```

```
getgroups(x,list);
```

```
for(i=0;i<x;i++)
```

```
printf("%d:%d\n",i,list[i]);
```

```
}
```

执行 `0:00`

`1:01`

`2:02`

`3:03`

`4:04`

`5:06`

`6:10`

`getpw`（取得指定用户的密码文件数据）

相关函数 `getpwent`

表头文件 `#include <pwd.h>`
`#include <sys/types.h>`

定义函数 `int getpw(uid_t uid, char *buf);`

函数说明 `getpw()` 会从 `/etc/passwd` 中查找符合参数 `uid` 所指定的用户账号数据，找不到相关数据就返回 -1。所返回的 `buf` 字符串格式如下: 账号: 密码: 用户识别码(`uid`): 组识别码(`gid`): 全名: 根目录: shell

返回值 返回 0 表示成功，有错误发生时返回 -1。

附加说明 1. `getpw()` 会有潜在的安全性问题，请尽量使用别的函数取代。

2. 使用 shadow 的系统已把用户密码抽出 `/etc/passwd`，因此使用 `getpw()` 取得的密码将为 "x"。

范例 `#include <pwd.h>`
`#include <sys/types.h>`
`main()`
{
 `char buffer[80];`
 `getpw(0, buffer);`
 `printf("%s\n", buffer);`
}

执行 `root:x:0:0:root:/root:/bin/bash`

`getpwent`（从密码文件中取得账号的数据）

相关函数 `getpw`, `fgetpwent`, `getpwnam`, `getpwuid`, `setpwent`, `endpwent`

表头文件 `#include <pwd.h>`
`#include <sys/types.h>`

定义函数 `struct passwd * getpwent(void);`

函数说明 `getpwent()` 用来从密码文件（`/etc/passwd`）中读取一项用户数据，该用户的数据以 `passwd` 结构返回。第一次调用时会取得第一位用户数据，之后每调用一次就会返回下一项数据，直到已无任何数据时返回 `NULL`。

`passwd` 结构定义如下

```
struct passwd{
char * pw_name; /*用户账号*/
char * pw_passwd; /*用户密码*/
uid_t pw_uid; /*用户识别码*/
gid_t pw_gid; /*组识别码*/
char * pw_gecos; /*用户全名*/
char * pw_dir; /*家目录*/
char * pw_shell; /*所使用的shell路径*/
};
```

返回值 返回 `passwd` 结构数据，如果返回 `NULL` 则表示已无数据，或有错误发生。

附加说明 `getpwent()` 在第一次调用时会打开密码文件，读取数据完毕后可使用 `endpwent()` 来关闭该密码文件。错误代码 `ENOMEM` 内存不足，无法配置 `passwd` 结构。

范例 `#include <pwd.h>`
`#include <sys/types.h>`
`main()`
{
 `struct passwd *user;`
 while((`user = getpwent()`) != 0){
 `printf("%s:%d:%d:%s:%s:%s\n", user->pw_name, user->pw_uid, user->pw_gid,`
 `user->pw_gecos, user->pw_dir, user->pw_shell);`
 }
 `endpwent();`
}

执行 root:0:0:root:/root:/bin/bash
bin:1:1:bin:/bin:
daemon:2:2:daemon:/sbin:
adm:3:4:adm:/var/adm:
lp:4:7:lp:/var/spool/lpd:
sync:5:0:sync:/sbin:/bin/sync
shutdown:6:0:shutdown:/sbin:/sbin/shutdown
halt:7:0:halt:/sbin:/sbin/halt
mail:8:12:mail:/var/spool/mail:
news:9:13:news:/var/spool/news
uucp:10:14:uucp:/var/spool/uucp:
operator:11:0:operator :/root:
games:12:100:games:/usr/games:
gopher:13:30:gopher:/usr/lib/gopher-data:
ftp:14:50:FTP User:/home/ftp:
nobody:99:99:Nobody:/:
xfs:100:101:X Font Server: /etc/X11/fs:/bin/false
gdm:42:42:/home/gdm:/bin/bash
kids:500:500: : /home/kids:/bin/bash

getpwnam (从密码文件中取得指定账号的数据)

相关函数 getpw, fgetpwent, getpwent, getpwuid

表头文件 #include <pwd.h>

#include <sys/types.h>

定义函数 struct passwd * getpwnam(const char * name);

函数说明 getpwnam()用来逐一搜索参数name 指定的账号名称，找到时便将该用户的数据以passwd结构返回。passwd结构请参考getpwent()。

返回值 返回passwd 结构数据，如果返回NULL 则表示已无数据，或有错误发生。

范例 /*取得root账号的识别码和根目录*/

```
#include <pwd.h>
#include <sys/types.h>
main()
{
    struct passwd *user;
    user = getpwnam("root");
    printf("name:%s\n",user->pw_name);
    printf("uid:%d\n",user->pw_uid);
    printf("home:%s\n",user->pw_dir);
}
```

执行 name:root
uid:0
home:/root

getpwuid (从密码文件中取得指定uid 的数据)

相关函数 getpw, fgetpwent, getpwent, getpwnam

表头文件 #include <pwd.h>

#include <sys/types.h>

定义函数 struct passwd * getpwuid(uid_t uid);

函数说明 `getpwuid()`用来逐一搜索参数`uid` 指定的用户识别码，找到时便将该用户的数据以结构返回结构请参考将该用户的数据以`passwd` 结构返回。`passwd` 结构请参考`getpwent()`。

返回值 返回`passwd` 结构数据，如果返回`NULL` 则表示已无数据，或者有错误发生。

范例

```
#include <pwd.h>
#include <sys/types.h>
main()
{
    struct passwd *user;
    user = getpwuid(6);
    printf("name:%s\n", user->pw_name);
    printf("uid:%d\n", user->pw_uid);
    printf("home:%s\n", user->pw_dir);
}
```

执行

```
name:shutdown
uid:6
home:/sbin
```

`getuid`（取得真实的用户识别码）

相关函数 `geteuid`, `setreuid`, `setuid`

表头文件 `#include <unistd.h>`
`#include <sys/types.h>`

定义函数 `uid_t getuid(void);`

函数说明 `getuid()`用来取得执行目前进程的用户识别码。

返回值 用户识别码

范例

```
main()
{
    printf("uid is %d\n", getuid());
}
```

执行 `uid is 0` /*当使用root身份执行范例程序时*/

`getutent`（从`utmp` 文件中取得账号登录数据）

相关函数 `getutent`, `getutid`, `getutline`, `setutent`, `endutent`, `pututline`, `utmpname`

表头文件 `#include <utmp.h>`

定义函数 `struct utmp *getutent(void);`

函数说明 `getutent()`用来从`utmp` 文件(`/var/run/utmp`)中读取一项登录数据，该数据以`utmp` 结构返回。第一次调用时会取得第一位用户数据，之后每调用一次就会返回下一项数据，直到已无任何数据时返回`NULL`。

`utmp`结构定义如下

```
struct utmp
{
    short int ut_type; /*登录类型*/
    pid_t ut_pid; /*login进程的pid*/
    char ut_line[UT_LINESIZE]; /*登录装置名，省略了"/dev/"*/
    char ut_id[4]; /* Inittab ID*/
    char ut_user[UT_NAMESIZE]; /*登录账号*/
    char ut_host[UT_HOSTSIZE]; /*登录账号的远程主机名称*/
    struct exit_status ut_exit; /* 当类型为DEAD_PROCESS时进程的结束状态*/
    long int ut_session; /*Sessioc ID*/
}
```

```

struct timeval ut_tv; /*时间记录*/
int32_t ut_addr_v6[4]; /*远程主机的网络地址*/
char __unused[20]; /* 保留未使用*/
};
ut_type有以下几种类型:
EMPTY 此为空的记录。
RUN_LVL 记录系统run—level的改变
BOOT_TIME 记录系统开机时间
NEW_TIME 记录系统时间改变后的时间
OLD_TIME 记录当改变系统时间时的时间。
INIT_PROCESS 记录一个由init衍生出来的进程。
LOGIN_PROCESS 记录login进程。
USER_PROCESS 记录一般进程。
DEAD_PROCESS 记录一结束的进程。
ACCOUNTING 目前尚未使用。
exit_status结构定义
struct exit_status
{
short int e_termination; /*进程结束状态*/
short int e_exit; /*进程退出状态*/
};
timeval的结构定义请参考gettimeofday（）。
相关常数定义如下:
UT_LINESIZE 32
UT_NAMESIZE 32
UT_HOSTSIZE 256

```

返回值 返回utmp 结构数据，如果返回NULL 则表示已无数据，或有错误发生。

附加说明 getutent()在第一次调用时会打开utmp 文件，读取数据完毕后可使用endutent()来关闭该utmp文件。

范例 #include<utmp.h>

```

main()
{
struct utmp *u;
while((u=getutent())){
if(u->ut_type == USER_PROCESS)
printf("%d %s %s %s \n",u->ut_type,u->ut_user,u->ut_line,u->ut_host);
}
endutent();
}

```

执行 /* 表示有三个root账号分别登录/dev/pts/0, /dev/pts/1, /dev/pts/2 */

```

7 root pts/0
7 root pts/1
7 root pts/2

```

getutid（从utmp 文件中查找特定的记录）

相关函数 getutent, getutline

表头文件 #include<utmp.h>

定义函数 struct utmp *getutid(struct utmp *ut);

函数说明 getutid()用来从目前utmp 文件的读写位置逐一往后搜索参数ut指定的记录，如果ut->ut_type 为 RUN_LVL, BOOT_TIME, NEW_TIME, OLD_TIME 其中之一则查找与ut->ut_type 相符的记录；若 ut->ut_type 为INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS或DEAD_PROCESS其中之一，则

查找与ut->ut_id相符的记录。找到相符的记录便将该数据以utmp 结构返回。utmp结构请参考getutent()。

返回值 返回utmp 结构数据，如果返回NULL 则表示已无数据，或有错误发生。

```
范例 #include<utmp.h>
main()
{
    struct utmp ut,*u;
    ut.ut_type=RUN_LVL;
    while((u= getutid(&ut))){
        printf("%d %s %s %s\n",u->ut_type,u->ut_user,u->ut_line,u->ut_host);
    }
}
```

执行 1 runlevel -

getutline（从utmp 文件中查找特定的记录）

相关函数 getutent, getutid, pututline

表头文件 #include<utmp.h>

定义函数 struct utmp * getutline (struct utmp *ut);

函数说明 getutline()用来从目前utmp文件的读写位置逐一往后搜索ut_type为用户_PROCESS 或 LOGIN_PROCESS 的记录，而且ut_line 和ut->ut_line 相符。找到相符的记录便将该数据以utmp 结构返回， utmp结构请参考getutent()。

返回值 返回utmp 结构数据，如果返回NULL 则表示已无数据，或有错误发生。

```
范例 #include<utmp.h>
main()
{
    struct utmp ut,*u;
    strcpy (ut.ut_line,"pts/1");
    while ((u=getutline(&ut))){
        printf("%d %s %s %s \n",u->ut_type,u->ut_user,u->ut_line,u->ut_host);
    }
}
```

执行 7 root pts/1

initgroups（初始化组清单）

相关函数 setgrent, endgrent

表头文件 #include<grp.h>

#include<sys/types.h>

定义函数 int initgroups(const char *user,gid_t group);

函数说明 initgroups（）用来从组文件（/etc/group）中读取一项组数据，若该组数据的成员中有参数user 时，便将参数group组识别码加入到此数据中。

返回值 执行成功则返回0，失败则返回-1，错误码存于errno。

pututline（将utmp 记录写入文件）

相关函数 getutent, getutid, getutline

表头文件 `#include<utmp.h>`

定义函数 `void pututline(struct utmp *ut);`

函数说明 `pututline()`用来将参数`ut`的`utmp`结构记录到`utmp`文件中。此函数会先用`getutid()`来取得正确的写入位置，如果没有找到相符的记录则会加入到`utmp`文件尾，`utmp`结构请参考`getutent()`。

返回值

附加说明 需要有写入`/var/run/utmp` 的权限

```
范例 #include<utmp.h>
      main()
      {
        struct utmp ut;
        ut.ut_type =USER_PROCESS;
        ut.ut_pid=getpid();
        strcpy(ut.ut_user,"kids");
        strcpy(ut.ut_line,"pts/1");
        strcpy(ut.ut_host,"www.gnu.org");
        pututline(&ut);
      }
```

```
执行 /*执行范例后用指令who -l 观察*/
root pts/0 dec9 19:20
kids pts/1 dec12 10:31(www.gnu.org)
root pts/2 dec12 13:33
```

`seteuid`（设置有效的用户识别码）

相关函数 `setuid`，`setreuid`，`setfsuid`

表头文件 `#include<unistd.h>`

定义函数 `int seteuid(uid_t euid);`

函数说明 `seteuid()`用来重新设置执行目前进程的有效用户识别码。在Linux下，`seteuid(euid)`相当于`setreuid(-1,euid)`。

返回值 执行成功则返回0，失败则返回-1，错误代码存于`errno`

附加说明 请参考`setuid`

`setfsgid`（设置文件系统的组织识别码）

相关函数 `setuid`，`setreuid`，`seteuid`，`setfsuid`

表头文件 `#include<unistd.h>`

定义函数 `int setfsgid(uid_t fsgid);`

函数说明 `setfsgid()`用来重新设置目前进程的文件系统的组织识别码。一般情况下，文件系统的组织识别码(`fsgid`)与有效的组织识别码(`egid`)是相同的。如果是超级用户调用此函数，参数`fsgid`可以为任何值，否则参数`fsgid`必须为`real/effective/saved`的组织识别码之一。

返回值 执行成功则返回0，失败则返回-1，错误代码存于`errno`。

附加说明 此函数为Linux特有。

错误代码 `EPERM` 权限不够，无法完成设置。

`setfsuid`（设置文件系统的用户识别码）

相关函数 `setuid`，`setreuid`，`seteuid`，`setfsgid`

表头文件 `#include<unistd.h>`

定义函数 `int setfsuid(uid_t fsuid);`

函数说明 `setfsuid()`用来重新设置目前进程的文件系统的用户识别码。一般情况下，文件系统的用户识别码(`fsuid`)与有效的用户识别码(`uid`)是相同的。如果是超级用户调用此函数，参数`fsuid`可以为任何值，否则参数`fsuid`必须为`real/effective/saved`的用户识别码之一。

返回值 执行成功则返回0，失败则返回-1，错误代码存于`errno`

附加说明 此函数为Linux特有

错误代码 `EPERM` 权限不够，无法完成设置。

`setgid`（设置真实的组织识别码）

相关函数 `getgid`, `setregid`, `getegid`, `setegid`

表头文件 `#include<unistd.h>`

定义函数 `int setgid(gid_t gid);`

函数说明 `setgid()`用来将目前进程的真实组织识别码(`real gid`)设成参数`gid`值。如果是超级用户身份执行此调用，则`real`、`effective`与`savedgid`都会设成参数`gid`。

返回值 设置成功则返回0，失败则返回-1，错误代码存于`errno`中。

错误代码 `EPERM` 并非以超级用户身份调用，而且参数`gid`并非进程的`effective gid`或`saved gid`值之一。

`setgrent`（从头读取组文件中的组数据）

相关函数 `getgrent`, `endgrent`

表头文件 `#include<grp.h>`

`#include<sys/types.h>`

定义函数 `void setgrent(void);`

函数说明 `setgrent()`用来将`getgrent()`的读写地址指回组文件开头。

返回值

附加说明 请参考`setpwnent()`。

`setgroups`（设置组代码）

相关函数 `initgroups`, `getgroup`, `getgid`, `setgid`

表头文件 `#include<grp.h>`

定义函数 `int setgroups(size_t size,const gid_t * list);`

函数说明 `setgroups()`用来将`list`数组中所标明的组加入到目前进程的组设置中。参数`size`为`list()`的`gid_t`数目，最大值为`NGROUP(32)`。

返回值 设置成功则返回0，如有错误则返回-1。

错误代码 `EFAULT` 参数`list`数组地址不合法。

`EPERM` 权限不足，必须是`root`权限

`EINVAL` 参数`size`值大于`NGROUP(32)`。

`setpwnent`（从头读取密码文件中的账号数据）

相关函数 `getpwnent`, `endpwnent`

表头文件 `#include<pwd.h>`

```
#include<sys/types.h>
```

定义函数 `void setpwent(void);`

函数说明 `setpwent()`用来将`getpwent()`的读写地址指回密码文件开头。

返回值

范例 `#include<pwd.h>`

```
#include<sys/types.h>
```

```
main()
```

```
{
```

```
    struct passwd *user;
```

```
    int i;
```

```
    for(i=0;i<4;i++){
```

```
        user=getpwent();
```

```
        printf("%s :%d :%d :%s:%s:%s\n",user->pw_name,user->pw_uid,user->pw_gid,
```

```
        user->pw_gecos,user->pw_dir,user->pw_shell);
```

```
    }
```

```
    setpwent();
```

```
    user=getpwent();
```

```
    printf("%s :%d :%d :%s:%s:%s\n",user->pw_name,user->pw_uid,user->pw_gid,
```

```
    user->pw_gecos,user->pw_dir,user->pw_shell);
```

```
    endpwent();
```

```
}
```

执行 `root:0:0:root:/root:/bin/bash`

`bin:1:1:bin:/bin`

`daemon:2:2:daemon:/sbin`

`adm:3:4:adm:/var/adm`

`root:0:0:root:/root:/bin/bash`

`setregid`（设置真实及有效的组织识别码）

相关函数 `setgid`, `setegid`, `setfsuid`

表头文件 `#include<unistd.h>`

定义函数 `int setregid(gid_t rgid,gid_t egid);`

函数说明 `setregid()`用来将参数`rgid`设为目前进程的真实组织识别码，将参数`egid`设置为目前进程的有效组织识别码。如果参数`rgid`或`egid`值为-1，则对应的识别码不会改变。

返回值 执行成功则返回0，失败则返回-1，错误代码存于`errno`。

`setreuid`（设置真实及有效的用户识别码）

相关函数 `setuid`, `seteuid`, `setfsuid`

表头文件 `#include<unistd.h>`

定义函数 `int setreuid(uid_t ruid,uid_t euid);`

函数说明 `setreuid()`用来将参数`ruid` 设为目前进程的真实用户识别码，将参数`euid` 设置为目前进程的有效用户识别码。如果参数`ruid` 或`euid`值为-1，则对应的识别码不会改变。

返回值 执行成功则返回0，失败则返回-1，错误代码存于`errno`。

附加说明 请参考`setuid`（）。

`setuid`（设置真实的用户识别码）

相关函数 `getuid`, `setreuid`, `seteuid`, `setfsuid`

表头文件 `#include<unistd.h>`

定义函数 `int setuid(uid_t uid)`

函数说明 `setuid()`用来重新设置执行目前进程的用户识别码。不过，要让此函数有作用，其有效的用户识别码必须为0(root)。在Linux下，当root使用`setuid()`来变换成其他用户识别码时，root权限会被抛弃，完全转换成该用户身份，也就是说，该进程往后将不再具有可`setuid()`的权利，如果只是向暂时抛弃root 权限，稍后想重新取回权限，则必须使用`seteuid()`。

返回值 执行成功则返回0，失败则返回-1，错误代码存于`errno`。

附加说明 一般在编写具`setuid root`的程序时，为减少此类程序带来的系统安全风险，在使用完root权限后建议马上执行`setuid(getuid());`来抛弃root权限。此外，进程uid和euid不一致时Linux系统将不会产生core dump。

`setutent`（从头读取utmp 文件中的登录数据）

相关函数 `getutent`, `endutent`

表头文件 `#include<utmp.h>`

定义函数 `void setutent(void);`

函数说明 `setutent()`用来将`getutent()`的读写地址指回utmp文件开头。

附加说明 请参考`setpwent()`或`setgrent()`。

`utmpname`（设置utmp 文件路径）

相关函数 `getutent`, `getutid`, `getutline`, `setutent`, `endutent`, `pututline`

表头文件 `#include<utmp.h>`

定义函数 `void utmpname(const char * file);`

函数说明 `utmpname()`用来设置utmp文件的路径，以提供utmp相关函数的存取路径。如果没有使用`utmpname()`则默认utmp文件路径为/var/run/utmp。

返回值

crypt (将密码或数据编码)

相关函数 `getpass`

表头文件 `#define _XOPEN_SOURCE`
`#include <unistd.h>`

定义函数 `char * crypt (const char *key, const char * salt);`

函数说明 `crypt()`将使用Data Encryption Standard(DES)演算法将参数`key`所指的字符串加以编码, `key`字符串长度仅取前8个字符, 超过此长度的字符没有意义。参数`salt`为两个字符组成的字符串, 由a-z、A-Z、0-9, "."和"/"所组成, 用来决定使用4096 种不同内建表格的哪一个。函数执行成功后会返回指向编码过的字符串指针, 参数`key` 所指的字符串不会有所更动。编码过的字符串长度为13 个字符, 前两个字符为参数`salt`代表的字符串。

返回值 返回一个指向以NULL结尾的密码字符串。

附加说明 使用GCC编译时需加-lcrypt。

范例 `#include <unistd.h>`
`main()`
`{`
`char passwd[13];`
`char *key;`
`char slat[2];`
`key= getpass("Input First Password:");`
`slat[0]=key[0];`
`slat[1]=key[1];`
`strcpy(passwd,crypt(key slat));`
`key=getpass("Input Second Password:");`
`slat[0]=passwd[0];`
`slat[1]=passwd[1];`
`printf("After crypt(),1st passwd :%s\n",passwd);`
`printf("After crypt(),2nd passwd:%s \n",crypt(key slat));`
`}`

执行 Input First Password: /* 输入test, 编码后存于passwd[] */
Input Second Password /*输入test, 密码相同编码后也会相同*/
After crypt () 1st Passwd : teH0wLIpW0gyQ
After crypt () 2nd Passwd : teH0wLIpW0gyQ

bsearch (二元搜索)

相关函数 `qsort`

表头文件 `#include <stdlib.h>`

定义函数 `void *bsearch(const void *key, const void *base, size_t nmem, size_t size, int (*compar) (const void *, const void *));`

函数说明 `bsearch()`利用二元搜索从排序好的数组中查找数据。参数`key`指向欲查找的关键数据, 参数`base`指向要被搜索的数组开头地址, 参数`nmem` 代表数组中的元素数量, 每一元素的大小则由参数`size`决定, 最后一项参数`compar` 为一函数指针, 这个函数用来判断两个元素之间的大小关系, 若传给`compar` 的第一个参数所指的元素数据大于第二个参数所指的元素数据则必须回传大于0 的值, 两个元素数据相等则回传0。

附加说明 找到关键数据则返回找到的地址, 如果在数组中找不到关键数据则返回NULL。

```

范例 #include<stdio.h>
#include<stdlib.h>
#define NMEMB 5
#define SIZE 10
int compar(const void *a,const void *b)
{
return (strcmp((char *)a,(char *)b));
}
main()
{
char data[50][size]={“linux”,“freebsd”,“solaris”,“sunos”,“windows”};
char key[80],*base ,*offset;
int i, nmemb=NMEMB,size=SIZE;
while(1){
printf(">");
fgets(key,sizeof(key),stdin);
key[strlen(key)-1]='\0';
if(!strcmp(key,“exit”))break;
if(!strcmp(key,“list”)){
for(i=0;i<nmemb;i++)
printf(“%s\n”,data[i]);
continue;
}
base = data[0];
qsort(base,nmemb,size,compar);
offset = (char *) bsearch(key,base,nmemb,size,compar);
if( offset == NULL){
printf(“%s not found!\n”,key);
strcpy(data[nmemb++],key);
printf(“Add %s to data array\n”,key);
}else{
printf(“found: %s \n”,offset);
}
}
}

```

```

执行 >hello /*输入hello字符串*/
hello not found! /*找不到hello 字符串*/
add hello to data array /*将hello字符串加入*/
>.list /*列出所有数据*/
freebsd
linux
solaris
sunos
windows
hello
>hello
found: hello

```

lfind（线性搜索）

相关函数 lsearch

表头文件 #include<stdlib.h>

定义函数 void *lfind (const void *key,const void *base,size_t *nmemb,size_t size,int(* compar) (const void *,const void *));

函数说明 `lfind()`利用线性搜索在数组中从头至尾一项项查找数据。参数`key`指向欲查找的关键数据，参数`base`指向要被搜索的数组开头地址，参数`nmemb`代表数组中的元素数量，每一元素的大小则由参数`size`决定，最后一项参数`compar`为一函数指针，这个函数用来判断两个元素是否相同，若传给`compar`的异地个参数所指的元素数据和第二个参数所指的元素数据相同时则返回0，两个元素数据不相同则返回非0值。`lfind()`与`lsearch()`不同点在于，当找不到关键数据时`lfind()`仅会返回NULL，而不会主动把该笔数据加入数组尾端。

返回值 找到关键数据则返回找到的该笔元素的地址，如果在数组中找不到关键数据则返回空指针(NULL)。

范例 参考`lsearch()`。

`lsearch`（线性搜索）

相关函数 `lfind`

表头文件 `#include <stdlib.h>`

定义函数 `void *lsearch(const void * key ,const void * base ,size_t * nmemb,size_t size, int (* compar) (const void * ,const void *));`

函数说明 `lsearch()`利用线性搜索在数组中从头至尾一项项查找数据。参数`key`指向欲查找的关键数据，参数`base`指向要被搜索的数组开头地址，参数`nmemb`代表数组中的元素数量，每一元素的大小则由参数`size`决定，最后一项参数`compar`为一函数指针，这个函数用来判断两个元素是否相同，若传给`compar`的第一个参数所指的元素数据和第二个参数所指的元素数据相同时则返回0，两个元素数据不相同则返回非0值。如果`lsearch()`找不到关键数据时会主动把该项数据加入数组里。

返回值 找到关键数据则返回找到的该笔元素的地址，如果在数组中找不到关键数据则将此关键数据加入数组，再把加入数组后的地址返回。

范例

```
#include <stdio.h>
#include <stdlib.h>
#define NMEMB 50
#define SIZE 10
int compar (const void *a,const void *b)
{
    return (strcmp((char *) a, (char *) b));
}
main()
{
    char data[NMEMB][SIZE]={"Linux","freebsd","solzris","sunos","windows"};
    char key[80],*base,*offset;
    int i, nmemb=NMEMB,size=SIZE;
    for(i=1;i<5;i++){
        fgets(key,sizeof key,stdin);
        key[strlen(key)-1]='\0';
        base = data[0];
        offset = (char *)lfind(key,base,&nmemb,size,compar);
        if(offset ==NULL){
            printf("%s not found!\n",key);
            offset=(char *) lsearch(key,base,&nmemb,size,compar);
            printf("Add %s to data array\n",offset);
        }else{
            printf("found : %s \n",offset);
        }
    }
}
```

执行 `linux`

`found:linux`

`os/2`

`os/2 not found!`

```
add os/2 to data array
os/2
found:os/2
```

qsort（利用快速排序法排列数组）

相关函数 bsearch

表头文件 #include<stdlib.h>

定义函数 void qsort(void * base,size_t nmemb,size_t size,int (* compar)(const void *, const void *));

函数说明 参数base指向欲排序的数组开头地址，参数nmemb代表数组中的元素数量，每一元素的大小则由参数size决定，最后一项参数compar为一函数指针，这个函数用来判断两个元素间的大小关系，若传给compar的第一个参数所指的元素数据大于第二个参数所指的元素数据则必须回传大于零的值，两个元素数据相等则回传0。

返回值

附加说明

```
范例 #define nmemb 7
#include <stdlib.h>
int compar (const void *a ,const void *b)
{
    int *aa=(int * ) a,*bb = (int * )b;
    if( * aa >* bb)return 1;
    if( * aa == * bb) return 0;
    if( * aa < *bb) return -1;
}
main( )
{
    int base[nmemb]={ 3,102,5,-2,98,52,18};
    int i;
    for ( i=0; i<nmemb;i++)
        printf("%d ",base[i]);
    printf("\n");
    qsort(base,nmemb,sizeof(int),compar);
    for(i=0;i<nmemb;i++)
        printf("%d"base[i]);
    printf("\n");
}
```

```
执行 3 102 5 -2 98 52 18
      -2 3 5 18 52 98 102
```

rand（产生随机数）

相关函数 srand，random，srandom

表头文件 #include<stdlib.h>

定义函数 int rand(void)

函数说明 rand()会返回一随机数值，范围在0至RAND_MAX 间。在调用此函数产生随机数前，必须先利用srand()设好随机数种子，如果未设随机数种子，rand()在调用时会自动设随机数种子为1。关于随机数种子请参考srand()。

返回值 返回0至RAND_MAX之间的随机数值，RAND_MAX定义在stdlib.h，其值为2147483647。

```
范例 /* 产生介于1 到10 间的随机数值，此范例未设随机数种子，完整的随机数产生请参考
srand ( ) */
```

```

#include<stdlib.h>
main()
{
    int i,j;
    for(i=0;i<10;i++)
    {
        j=1+(int)(10.0*rand()/(RAND_MAX+1.0));
        printf("%d ",j);
    }
}

```

执行 9 4 8 8 10 2 4 8 3 6
9 4 8 8 10 2 4 8 3 6

srand（设置随机数种子）

相关函数 rand，random srandom

表头文件 #include<stdlib.h>

定义函数 void srand (unsigned int seed);

函数说明 srand()用来设置rand()产生随机数时的随机数种子。参数seed必须是个整数，通常可以利用geypid()或time(0)的返回值来当做seed。如果每次seed都设相同值，rand()所产生的随机数值每次就会一样。

返回值

范例 /* 产生介于1 到10 间的随机数值，此范例与执行结果可与rand（）参照*/

```

#include<time.h>
#include<stdlib.h>
main()
{
    int i,j;
    srand((int)time(0));
    for(i=0;i<10;i++)
    {
        j=1+(int)(10.0*rand()/(RAND_MAX+1.0));
        printf(" %d ",j);
    }
}

```

执行 5 8 8 8 10 2 10 8 9 9
2 9 7 4 10 3 2 10 8 7

close（关闭文件）

相关函数 open, fcntl, shutdown, unlink, fclose

表头文件 #include<unistd.h>

定义函数 int close(int fd);

函数说明 当使用完文件后若已不再需要则可使用close()关闭该文件，二close()会让数据写回磁盘，并释放该文件所占用的资源。参数fd为先前由open()或creat()所返回的文件描述词。

返回值 若文件顺利关闭则返回0，发生错误时返回-1。

错误代码 EBADF 参数fd 非有效的文件描述词或该文件已关闭。

附加说明 虽然在进程结束时，系统会自动关闭已打开的文件，但仍建议自行关闭文件，并确实检查返回值。

范例 参考open()

creat（建立文件）

相关函数 read, write, fcntl, close, link, stat, umask, unlink, fopen

表头文件 #include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

定义函数 int creat(const char * pathname, mode_t mode);

函数说明 参数pathname指向欲建立的文件路径字符串。Creat()相当于使用下列的调用方式调用open()
open(const char * pathname, (O_CREAT|O_WRONLY|O_TRUNC));

错误代码 关于参数mode请参考open（）函数。

返回值 creat()会返回新的文件描述词，若有错误发生则会返回-1，并把错误代码设给errno。

EEXIST 参数pathname所指的文件已存在。

EACCESS 参数pathname 所指定的文件不符合所要求测试的权限

EROFS 欲打开写入权限的文件存在于只读文件系统内

EFAULT 参数pathname 指针超出可存取的内存空间

EINVAL 参数mode 不正确。

ENAMETOOLONG 参数pathname太长。

ENOTDIR 参数pathname为一目录

ENOMEM 核心内存不足

ELOOP 参数pathname有过多符号连接问题。

EMFILE 已达到进程可同时打开的文件数上限

ENFILE 已达到系统可同时打开的文件数上限

附加说明 creat()无法建立特别的装置文件，如果需要请使用mknod()。

范例 请参考open()。

dup（复制文件描述词）

相关函数 open, close, fcntl, dup2

表头文件 #include<unistd.h>

定义函数 int dup (int oldfd);

函数说明 dup()用来复制参数oldfd所指的的文件描述词，并将它返回。此新的文件描述词和参数oldfd指的是同

一个文件，共享所有的锁定、读写位置和各项权限或旗标。例如，当利用lseek()对某个文件描述词作用时，另一个文件描述词的读写位置也会随着改变。不过，文件描述词之间并不共享close-on-exec旗标。

返回值 当复制成功时，则返回最小及尚未使用的文件描述词。若有错误则返回-1，errno会存放错误代码。错误代码EBADF参数fd非有效的文件描述词，或该文件已关闭。

dup2（复制文件描述词）

相关函数 open, close, fcntl, dup

表头文件 #include<unistd.h>

定义函数 int dup2(int oldfd,int newfd);

函数说明 dup2()用来复制参数oldfd所指的的文件描述词，并将它拷贝至参数newfd后一块返回。若参数newfd为一已打开的文件描述词，则newfd所指的的文件会先被关闭。dup2()所复制的文件描述词，与原来的文件描述词共享各种文件状态，详情可参考dup()。

返回值 当复制成功时，则返回最小及尚未使用的文件描述词。若有错误则返回-1，errno会存放错误代码。

附加说明 dup2()相当于调用fcntl(oldfd, F_DUPFD, newfd); 请参考fcntl()。

错误代码 EBADF 参数fd 非有效的文件描述词，或该文件已关闭

fcntl（文件描述词操作）

相关函数 open, flock

表头文件 #include<unistd.h>

#include<fcntl.h>

定义函数 int fcntl(int fd , int cmd);
int fcntl(int fd,int cmd,long arg);
int fcntl(int fd,int cmd,struct flock * lock);

函数说明 fcntl()用来操作文件描述词的一些特性。参数fd代表欲设置的文件描述词，参数cmd代表欲操作的指令。

有以下几种情况:

F_DUPFD用来查找大于或等于参数arg的最小且仍未使用的文件描述词，并且复制参数fd的文件描述词。执行成功则返回新复制的文件描述词。请参考dup2()。F_GETFD取得close-on-exec旗标。若此旗标的FD_CLOEXEC位为0，代表在调用exec()相关函数时文件将不会关闭。

F_SETFD 设置close-on-exec 旗标。该旗标以参数arg 的FD_CLOEXEC位决定。

F_GETFL 取得文件描述词状态旗标，此旗标为open（）的参数flags。

F_SETFL 设置文件描述词状态旗标，参数arg为新旗标，但只允许O_APPEND、O_NONBLOCK和O_ASYNC位的改变，其他位的改变将不受影响。

F_GETLK 取得文件锁定的状态。

F_SETLK 设置文件锁定的状态。此时flock 结构的l_type 值必须是F_RDLCK、F_WRLCK或F_UNLCK。

如果无法建立锁定，则返回-1，错误代码为EACCES 或EAGAIN。

F_SETLKW F_SETLK 作用相同，但是无法建立锁定时，此调用会一直等到锁定动作成功为止。若在等待锁定的过程中被信号中断时，会立即返回-1，错误代码为EINTR。参数lock指针为flock 结构指针，定义如下

```
struct flock
{
    short int l_type; /* 锁定的状态*/
    short int l_whence; /*决定l_start位置*/
    off_t l_start; /*锁定区域的开头位置*/
    off_t l_len; /*锁定区域的大小*/
    pid_t l_pid; /*锁定动作的进程*/
};
```

l_type 有三种状态:

F_RDLCK 建立一个供读取用的锁定
F_WRLCK 建立一个供写入用的锁定
F_UNLCK 删除之前建立的锁定
l_whence 也有三种方式:
SEEK_SET 以文件开头为锁定的起始位置。
SEEK_CUR 以目前文件读写位置为锁定的起始位置
SEEK_END 以文件结尾为锁定的起始位置。

返回值 成功则返回0, 若有错误则返回-1, 错误原因存于errno.

flock (锁定文件或解除锁定)

相关函数 open,fcntl

表头文件 #include<sys/file.h>

定义函数 int flock(int fd,int operation);

函数说明 flock()会依参数operation所指定的方式对参数fd所指的文件做各种锁定或解除锁定的动作。此函数只能锁定整个文件, 无法锁定文件的某一区域。

参数 operation有下列四种情况:

LOCK_SH 建立共享锁定。多个进程可同时对同一个文件作共享锁定。

LOCK_EX 建立互斥锁定。一个文件同时只有一个互斥锁定。

LOCK_UN 解除文件锁定状态。

LOCK_NB 无法建立锁定时, 此操作可不被阻断, 马上返回进程。通常与LOCK_SH或LOCK_EX 做OR()组合。

单一文件无法同时建立共享锁定和互斥锁定, 而当使用dup()或fork()时文件描述词不会继承此种锁定。

返回值 返回0表示成功, 若有错误则返回-1, 错误代码存于errno。

fsync (将缓冲区数据写回磁盘)

相关函数 sync

表头文件 #include<unistd.h>

定义函数 int fsync(int fd);

函数说明 fsync()负责将参数fd所指的文件数据, 由系统缓冲区写回磁盘, 以确保数据同步。

返回值 成功则返回0, 失败返回-1, errno为错误代码。

lseek (移动文件的读写位置)

相关函数 dup, open, fseek

表头文件 #include<sys/types.h>

#include<unistd.h>

定义函数 off_t lseek(int fildes,off_t offset ,int whence);

函数说明 每一个已打开的文件都有一个读写位置, 当打开文件时通常其读写位置是指向文件开头, 若是以附加的方式打开文件(如O_APPEND), 则读写位置会指向文件尾。当read()或write()时, 读写位置会随之增加, lseek()便是用来控制该文件的读写位置。参数fildes 为已打开的文件描述词, 参数offset 为根据参数whence来移动读写位置的位移数。

参数 whence为下列其中一种:

SEEK_SET 参数offset即为新的读写位置。

SEEK_CUR 以目前的读写位置往后增加offset个位移量。

SEEK_END 将读写位置指向文件尾后再增加offset个位移量。

当whence 值为SEEK_CUR 或SEEK_END时，参数offset允许负值的出现。

下列是教特别的使用方式:

- 1) 欲将读写位置移到文件开头时:lseek (int fildes,0,SEEK_SET) ;
- 2) 欲将读写位置移到文件尾时:lseek (int fildes, 0,SEEK_END) ;
- 3) 想要取得目前文件位置时:lseek (int fildes, 0,SEEK_CUR) ;

返回值 当调用成功时则返回目前的读写位置，也就是距离文件开头多少个字节。若有错误则返回-1，errno 会存放错误代码。

附加说明 Linux系统不允许lseek () 对tty装置作用，此项动作会令lseek () 返回ESPIPE。

范例 参考本函数说明

mkstemp (建立唯一的临时文件)

相关函数 mktemp

表头文件 #include<stdlib.h>

定义函数 int mkstemp(char * template);

函数说明 mkstemp()用来建立唯一的临时文件。参数template 所指的文件名称字符串中最后六个字符必须是XXXXXX。Mkstemp()会以可读写模式和0600 权限来打开该文件，如果该文件不存在则会建立该文件。打开该文件后其文件描述词会返回。文件顺利打开后返回可读写的文件描述词。若果文件打开失败则返回NULL，并把错误代码存在errno 中。

错误代码 EINVAL 参数template 字符串最后六个字符非XXXXXX。EEXIST 无法建立临时文件。

附加说明 参数template所指的文件名称字符串必须声明为数组，如:

```
char template[ ] = "template-XXXXXX";  
千万不可以使用下列的表达式  
char *template = "template-XXXXXX";
```

范例 #include<stdlib.h>

```
main( )  
{  
    int fd;  
    char template[ ] = "template-XXXXXX";  
    fd=mkstemp(template);  
    printf("template = %s\n",template);  
    close(fd);  
}
```

执行 template = template-lgZcbo

open (打开文件)

相关函数 read, write, fcntl, close, link, stat, umask, unlink, fopen

表头文件 #include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

定义函数 int open(const char * pathname, int flags);

int open(const char * pathname,int flags, mode_t mode);

函数说明 参数pathname 指向欲打开的文件路径字符串。下列是参数flags 所能使用的旗标:

O_RDONLY 以只读方式打开文件

O_WRONLY 以只写方式打开文件

O_RDWR 以可读写方式打开文件。上述三种旗标是互斥的，也就是不可同时使用，但可与下列的旗标利用OR()运算符组合。

O_CREAT 若欲打开的文件不存在则自动建立该文件。

O_EXCL 如果O_CREAT 也被设置，此指令会去检查文件是否存在。文件若不存在则建立该文件，否

则将导致打开文件错误。此外，若O_CREAT与O_EXCL同时设置，并且欲打开的文件为符号连接，则会打开文件失败。

O_NOCTTY 如果欲打开的文件为终端机设备时，则不会将该终端机当成进程控制终端机。

O_TRUNC 若文件存在并且以可写的方式打开时，此旗标会令文件长度清为0，而原来存于该文件的资料也会消失。

O_APPEND 当读写文件时会从文件尾开始移动，也就是所写入的数据会以附加的方式加入到文件后面。

O_NONBLOCK 以不可阻断的方式打开文件，也就是无论有无数据读取或等待，都会立即返回进程之中。

O_NDELAY 同O_NONBLOCK。

O_SYNC 以同步的方式打开文件。

O_NOFOLLOW 如果参数pathname所指的文件为一符号连接，则会令打开文件失败。

O_DIRECTORY 如果参数pathname所指的文件并非为一目录，则会令打开文件失败。

此为Linux2.2以后特有的旗标，以避免一些系统安全问题。参数mode则有下列数种组合，只有在建立新文件时才会生效，此外真正建文件时的权限会受到umask值所影响，因此该文件权限应该为(mode-umaks)。

S_IRWXU 000700 权限，代表该文件所有者具有可读、可写及可执行的权限。

S_IRUSR 或S_IREAD, 00400 权限，代表该文件所有者具有可读取的权限。

S_IWUSR 或S_IWRITE, 00200 权限，代表该文件所有者具有可写入的权限。

S_IXUSR 或S_IEXEC, 00100 权限，代表该文件所有者具有可执行的权限。

S_IRWXG 00070 权限，代表该文件用户组具有可读、可写及可执行的权限。

S_IRGRP 00040 权限，代表该文件用户组具有可读的权限。

S_IWGRP 00020 权限，代表该文件用户组具有可写入的权限。

S_IXGRP 00010 权限，代表该文件用户组具有可执行的权限。

S_IRWXO 00007 权限，代表其他用户具有可读、可写及可执行的权限。

S_IROTH 00004 权限，代表其他用户具有可读的权限

S_IWOTH 00002 权限，代表其他用户具有可写入的权限。

S_IXOTH 00001 权限，代表其他用户具有可执行的权限。

返回值 若所有欲核查的权限都通过了检查则返回0 值，表示成功，只要有一个权限被禁止则返回-1。

错误代码 EEXIST 参数pathname所指的文件已存在，却使用了O_CREAT和O_EXCL旗标。

EACCESS 参数pathname所指的文件不符合所要求测试的权限。

EROFS 欲测试写入权限的文件存在于只读文件系统内。

EFAULT 参数pathname指针超出可存取内存空间。

EINVAL 参数mode 不正确。

ENAMETOOLONG 参数pathname太长。

ENOTDIR 参数pathname不是目录。

ENOMEM 核心内存不足。

ELOOP 参数pathname有过多符号连接问题。

EIO I/O 存取错误。

附加说明 使用access()作用户认证方面的判断要特别小心，例如在access()后再作open()空文件可能会造成系统安全上的问题。

```
范例 #include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
main()
{
    int fd,size;
    char s [ ]="Linux Programmer!\n",buffer[80];
    fd=open("/tmp/temp",O_WRONLY|O_CREAT);
    write(fd,s,sizeof(s));
    close(fd);
    fd=open("/tmp/temp",O_RDONLY);
    size=read(fd,buffer,sizeof(buffer));
    close(fd);
    printf("%s",buffer);
}
```

}

执行 Linux Programmer!

read (由已打开的文件读取数据)

相关函数 `readdir`, `write`, `fcntl`, `close`, `lseek`, `readlink`, `fread`

表头文件 `#include <unistd.h>`

定义函数 `ssize_t read(int fd, void * buf, size_t count);`

函数说明 `read()` 会把参数 `fd` 所指的文件传送 `count` 个字节到 `buf` 指针所指的内存中。若参数 `count` 为 0, 则 `read()` 不会有作用并返回 0。返回值为实际读取到的字节数, 如果返回 0, 表示已到达文件尾或是无可读取的数据, 此外文件读写位置会随读取到的字节移动。

附加说明 如果顺利 `read()` 会返回实际读到的字节数, 最好能将返回值与参数 `count` 作比较, 若返回的字节数比要求读取的字节数少, 则有可能读到了文件尾、从管道(`pipe`)或终端机读取, 或者是 `read()` 被信号中断了读取动作。当有错误发生时则返回 -1, 错误代码存入 `errno` 中, 而文件读写位置则无法预期。

错误代码 `EINTR` 此调用被信号所中断。

`EAGAIN` 当使用不可阻断 I/O 时 (`O_NONBLOCK`), 若无数据可读取则返回此值。

`EBADF` 参数 `fd` 非有效的文件描述词, 或该文件已关闭。

范例 参考 `open()`。

sync (将缓冲区数据写回磁盘)

相关函数 `fsync`

表头文件 `#include <unistd.h>`

定义函数 `int sync(void)`

函数说明 `sync()` 负责将系统缓冲区数据写回磁盘, 以确保数据同步。

返回值 返回 0。

write (将数据写入已打开的文件内)

相关函数 `open`, `read`, `fcntl`, `close`, `lseek`, `sync`, `fsync`, `fwrite`

表头文件 `#include <unistd.h>`

定义函数 `ssize_t write (int fd, const void * buf, size_t count);`

函数说明 `write()` 会把参数 `buf` 所指的内存写入 `count` 个字节到参数 `fd` 所指的文件内。当然, 文件读写位置也会随之移动。

返回值 如果顺利 `write()` 会返回实际写入的字节数。当有错误发生时则返回 -1, 错误代码存入 `errno` 中。

错误代码 `EINTR` 此调用被信号所中断。

`EAGAIN` 当使用不可阻断 I/O 时 (`O_NONBLOCK`), 若无数据可读取则返回此值。

`EADF` 参数 `fd` 非有效的文件描述词, 或该文件已关闭。

范例 请参考 `open()`。

`clearerr`（清除文件流的错误旗标）

相关函数 `feof`

表头文件 `#include<stdio.h>`

定义函数 `void clearerr(FILE * stream);`

函数说明 `clearerr()` 清除参数`stream`指定的文件流所使用的错误旗标。

返回值

`fclose`（关闭文件）

相关函数 `close`, `fflush`, `fopen`, `setbuf`

表头文件 `#include<stdio.h>`

定义函数 `int fclose(FILE * stream);`

函数说明 `fclose()`用来关闭先前`fopen()`打开的文件。此动作会让缓冲区内的数据写入文件中，并释放系统所提供的文件资源。

返回值 若关文件动作成功则返回0，有错误发生时则返回EOF并把错误代码存到`errno`。

错误代码 EBADF表示参数`stream`非已打开的文件。

范例 请参考`fopen()`。

`fdopen`（将文件描述词转为文件指针）

相关函数 `fopen`, `open`, `fclose`

表头文件 `#include<stdio.h>`

定义函数 `FILE * fdopen(int fildes,const char * mode);`

函数说明 `fdopen()`会将参数`fildes`的文件描述词，转换为对应的文件指针后返回。参数`mode`字符串则代表着文件指针的流形态，此形态必须和原先文件描述词读写模式相同。关于`mode`字符串格式请参考`fopen()`。

返回值 转换成功时返回指向该流的文件指针。失败则返回NULL，并把错误代码存在`errno`中。

范例 `#include<stdio.h>`

```
main()
{
    FILE * fp =fdopen(0,"w+");
    fprintf(fp,"%s\n","hello!");
    fclose(fp);
}
```

执行 hello!

`feof`（检查文件流是否读到了文件尾）

相关函数 `fopen`, `fgetc`, `fgets`, `fread`

表头文件 `#include<stdio.h>`

定义函数 `int feof(FILE * stream);`

函数说明 `feof()`用来侦测是否读取到了文件尾，尾数`stream`为`fopen（）`所返回之文件指针。如果已到文件尾则返回非零值，其他情况返回0。

返回值 返回非零值代表已到达文件尾。

`fflush（更新缓冲区）`

相关函数 `write`, `fopen`, `fclose`, `setbuf`

表头文件 `#include<stdio.h>`

定义函数 `int fflush(FILE* stream);`

函数说明 `fflush()`会强迫将缓冲区内的数据写回参数`stream`指定的文件中。如果参数`stream`为NULL，`fflush()`会将所有打开的文件数据更新。

返回值 成功返回0，失败返回EOF，错误代码存于`errno`中。

错误代码 EBADF 参数`stream` 指定的文件未被打开，或打开状态为只读。其它错误代码参考`write（）`。

`fgetc（由文件中读取一个字符）`

相关函数 `open`, `fread`, `fscanf`, `getc`

表头文件 `include<stdio.h>`

定义函数 `nt fgetc(FILE * stream);`

函数说明 `fgetc()`从参数`stream`所指的文件中读取一个字符。若读到文件尾而无数据时便返回EOF。

返回值 `getc()`会返回读取到的字符，若返回EOF则表示到了文件尾。

范例 `#include<stdio.h>`

```
main()
{
    FILE *fp;
    int c;
    fp=fopen("exist","r");
    while((c=fgetc(fp))!=EOF)
        printf("%c",c);
    fclose(fp);
}
```

`fgets（由文件中读取一字符串）`

相关函数 `open`, `fread`, `fscanf`, `getc`

表头文件 `include<stdio.h>`

定义函数 `har * fgets(char * s,int size,FILE * stream);`

函数说明 `fgets()`用来从参数`stream`所指的文件内读入字符并存到参数`s`所指的内存空间，直到出现换行字符、读到文件尾或是已读了`size-1`个字符为止，最后会加上NULL作为字符串结束。

返回值 `gets()`若成功则返回`s`指针，返回NULL则表示有错误发生。

范例 `#include<stdio.h>`

```
main()
{
    char s[80];
    fputs(fgets(s,80,stdin),stdout);
}
```

执行 `this is a test /*输入*/`
`this is a test /*输出*/`

`fileno`（返回文件流所使用的文件描述词）

相关函数 `open`, `fopen`

表头文件 `#include <stdio.h>`

定义函数 `int fileno(FILE * stream);`

函数说明 `fileno()`用来取得参数`stream`指定的文件流所使用的文件描述词。

返回值 返回文件描述词。

范例 `#include <stdio.h>`
`main()`
`{`
`FILE * fp;`
`int fd;`
`fp=fopen("/etc/passwd","r");`
`fd=fileno(fp);`
`printf("fd=%d\n",fd);`
`fclose(fp);`
`}`

执行 `fd=3`

`fopen`（打开文件）

相关函数 `open`, `fclose`

表头文件 `#include <stdio.h>`

定义函数 `FILE * fopen(const char * path,const char * mode);`

函数说明 参数`path`字符串包含欲打开的文件路径及文件名，参数`mode`字符串则代表着流形态。

`mode`有下列几种形态字符串：

`r` 打开只读文件，该文件必须存在。

`r+` 打开可读写的文件，该文件必须存在。

`w` 打开只写文件，若文件存在则文件长度清为0，即该文件内容会消失。若文件不存在则建立该文件。

`w+` 打开可读写文件，若文件存在则文件长度清为零，即该文件内容会消失。若文件不存在则建立该文件。

`a` 以附加的方式打开只写文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾，即文件原先的内容会被保留。

`a+` 以附加方式打开可读写的文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾后，即文件原先的内容会被保留。

上述的形态字符串都可以再加一个**b**字符，如**rb**、**w+b**或**ab+**等组合，加入**b**字符用来告诉函数库打开的文件为二进制文件，而非纯文字文件。不过在POSIX系统，包含Linux都会忽略该字符。由

`fopen()`所建立的新文件会具有S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH(0666)权限，此文件权限也会参考`umask`值。

返回值 文件顺利打开后，指向该流的文件指针就会被返回。若果文件打开失败则返回NULL，并把错误代码存在`errno`中。

附加说明 一般而言，开文件后会作一些文件读取或写入的动作，若开文件失败，接下来的读写动作也无法顺利进行，所以在`fopen()`后请作错误判断及处理。

范例 `#include <stdio.h>`
`main()`
`{`

```
FILE * fp;
fp=fopen("noexist","a+");
if(fp==NULL) return;
fclose(fp);
}
```

fputc（将一指定字符写入文件流中）

相关函数 fopen, fwrite, fscanf, putc

表头文件 #include<stdio.h>

定义函数 int fputc(int c, FILE * stream);

函数说明 fputc 会将参数c 转为unsigned char 后写入参数stream 指定的文件中。

返回值 fputc()会返回写入成功的字符，即参数c。若返回EOF则代表写入失败。

范例 #include<stdio.h>

```
main()
{
FILE * fp;
char a[26]="abcdefghijklmnopqrstuvwxyz";
int i;
fp= fopen("noexist","w");
for(i=0;i<26;i++)
fputc(a[i],fp);
fclose(fp);
}
```

fputs（将一指定的字符串写入文件内）

相关函数 fopen, fwrite, fscanf, fputc, putc

表头文件 #include<stdio.h>

定义函数 int fputs(const char * s, FILE * stream);

函数说明 fputs()用来将参数s所指的字符串写入到参数stream所指的文件内。

返回值 若成功则返回写出的字符个数，返回EOF则表示有错误发生。

范例 请参考fgets（）。

fread（从文件流读取数据）

相关函数 fopen, fwrite, fseek, fscanf

表头文件 #include<stdio.h>

定义函数 size_t fread(void * ptr, size_t size, size_t nmemb, FILE * stream);

函数说明 fread()用来从文件流中读取数据。参数stream为已打开的文件指针，参数ptr 指向欲存放读取进来的数据空间，读取的字符数以参数size*nmemb来决定。Fread()会返回实际读取到的nmemb数目，如果此值比参数nmemb 来得小，则代表可能读到了文件尾或有错误发生，这时必须用feof()或ferror()来决定发生什么情况。

返回值 返回实际读取到的nmemb数目。

附加说明

范例 #include<stdio.h>
#define nmemb 3
struct test

```

{
char name[20];
int size;
}s[nmemb];
main()
{
FILE * stream;
int i;
stream = fopen("/tmp/fwrite","r");
fread(s,sizeof(struct test),nmemb,stream);
fclose(stream);
for(i=0;i<nmemb;i++)
printf("name[%d]=%-20s:size[%d]=%d\n",i,s[i].name,i,s[i].size);
}

```

执行 name[0]=Linux! size[0]=6
name[1]=FreeBSD! size[1]=8
name[2]=Windows2000 size[2]=11

freopen（打开文件）

相关函数 fopen, fclose

表头文件 #include<stdio.h>

定义函数 FILE * freopen(const char * path,const char * mode,FILE * stream);

函数说明 参数path字符串包含欲打开的文件路径及文件名，参数mode请参考fopen()说明。参数stream为已打开的文件指针。Freopen()会将原stream所打开的文件流关闭，然后打开参数path的文件。

返回值 文件顺利打开后，指向该流的文件指针就会被返回。如果文件打开失败则返回NULL，并把错误代码存在errno 中。

范例 #include<stdio.h>

```

main()
{
FILE * fp;
fp=fopen("/etc/passwd","r");
fp=freopen("/etc/group","r",fp);
fclose(fp);
}

```

fseek（移动文件流的读写位置）

相关函数 rewind, ftell, fgetpos, fsetpos, lseek

表头文件 #include<stdio.h>

定义函数 int fseek(FILE * stream,long offset,int whence);

函数说明 fseek()用来移动文件流的读写位置。参数stream为已打开的文件指针，参数offset为根据参数whence来移动读写位置的位移数。

参数 whence为下列其中一种:

SEEK_SET从距文件开头offset位移量为新的读写位置。SEEK_CUR 以目前的读写位置往后增加offset个位移量。

SEEK_END将读写位置指向文件尾后再增加offset个位移量。

当whence值为SEEK_CUR 或SEEK_END时，参数offset允许负值的出现。

下列是较特别的使用方式:

- 1) 欲将读写位置移动到文件开头时:fseek(FILE *stream,0,SEEK_SET);
- 2) 欲将读写位置移动到文件尾时:fseek(FILE *stream,0,SEEK_END);

返回值 当调用成功时则返回0，若有错误则返回-1，`errno`会存放错误代码。

附加说明 `fseek()`不像`lseek()`会返回读写位置，因此必须使用`ftell()`来取得目前读写的位置。

```
范例 #include<stdio.h>
      main()
      {
        FILE * stream;
        long offset;
        fpos_t pos;
        stream=fopen("/etc/passwd","r");
        fseek(stream,5,SEEK_SET);
        printf("offset=%d\n",ftell(stream));
        rewind(stream);
        fgetpos(stream,&pos);
        printf("offset=%d\n",pos);
        pos=10;
        fsetpos(stream,&pos);
        printf("offset = %d\n",ftell(stream));
        fclose(stream);
      }
```

执行 offset = 5
offset =0
offset=10

`ftell`（取得文件流的读取位置）

相关函数 `fseek`，`rewind`，`fgetpos`，`fsetpos`

表头文件 `#include<stdio.h>`

定义函数 `long ftell(FILE * stream);`

函数说明 `ftell()`用来取得文件流目前的读写位置。参数`stream`为已打开的文件指针。

返回值 当调用成功时则返回目前的读写位置，若有错误则返回-1，`errno`会存放错误代码。

错误代码 `EBADF` 参数`stream`无效或可移动读写位置的文件流。

范例 参考`fseek()`。

`fwrite`（将数据写至文件流）

相关函数 `fopen`，`fread`，`fseek`，`fscanf`

表头文件 `#include<stdio.h>`

定义函数 `size_t fwrite(const void * ptr,size_t size,size_t nmemb,FILE * stream);`

函数说明 `fwrite()`用来将数据写入文件流中。参数`stream`为已打开的文件指针，参数`ptr`指向欲写入的数据地址，总共写入的字符数以参数`size*nmemb`来决定。`Fwrite()`会返回实际写入的`nmemb`数目。

返回值 返回实际写入的`nmemb`数目。

```
范例 #include<stdio.h>
      #define set_s(x,y) {strcpy(s[x].name,y);s[x].size=strlen(y);}
      #define nmemb 3
      struct test
      {
        char name[20];
        int size;
      }s[nmemb];
```

```

main()
{
    FILE * stream;
    set_s(0,"Linux!");
    set_s(1,"FreeBSD!");
    set_s(2,"Windows2000.");
    stream=fopen("/tmp/fwrite","w");
    fwrite(s,sizeof(struct test),nmemb,stream);
    fclose(stream);
}

```

执行 参考fread（）。

getc（由文件中读取一个字符）

相关函数 read, fopen, fread, fgetc

表头文件 #include<stdio.h>

定义函数 int getc(FILE * stream);

函数说明 getc()用来从参数stream所指的文件中读取一个字符。若读到文件尾而无数据时便返回EOF。虽然getc()与fgetc()作用相同，但getc()为宏定义，非真正的函数调用。

返回值 getc()会返回读取到的字符，若返回EOF则表示到了文件尾。

范例 参考fgetc()。

getchar（由标准输入设备内读进一字符）

相关函数 fopen, fread, fscanf, getc

表头文件 #include<stdio.h>

定义函数 int getchar(void);

函数说明 getchar()用来从标准输入设备中读取一个字符。然后将该字符从unsigned char转换成int后返回。

返回值 getchar()会返回读取到的字符，若返回EOF则表示有错误发生。

附加说明 getchar()非真正函数，而是getc(stdin)宏定义。

范例 #include<stdio.h>

```

main()
{
    FILE * fp;
    int c,i;
    for(i=0;i<5;i++)
    {
        c=getchar();
        putchar(c);
    }
}

```

执行 1234 /*输入*/
1234 /*输出*/

gets（由标准输入设备内读进一字符串）

相关函数 fopen, fread, fscanf, fgetc

表头文件 #include<stdio.h>

定义函数 `char * gets(char *s);`

函数说明 `gets()`用来从标准设备读入字符并存到参数s所指的内存空间，直到出现换行字符或读到文件尾为止，最后加上NULL作为字符串结束。

返回值 `gets()`若成功则返回s指针，返回NULL则表示有错误发生。

附加说明 由于`gets()`无法知道字符串s的大小，必须遇到换行字符或文件尾才会结束输入，因此容易造成缓冲溢出的安全性问题。建议使用`fgets()`取代。

范例 参考`fgets()`

`mktemp`（产生唯一的临时文件名）

相关函数 `tmpfile`

表头文件 `#include <stdlib.h>`

定义函数 `char * mktemp(char * template);`

函数说明 `mktemp()`用来产生唯一的临时文件名。参数`template`所指的文件名称字符串中最后六个字符必须是XXXXXX。产生后的文件名会借字符串指针返回。

返回值 文件顺利打开后，指向该流的文件指针就会被返回。如果文件打开失败则返回NULL，并把错误代码存在`errno`中。

附加说明 参数`template`所指的文件名称字符串必须声明为数组，如：

```
char template[]="template-XXXXXX";  
不可用char * template="template-XXXXXX";
```

```
范例 #include <stdlib.h>  
main()  
{  
    char template[]="template-XXXXXX";  
    mktemp(template);  
    printf("template=%s\n",template);  
}
```

`putc`（将一指定字符写入文件中）

相关函数 `fopen`, `fwrite`, `fscanf`, `fputc`

表头文件 `#include <stdio.h>`

定义函数 `int putc(int c,FILE * stream);`

函数说明 `putc()`会将参数c转为unsigned char后写入参数stream指定的文件中。虽然`putc()`与`fputc()`作用相同，但`putc()`为宏定义，非真正的函数调用。

返回值 `putc()`会返回写入成功的字符，即参数c。若返回EOF则代表写入失败。

范例 参考`fputc()`。

`putchar`（将指定的字符写到标准输出设备）

相关函数 `fopen`, `fwrite`, `fscanf`, `fputc`

表头文件 `#include <stdio.h>`

定义函数 `int putchar (int c);`

函数说明 `putchar()`用来将参数c字符写到标准输出设备。

返回值 `putchar()`会返回输出成功的字符，即参数c。若返回EOF则代表输出失败。

附加说明 `putchar()`非真正函数，而是`putc(c, stdout)`宏定义。

范例 参考getchar()。

rewind（重设文件流的读写位置为文件开头）

相关函数 fseek, ftell, fgetpos, fsetpos

表头文件 #include<stdio.h>

定义函数 void rewind(FILE * stream);

函数说明 rewind()用来把文件流的读写位置移至文件开头。参数stream为已打开的文件指针。此函数相当于调用fseek(stream,0,SEEK_SET)。

返回值

范例 参考fseek()

setbuf（设置文件流的缓冲区）

相关函数 setbuffer, setlinebuf, setvbuf

表头文件 #include<stdio.h>

定义函数 void setbuf(FILE * stream,char * buf);

函数说明 在打开文件流后，读取内容之前，调用setbuf()可以用来设置文件流的缓冲区。参数stream为指定的文件流，参数buf指向自定的缓冲区起始地址。如果参数buf为NULL指针，则为无缓冲IO。Setbuf()相当于调用:setvbuf(stream,buf,buf?_IOFBF:_IONBF,BUFSIZ)

返回值

setbuffer（设置文件流的缓冲区）

相关函数 setlinebuf, setbuf, setvbuf

表头文件 #include<stdio.h>

定义函数 void setbuffer(FILE * stream,char * buf,size_t size);

函数说明 在打开文件流后，读取内容之前，调用setbuffer()可用来设置文件流的缓冲区。参数stream为指定的文件流，参数buf指向自定的缓冲区起始地址，参数size为缓冲区大小。

返回值

setlinebuf（设置文件流为线性缓冲区）

相关函数 setbuffer, setbuf, setvbuf

表头文件 #include<stdio.h>

定义函数 void setlinebuf(FILE * stream);

函数说明 setlinebuf()用来设置文件流以换行为依据的无缓冲IO。相当于调用:setvbuf(stream,(char *)NULL,_IOLBF,0);请参考setvbuf()。

返回值

setvbuf（设置文件流的缓冲区）

相关函数 setbuffer, setlinebuf, setbuf

表头文件 #include<stdio.h>

定义函数 `int setvbuf(FILE * stream, char * buf, int mode, size_t size);`

函数说明 在打开文件流后，读取内容之前，调用`setvbuf()`可以用来设置文件流的缓冲区。参数`stream`为指定的文件流，参数`buf`指向自定的缓冲区起始地址，参数`size`为缓冲区大小，参数`mode`有下列几种

`_IONBF` 无缓冲IO

`_IOLBF` 以换行为依据的无缓冲IO

`_IOFBF` 完全无缓冲IO。如果参数`buf`为NULL指针，则为无缓冲IO。

返回值

`ungetc`（将指定字符写回文件流中）

相关函数 `fputc`，`getchar`，`getc`

表头文件 `#include <stdio.h>`

定义函数 `int ungetc(int c, FILE * stream);`

函数说明 `ungetc()`将参数`c`字符写回参数`stream`所指定的文件流。这个写回的字符会由下一个读取文件流的函数取得。

返回值 成功则返回`c` 字符，若有错误则返回`EOF`。

atexit（设置程序正常结束前调用的函数）

相关函数 `_exit`, `exit`, `on_exit`

表头文件 `#include <stdlib.h>`

定义函数 `int atexit (void (*function)(void));`

函数说明 `atexit()`用来设置一个程序正常结束前调用的函数。当程序通过调用`exit()`或从`main`中返回时，参数`function`所指定的函数会先被调用，然后才真正由`exit()`结束程序。

返回值 如果执行成功则返回0，否则返回-1，失败原因存于`errno`中。

```
范例 #include <stdlib.h>
      void my_exit(void)
      {
        printf("before exit () !\n");
      }
      main()
      {
        atexit (my_exit);
        exit(0);
      }
```

执行 before exit()!

execl（执行文件）

相关函数 `fork`, `execle`, `execlp`, `execv`, `execve`, `execvp`

表头文件 `#include <unistd.h>`

定义函数 `int execl(const char * path,const char * arg,...);`

函数说明 `execl()`用来执行参数`path`字符串所代表的文件路径，接下来的参数代表执行该文件时传递过去的`argv(0)`、`argv[1]`.....，最后一个参数必须用空指针(`NULL`)作结束。

返回值 如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于`errno`中。

```
范例 #include <unistd.h>
      main()
      {
        execl("/bin/l", "l", "-al", "/etc/passwd", (char *)0);
      }
```

执行 /*执行/bin/l -al /etc/passwd */
-rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd

execlp（从`PATH` 环境变量中查找文件并执行）

相关函数 `fork`, `execl`, `execle`, `execv`, `execve`, `execvp`

表头文件 `#include <unistd.h>`

定义函数 `int execlp(const char * file,const char * arg,.....);`

函数说明 `execlp()`会从`PATH` 环境变量所指的目录中查找符合参数`file`的文件名，找到后便执行该文件，然后将第二个以后的参数当做该文件的`argv[0]`、`argv[1]`.....，最后一个参数必须用空指针(`NULL`)作结束。

返回值 如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于errno 中。

错误代码 参考execve()。

```
范例 /* 执行ls -al /etc/passwd execlp()会依PATH 变量中的/bin找到/bin/ls */
#include<unistd.h>
main()
{
    execlp("ls","ls","-al","/etc/passwd",(char *)0);
}
```

执行 -rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd

execv（执行文件）

相关函数 fork, execl, execl, execlp, execve, execvp

表头文件 #include<unistd.h>

定义函数 int execv (const char * path, char * const argv[]);

函数说明 execv()用来执行参数path字符串所代表的文件路径，与execl()不同的地方在于execve()只需两个参数，第二个参数利用数组指针来传递给执行文件。

返回值 如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于errno 中。

错误代码 请参考execve（）。

```
范例 /* 执行/bin/ls -al /etc/passwd */
#include<unistd.h>
main()
{
    char * argv[]={"ls","-al","/etc/passwd",(char*)0};
    execv("/bin/ls",argv);
}
```

执行 -rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd

execve（执行文件）

相关函数 fork, execl, execl, execlp, execv, execvp

表头文件 #include<unistd.h>

定义函数 int execve(const char * filename,char * const argv[],char * const envp[]);

函数说明 execve()用来执行参数filename字符串所代表的文件路径，第二个参数系利用数组指针来传递给执行文件，最后一个参数则为传递给执行文件的新环境变量数组。

返回值 如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于errno 中。

错误代码 EACCES

1. 欲执行的文件不具有用户可执行的权限。
2. 欲执行的文件所属的文件系统是以noexec 方式挂上。
3. 欲执行的文件或script翻译器非一般文件。

EPERM

1. 进程处于被追踪模式，执行者并不具有root权限，欲执行的文件具有SUID 或SGID 位。
2. 欲执行的文件所属的文件系统是以nosuid方式挂上，欲执行的文件具有SUID 或SGID 位元，但执行者并不具有root权限。

E2BIG 参数数组过大

ENOEXEC 无法判断欲执行文件的执行文件格式，有可能是格式错误或无法在此平台执行。

EFAULT 参数filename所指的字符串地址超出可存取空间范围。

ENAMETOOLONG 参数filename所指的字符串太长。

ENOENT 参数filename字符串所指定的文件不存在。

ENOMEM 核心内存不足

ENOTDIR 参数filename字符串所包含的目录路径并非有效目录
EACCES 参数filename字符串所包含的目录路径无法存取，权限不足
ELOOP 过多的符号连接
ETXTBUSY 欲执行的文件已被其他进程打开而且正把数据写入该文件中
EIO I/O 存取错误
ENFILE 已达到系统所允许的打开文件总数。
EMFILE 已达到系统所允许单一进程所能打开的文件总数。
EINVAL 欲执行文件的ELF执行格式不只一个PT_INTERP节区
EISDIR ELF翻译器为一目录
ELIBBAD ELF翻译器有问题。

范例 #include<unistd.h>

```
main()
{
    char * argv[]={"ls","-al","/etc/passwd",(char *)0};
    char * envp[]={"PATH=/bin",0}
    execve("/bin/ls",argv,envp);
}
```

执行 -rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd

execvp（执行文件）

相关函数 fork, execl, execl, execlp, execv, execve

表头文件 #include<unistd.h>

定义函数 int execvp(const char *file, char * const argv []);

函数说明 execvp()会从PATH 环境变量所指的目录中查找符合参数file 的文件名，找到后便执行该文件，然后将第二个参数argv传给该欲执行的文件。

返回值 如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于errno中。

错误代码 请参考execve（）。

范例 /*请与execlp（）范例对照*/

```
#include<unistd.h>
main()
{
    char * argv[] ={"ls","-al","/etc/passwd",0};
    execvp("ls",argv);
}
```

执行 -rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd

exit（正常结束进程）

相关函数 _exit, atexit, on_exit

表头文件 #include<stdlib.h>

定义函数 void exit(int status);

函数说明 exit()用来正常终结目前进程的执行，并把参数status返回给父进程，而进程所有的缓冲区数据会自动写回并关闭未关闭的文件。

返回值

范例 参考wait（）

`_exit`（结束进程执行）

相关函数 `exit`, `wait`, `abort`

表头文件 `#include <unistd.h>`

定义函数 `void _exit(int status);`

函数说明 `_exit()`用来立刻结束目前进程的执行，并把参数`status`返回给父进程，并关闭未关闭的文件。此函数调用后不会返回，并且会传递`SIGCHLD`信号给父进程，父进程可以由`wait`函数取得子进程结束状态。

返回值

附加说明 `_exit()` 不会处理标准I/O 缓冲区，如要更新缓冲区请使用`exit()`。

`vfork`（建立一个新的进程）

相关函数 `wait`, `execve`

表头文件 `#include <unistd.h>`

定义函数 `pid_t vfork(void);`

函数说明 `vfork()`会产生一个新的子进程，其子进程会复制父进程的数据与堆栈空间，并继承父进程的用户代码，组代码，环境变量、已打开的文件代码、工作目录和资源限制等。Linux 使用`copy-on-write(COW)`技术，只有当其中一进程试图修改欲复制的空间时才会做真正的复制动作，由于这些继承的信息是复制而来，并非指相同的内存空间，因此子进程对这些变量的修改和父进程并不会同步。此外，子进程不会继承父进程的文件锁定和未处理的信号。注意，Linux不保证子进程会比父进程先执行或晚执行，因此编写程序时要注意

死锁或竞争条件的发生。

返回值 如果`vfork()`成功则在父进程会返回新建的子进程代码(PID)，而在新建的子进程中则返回0。如果`vfork` 失败则直接返回-1，失败原因存于`errno`中。

错误代码 `EAGAIN` 内存不足。`ENOMEM` 内存不足，无法配置核心所需的数据结构空间。

```
范例 #include <unistd.h>
      main()
      {
        if(vfork() == 0)
        {
          printf("This is the child process\n");
        }else{
          printf("This is the parent process\n");
        }
      }
```

执行 this is the parent process
this is the child process

`getpgid`（取得进程组识别码）

相关函数 `setpgid`, `setpgrp`, `getpgrp`

表头文件 `#include <unistd.h>`

定义函数 `pid_t getpgid(pid_t pid);`

函数说明 `getpgid()`用来取得参数`pid` 指定进程所属的组识别码。如果参数`pid`为0，则会取得目前进程的组识别码。

返回值 执行成功则返回组识别码，如果有错误则返回-1，错误原因存于`errno`中。

错误代码 `ESRCH` 找不到符合参数`pid` 指定的进程。

范例 /*取得init 进程（pid=1）的组识别码*/

```
#include<unistd.h>
main()
{
    printf("init gid = %d\n",getpgid(1));
}
```

执行 init gid = 0

getpgrp（取得进程组识别码）

相关函数 setpgid, getpgid, getpgrp

表头文件 #include<unistd.h>

定义函数 pid_t getpgrp(void);

函数说明 getpgrp()用来取得目前进程所属的组织识别码。此函数相当于调用getpgid(0);

返回值 返回目前进程所属的组织识别码。

```
范例 #include<unistd.h>
main()
{
    printf("my gid =%d\n",getpgrp());
}
```

执行 my gid =29546

getpid（取得进程识别码）

相关函数 fork, kill, getpid

表头文件 #include<unistd.h>

定义函数 pid_t getpid(void);

函数说明 getpid（）用来取得目前进程的进程识别码，许多程序利用取到的此值来建立临时文件，以避免临时文件相同带来的问题。

返回值 目前进程的进程识别码

```
范例 #include<unistd.h>
main()
{
    printf("pid=%d\n",getpid());
}
```

执行 pid=1494 /*每次执行结果都不一定相同*/

getppid（取得父进程的进程识别码）

相关函数 fork, kill, getpid

表头文件 #include<unistd.h>

定义函数 pid_t getppid(void);

函数说明 getppid()用来取得目前进程的父进程识别码。

返回值 目前进程的父进程识别码。

```
范例 #include<unistd.h>
main()
{
    printf("My parent 'pid =%d\n",getppid());
}
```

```
}
```

执行 My parent pid =463

getpriority（取得程序进程执行优先权）

相关函数 setpriority, nice

表头文件 #include<sys/time.h>

#include<sys/resource.h>

定义函数 int getpriority(int which,int who);

函数说明 getpriority()可用来取得进程、进程组和用户的进程执行优先权。

参数 which有三种数值，参数who 则依which值有不同定义

which who 代表的意义

PRIO_PROCESS who 为进程识别码

PRIO_PGRP who 为进程的组识别码

PRIO_USER who 为用户识别码

此函数返回的数值介于-20 至20之间，代表进程执行优先权，数值越低代表有较高的优先次序，执行会较频繁。

返回值 返回进程执行优先权，如有错误发生返回值则为-1 且错误原因存于errno。

附加说明 由于返回值有可能是-1，因此要同时检查errno是否存有错误原因。最好在调用次函数前先清除errno变量。

错误代码 ESRCH 参数which或who 可能有错，而找不到符合的进程。EINVAL 参数which 值错误。

nice（改变进程优先顺序）

相关函数 setpriority, getpriority

表头文件 #include<unistd.h>

定义函数 int nice(int inc);

函数说明 nice()用来改变进程的进程执行优先顺序。参数inc数值越大则优先顺序排在越后面，即表示进程执行会越慢。只有超级用户才能使用负的inc 值，代表优先顺序排在前面，进程执行会较快。

返回值 如果执行成功则返回0，否则返回-1，失败原因存于errno中。

错误代码 EPERM 一般用户企图转用负的参数inc值改变进程优先顺序。

on_exit（设置程序正常结束前调用的函数）

相关函数 _exit, atexit, exit

表头文件 #include<stdlib.h>

定义函数 int on_exit(void (* function)(int, void*),void *arg);

函数说明 on_exit()用来设置一个程序正常结束前调用的函数。当程序通过调用exit()或从main中返回时，参数function所指定的函数会先被调用，然后才真正由exit()结束程序。参数arg指针会传给参数function 函数，详细情况请见范例。

返回值 如果执行成功则返回0，否则返回-1，失败原因存于errno中。

附加说明

范例 #include<stdlib.h>

void my_exit(int status,void *arg)

{

printf("before exit()!\n");

printf("exit (%d)\n",status);

```

printf("arg = %s\n", (char*)arg);
}
main()
{
char * str="test";
on_exit(my_exit, (void *)str);
exit(1234);
}

```

执行 before exit()!
 exit (1234)
 arg = test

setpgid（设置进程组识别码）

相关函数 getpgid, setpgrp, getpgrp

表头文件 #include <unistd.h>

定义函数 int setpgid(pid_t pid, pid_t pgid);

函数说明 setpgid()将参数pid 指定进程所属的组织识别码设为参数pgid 指定的组织识别码。如果参数pid 为0，则会用来设置目前进程的组识别码，如果参数pgid为0，则会以目前进程的进程识别码来取代。

返回值 执行成功则返回组识别码，如果有错误则返回-1，错误原因存于errno中。

错误代码 EINVAL 参数pgid小于0。

EPERM 进程权限不足，无法完成调用。

ESRCH 找不到符合参数pid指定的进程。

setpgrp（设置进程组识别码）

相关函数 getpgid, setpgid, getpgrp

表头文件 #include <unistd.h>

定义函数 int setpgrp(void);

函数说明 setpgrp()将目前进程所属的组织识别码设为目前进程的进程识别码。此函数相当于调用setpgid(0,0)。

返回值 执行成功则返回组识别码，如果有错误则返回-1，错误原因存于errno中。

setpriority（设置程序进程执行优先权）

相关函数 getpriority, nice

表头文件 #include <sys/time.h>

#include <sys/resource.h>

定义函数 int setpriority(int which, int who, int prio);

函数说明 setpriority()可用来设置进程、进程组和用户的进程执行优先权。参数which有三种数值，参数who则依which值有不同定义

which who 代表的意义

PRIO_PROCESS who为进程识别码

PRIO_PGRP who 为进程的组识别码

PRIO_USER who为用户识别码

参数prio介于-20 至20 之间。代表进程执行优先权，数值越低代表有较高的优先次序，执行会较频繁。此优先权默认是0，而只有超级用户（root）允许降低此值。

返回值 执行成功则返回0，如果有错误发生返回值则为-1，错误原因存于errno。

ESRCH 参数which或who 可能有错，而找不到符合的进程

EINVAL 参数which值错误。
EPERM 权限不够，无法完成设置
EACCES 一般用户无法降低优先权

system（执行shell 命令）

相关函数 fork, execve, waitpid, popen

表头文件 #include<stdlib.h>

定义函数 int system(const char * string);

函数说明 system()会调用fork()产生子进程，由子进程来调用/bin/sh-c string来执行参数string字符串所代表的命令，此命令执行完后随即返回原调用的进程。在调用system()期间SIGCHLD 信号会被暂时搁置，SIGINT和SIGQUIT 信号则会被忽略。

返回值 如果system()在调用/bin/sh时失败则返回127，其他失败原因返回-1。若参数string为空指针(NULL)，则返回非零值。如果system()调用成功则最后会返回执行shell命令后的返回值，但是此返回值也有可能为system()调用/bin/sh失败所返回的127，因此最好能再检查errno 来确认执行成功。

附加说明 在编写具有SUID/SGID权限的程序时请勿使用system()，system()会继承环境变量，通过环境变量可能会造成系统安全的问题。

范例 #include<stdlib.h>
main()
{
system("ls -al /etc/passwd /etc/shadow");
}

执行 -rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd
-r----- 1 root root 572 Sep 2 15 :34 /etc/shadow

wait（等待子进程中断或结束）

相关函数 waitpid, fork

表头文件 #include<sys/types.h>

#include<sys/wait.h>

定义函数 pid_t wait (int * status);

函数说明 wait()会暂时停止目前进程的执行，直到有信号来到或子进程结束。如果在调用wait()时子进程已经结束，则wait()会立即返回子进程结束状态值。子进程的结束状态值会由参数status 返回，而子进程的进程识别码也会一并返回。如果不在意结束状态值，则

参数 status可以设成NULL。子进程的结束状态值请参考waitpid()。

返回值 如果执行成功则返回子进程识别码(PID)，如果有错误发生则返回-1。失败原因存于errno中。

附加说明

范例 #include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>
main()
{
pid_t pid;
int status,i;
if(fork() == 0){
printf("This is the child process .pid =%d\n",getpid());
exit(5);
}else{
sleep(1);

```
printf("This is the parent process ,wait for child...\n");
pid=wait(&status);
i=WEXITSTATUS(status);
printf("child's pid =%d .exit status= ^d\n",pid,i);
}
}
```

执行 This is the child process.pid=1501
This is the parent process .wait for child...
child's pid =1501,exit status =5

waitpid（等待子进程中断或结束）

相关函数 **wait**, **fork**

表头文件 **#include<sys/types.h>**
#include<sys/wait.h>

定义函数 **pid_t waitpid(pid_t pid,int * status,int options);**

函数说明 **waitpid()**会暂时停止目前进程的执行，直到有信号来到或子进程结束。如果在调用**wait()**时子进程已经结束，则**wait()**会立即返回子进程结束状态值。子进程的结束状态值会由参数**status**返回，而子进程的进程识别码也会一快返回。如果不在意结束状态值，则参数**status**可以设成**NULL**。参数**pid**为欲等待的子进程识别码，其他数值意义如下：

pid<-1 等待进程组识别码为**pid**绝对值的任何子进程。

pid=-1 等待任何子进程，相当于**wait()**。

pid=0 等待进程组识别码与目前进程相同的任何子进程。

pid>0 等待任何子进程识别码为**pid**的子进程。

参数**option**可以为0 或下面的OR 组合

WNOHANG 如果没有任何已经结束的子进程则马上返回，不予以等待。

WUNTRACED 如果子进程进入暂停执行情况则马上返回，但结束状态不予以理会。

子进程的结束状态返回后存于**status**，底下有几个宏可判别结束情况

WIFEXITED(status)如果子进程正常结束则为非0值。

WEXITSTATUS(status)取得子进程**exit()**返回的结束代码，一般会先用**WIFEXITED** 来判断是否正常结束才能使用此宏。

WIFSIGNALED(status)如果子进程是因为信号而结束则此宏值为真

WTERMSIG(status)取得子进程因信号而中止的信号代码，一般会先用**WIFSIGNALED** 来判断后才使用此宏。

WIFSTOPPED(status)如果子进程处于暂停执行情况则此宏值为真。一般只有使用**WUNTRACED** 时才会有此情况。

WSTOPSIG(status)取得引发子进程暂停的信号代码，一般会先用**WIFSTOPPED** 来判断后才使用此宏。

返回值 如果执行成功则返回子进程识别码(PID)，如果有错误发生则返回-1。失败原因存于**errno**中。

范例 参考**wait()**。

fprintf（格式化输出数据至文件）

相关函数 **printf**, **fscanf**, **vfprintf**

表头文件 **#include<stdio.h>**

定义函数 **int fprintf(FILE * stream, const char * format,.....);**

函数说明 **fprintf()**会根据参数**format**字符串来转换并格式化数据，然后将结果输出到参数**stream**指定的文件中，直到出现字符串结束('\0')为止。

返回值 关于参数**format**字符串的格式请参考**printf()**。成功则返回实际输出的字符数，失败则返回-1，错误原因存于**errno**中。

```

范例 #include<stdio.h>
main()
{
int i = 150;
int j = -100;
double k = 3.14159;
fprintf(stdout,"%d %f %x \n",j,k,i);
fprintf(stdout,"%2d %*d\n",i,2,i);
}

```

执行 -100 3.141590 96
150 150

fscanf（格式化字符串输入）

相关函数 scanf， sscanf

表头文件 #include<stdio.h>

定义函数 int fscanf(FILE * stream ,const char *format,...);

函数说明 fscanf()会自参数stream的文件流中读取字符串，再根据参数format字符串来转换并格式化数据。格式转换形式请参考scanf()。转换后的结构存于对应的参数内。

返回值 成功则返回参数数目，失败则返回-1，错误原因存于errno中。

附加说明

```

范例 #include<stdio.h>
main()
{
int i;
unsigned int j;
char s[5];
fscanf(stdin,"%d %x %5[a-z] %*s %f",&i,&j,s,s);
printf("%d %d %s \n",i,j,s);
}

```

执行 10 0x1b aaaaaaaaaa bbbbbbbbbbb /*从键盘输入*/
10 27 aaaaa

printf（格式化输出数据）

相关函数 scanf， snprintf

表头文件 #include<stdio.h>

定义函数 int printf(const char * format,.....);

函数说明 printf()会根据参数format字符串来转换并格式化数据，然后将结果写出到标准输出设备，直到出现字符串结束('\0')为止。参数format字符串可包含下列三种字符类型

- 1.一般文本，伴随直接输出。
- 2.ASCII控制字符，如\t、\n等。
- 3.格式转换字符。

格式转换为一个百分比符号(%)及其后的格式字符所组成。一般而言，每个%符号在其后都必需有一printf()的参数与之相呼应（只有当%%转换字符出现时会直接输出%字符），而欲输出的数据类型必须与其相对应的转换字符类型相同。

Printf()格式转换的一般形式如下

%(flags)(width)(.prec)type

以中括号括起来的参数为选择性参数，而%与type则是必要的。底下先介绍type的几种形式

整数

%d 整数的参数会被转成一有符号的十进制数字

%u 整数的参数会被转成一无符号的十进制数字

%o 整数的参数会被转成一无符号的八进制数字

%x 整数的参数会被转成一无符号的十六进制数字，并以小写abcdef表示

%X 整数的参数会被转成一无符号的十六进制数字，并以大写ABCDEF表示浮点型数

%f double 型的参数会被转成十进制数字，并取到小数点以下六位，四舍五入。

%e double型的参数以指数形式打印，有一个数字会在小数点前，六位数字在小数点后，而在指数部分会以小写的e来表示。

%E 与 %e 作用相同，唯一区别是指数部分将以大写的E 来表示。

%g double 型的参数会自动选择以 %f 或 %e 的格式来打印，其标准是根据欲打印的数值及所设置的有效位数来决定。

%G 与 %g 作用相同，唯一区别在以指数形态打印时会选择 %E 格式。

字符及字符串

%c 整型数的参数会被转成 unsigned char 型打印出。

%s 指向字符串的参数会被逐字输出，直到出现 NULL 字符为止

%p 如果是参数是 "void *" 型指针则使用十六进制格式显示。

prec 有几种情况

1. 正整数的最小位数。

2. 在浮点型数中代表小数位数

3. 在 %g 格式代表有效位数的最大值。

4. 在 %s 格式代表字符串的最大长度。

5. 若为 x 符号则代表下个参数值为最大长度。

width 为参数的最小长度，若此栏并非数值，而是 * 符号，则表示以下一个参数当做参数长度。

flags 有下列几种情况

#NAME?

+ 一般在打印负数时，printf () 会加印一个负号，整数则不加任何负号。此旗标会使得在打印正数前多一个正号 (+) 。

此旗标会根据其后转换字符的不同而有不同含义。当在类型为 o 之前 (如 %#o) ，则会在打印八进制数值前多印一个 o 。

而在类型为 x 之前 (%#x) 则会在打印十六进制数前多印 '0x' ，在型态为 e、E、f、g 或 G 之前则会强迫数值打印小数点。在类型为 g 或 G 之前时则同时保留小数点及小数位数末尾的零。

0 当有指定参数时，无数值的参数将补上 0。默认是关闭此旗标，所以一般会打印出空白字符。

返回值 成功则返回实际输出的字符数，失败则返回 -1，错误原因存于 errno 中。

范例 #include <stdio.h>

main()

{

int i = 150;

int j = -100;

double k = 3.14159;

printf ("%d %f %x\n", j, k, i);

printf ("%2d %*d\n", i, 2, i); /* 参数 2 会代入格式 * 中，而与 %2d 同意义 */

}

执行 -100 3.14159 96

150 150

sacnf (格式化字符串输入)

相关函数 fscanf, snprintf

表头文件 #include <stdio.h>

定义函数 int scanf(const char * format,.....);

函数说明 scanf() 会将输入的数据根据参数 format 字符串来转换并格式化数据。Scanf() 格式转换的一般形式如下

[%*][size][l][h]type

以中括号括起来的参数为选择性参数，而 % 与 type 则是必要的。

* 代表该对应的参数数据忽略不保存。

size 为允许参数输入的数据长度。

l 输入的数据数值以long int 或double型保存。

h 输入的数据数值以short int 型保存。

底下介绍type的几种形式

%d 输入的数据会被转成一有符号的十进制数字 (int)。

%i 输入的数据会被转成一有符号的十进制数字，若输入数据以“0x”或“0X”开头代表转换十六进制数字，若以“0”开头则转换八进制数字，其他情况代表十进制。

%O 输入的数据会被转换成一无符号的八进制数字。

%u 输入的数据会被转换成一无符号的正整数。

%x 输入的数据为无符号的十六进制数字，转换后存于unsigned int型变量。

%X 同%x

%f 输入的数据为有符号的浮点型数，转换后存于float型变量。

%e 同%f

%E 同%f

%g 同%f

%s 输入数据为以空格字符为终止的字符串。

%c 输入数据为单一字符。

[] 读取数据但只允许括号内的字符。如[a-z]。

[^] 读取数据但不允许中括号的^符号后的字符出现，如[^0-9]。

返回值 成功则返回参数数目，失败则返回-1，错误原因存于errno中。

范例 #include <stdio.h>

```
main()
{
    int i;
    unsigned int j;
    char s[5];
    scanf("%d %x %5[a-z] %s %f",&i,&j,s,s);
    printf("%d %d %s\n",i,j,s);
}
```

执行 10 0x1b aaaaaaaaaa bbbbbbbbbbb
10 27 aaaaa

sprintf (格式化字符串复制)

相关函数 printf, sprintf

表头文件 #include<stdio.h>

定义函数 int sprintf(char *str,const char * format,.....);

函数说明 sprintf()会根据参数format字符串来转换并格式化数据，然后将结果复制到参数str所指的字符串数组，直到出现字符串结束('\0')为止。关于参数format字符串的格式请参考printf()。

返回值 成功则返回参数str字符串长度，失败则返回-1，错误原因存于errno中。

附加说明 使用此函数得留意堆栈溢出，或改用snprintf ()。

范例 #include<stdio.h>

```
main()
{
    char * a="This is string A!";
    char buf[80];
    sprintf(buf,">>> %s<<<\n",a);
    printf("%s".buf);
}
```

执行 >>>This is string A!<<<

sscanf（格式化字符串输入）

相关函数 scanf, fscanf

表头文件 #include<stdio.h>

定义函数 int sscanf (const char *str,const char * format,.....);

函数说明 sscanf()会将参数str的字符串根据参数format字符串来转换并格式化数据。格式转换形式请参考scanf()。转换后的结果存于对应的参数内。

返回值 成功则返回参数数目，失败则返回-1，错误原因存于errno中。

```
范例 #include<stdio.h>
main()
{
    int i;
    unsigned int j;
    char input[ ]="10 0x1b aaaaaaaa bbbbbbbb";
    char s[5];
    sscanf(input,"%d %x %5[a-z] %*s %f",&i,&j,s,s);
    printf("%d %d %s\n",i,j,s);
}
```

执行 10 27 aaaaa

vfprintf（格式化输出数据至文件）

相关函数 printf, fscanf, fprintf

表头文件 #include<stdio.h>

#include<stdarg.h>

定义函数 int vfprintf(FILE *stream,const char * format,va_list ap);

函数说明 vfprintf()会根据参数format字符串来转换并格式化数据，然后将结果输出到参数stream指定的文件中，直到出现字符串结束('\0')为止。关于参数format字符串的格式请参考printf()。va_list用法请参考附录C或vprintf()范例。

返回值 成功则返回实际输出的字符数，失败则返回-1，错误原因存于errno中。

范例 参考fprintf()及vprintf()。

vfscanf（格式化字符串输入）

相关函数 scanf, sscanf, fscanf

表头文件 #include<stdio.h>

定义函数 int vfscanf(FILE * stream,const char * format ,va_list ap);

函数说明 vfscanf()会自参数stream 的文件流中读取字符串，再根据参数format字符串来转换并格式化数据。格式转换形式请参考scanf()。转换后的结果存于对应的参数内。va_list用法请参考附录C 或 vprintf()。

返回值 成功则返回参数数目，失败则返回-1，错误原因存于errno中。

范例 参考fscanf()及vprintf()。

vprintf（格式化输出数据）

相关函数 printf, fprintf, vsprintf

表头文件 #include<stdio.h>

```
#include<stdarg.h>
```

定义函数 `int vprintf(const char * format,va_list ap);`

函数说明 `vprintf()`作用和`printf()`相同，参数`format`格式也相同。`va_list`为不定个数的参数列，用法及范例请参考附录C。

返回值 成功则返回实际输出的字符数，失败则返回-1，错误原因存于`errno`中。

```
范例 #include<stdio.h>
#include<stdarg.h>
int my_printf( const char *format,.....)
{
    va_list ap;
    int retval;
    va_start(ap,format);
    printf("my_printf( ):");
    retval = vprintf(format,ap);
    va_end(ap);
    return retval;
}
main()
{
    int i = 150,j = -100;
    double k = 3.14159;
    my_printf("%d %f %x\n",j,k,i);
    my_printf("%2d %*d\n",i,2,i);
}
```

执行 `my_printf()` : -100 3.14159 96

`my_printf()` : 150 150

`vscanf`（格式化字符串输入）

相关函数 `vsscanf`，`vfscanf`

表头文件 `#include<stdio.h>`

```
#include<stdarg.h>
```

定义函数 `int vscanf(const char * format,va_list ap);`

函数说明 `vscanf()`会将输入的数据根据参数`format`字符串来转换并格式化数据。格式转换形式请参考`scanf()`。转换后的结果存于对应的参数内。`va_list`用法请参考附录C或`vprintf()`范例。

返回值 成功则返回参数数目，失败则返回-1，错误原因存于`errno`中。

范例 请参考`scanf()`及`vprintf()`。

`vsprintf`（格式化字符串复制）

相关函数 `vnsprintf`，`vprintf`，`snprintf`

表头文件 `#include<stdio.h>`

定义函数 `int vsprintf(char * str,const char * format,va_list ap);`

函数说明 `vsprintf()`会根据参数`format`字符串来转换并格式化数据，然后将结果复制到参数`str`所指的字符串数组，直到出现字符串结束('\0')为止。关于参数`format`字符串的格式请参考`printf()`。`va_list`用法请参考附录C或`vprintf()`范例。

返回值 成功则返回参数`str`字符串长度，失败则返回-1，错误原因存于`errno`中。

范例 请参考`vprintf()`及`vsprintf()`。

vsscanf（格式化字符串输入）

相关函数 vscanf, vfscanf

表头文件 #include<stdio.h>

定义函数 int vsscanf(const char * str,const char * format,va_list ap);

函数说明 vsscanf()会将参数str的字符串根据参数format字符串来转换并格式化数据。格式转换形式请参考附录C 或vprintf()范例。

返回值 成功则返回参数数目，失败则返回-1，错误原因存于errno中。

范例 请参考sscanf()及vprintf()。

access（判断是否具有存取文件的权限）

相关函数 stat, open, chmod, chown, setuid, setgid

表头文件 #include<unistd.h>

定义函数 int access(const char * pathname,int mode);

函数说明 access()会检查是否可以读/写某一已存在的文件。参数mode有几种情况组合，R_OK，W_OK，X_OK 和F_OK。R_OK，W_OK与X_OK用来检查文件是否具有读取、写入和执行的权限。F_OK则是用来判断该文件是否存在。由于access()只作权限的核查，并不理会文件形态或文件内容，因此，如果一目录表示为“可写入”，表示可以在该目录中建立新文件等操作，而非意味此目录可以被当做文件处理。例如，你会发现DOS的文件都具有“可执行”权限，但用execve()执行时则会失败。

返回值 若所有欲查核的权限都通过了检查则返回0值，表示成功，只要有一权限被禁止则返回-1。

错误代码 EACCESS 参数pathname 所指定的文件不符合所要求测试的权限。

EROFS 欲测试写入权限的文件存在于只读文件系统内。

EFAULT 参数pathname指针超出可存取内存空间。

EINVAL 参数mode 不正确。

ENAMETOOLONG 参数pathname太长。

ENOTDIR 参数pathname为一目录。

ENOMEM 核心内存不足

ELOOP 参数pathname有过多符号连接问题。

EIO I/O 存取错误。

附加说明 使用access()作用户认证方面的判断要特别小心，例如在access()后再做open()的空文件可能会造成系统安全上的问题。

```
范例 /* 判断是否允许读取/etc/passwd */
#include<unistd.h>
int main()
{
    if (access("/etc/passwd",R_OK) == 0)
        printf("/etc/passwd can be read\n");
}
```

执行 /etc/passwd can be read

alphasort（依字母顺序排序目录结构）

相关函数 scandir, qsort

表头文件 #include<dirent.h>

定义函数 int alphasort(const struct dirent **a,const struct dirent **b);

函数说明 alphasort()为scandir()最后调用qsort()函数时传给qsort()作为判断的函数，详细说明请参考scandir()及qsort()。

返回值 参考qsort()。

```
范例 /* 读取/目录下所有的目录结构，并依字母顺序排列*/
main()
{
    struct dirent **namelist;
    int i,total;
    total = scandir("/",&namelist,0,alphasort);
    if(total < 0)
        perror("scandir");
}
```

```

else{
for(i=0;i<total;i++)
printf("%s\n",namelist[i]->d_name);
printf("total = %d\n",total);
}
}

```

执行 ..

```

.gnome
.gnome_private
ErrorLog
Weblog
bin
boot
dev
dosc
dosd
etc
home
lib
lost+found
misc
mnt
opt
proc
root
sbin
tmp
usr
var
total = 24

```

chdir（改变当前的工作（目录））

相关函数 `getcwd`，`chroot`

表头文件 `#include <unistd.h>`

定义函数 `int chdir(const char * path);`

函数说明 `chdir()`用来将当前的工作目录改变成以参数`path`所指的目录。

返回值 执行成功则返回0，失败返回-1，`errno`为错误代码。

范例 `#include <unistd.h>`

```

main()
{
chdir("/tmp");
printf("current working directory: %s\n",getcwd(NULL,NULL));
}

```

执行 current working directory :/tmp

chmod（改变文件的权限）

相关函数 `fchmod`，`stat`，`open`，`chown`

表头文件 `#include <sys/types.h>`

`#include <sys/stat.h>`

定义函数 `int chmod(const char * path,mode_t mode);`

函数说明 `chmod()`会依参数`mode` 权限来更改参数`path` 指定文件的权限。

参数 `mode` 有下列数种组合

`S_ISUID 04000` 文件的（set user-id on execution）位

`S_ISGID 02000` 文件的（set group-id on execution）位

`S_ISVTX 01000` 文件的sticky位

`S_IRUSR (S_IREAD) 00400` 文件所有者具可读取权限

`S_IWUSR (S_IWRITE) 00200` 文件所有者具可写入权限

`S_IXUSR (S_IEXEC) 00100` 文件所有者具可执行权限

`S_IRGRP 00040` 用户组具可读取权限

`S_IWGRP 00020` 用户组具可写入权限

`S_IXGRP 00010` 用户组具可执行权限

`S_IROTH 00004` 其他用户具可读取权限

`S_IWOTH 00002` 其他用户具可写入权限

`S_IXOTH 00001` 其他用户具可执行权限

只有该文件的所有者或有效用户识别码为0，才可以修改该文件权限。基于系统安全，如果欲将数据写入一执行文件，而该执行文件具有`S_ISUID` 或`S_ISGID` 权限，则这两个位会被清除。如果一目录具有`S_ISUID` 位权限，表示在此目录下只有该文件的所有者或root可以删除该文件。

返回值 权限改变成功返回0，失败返回-1，错误原因存于`errno`。

错误代码 `EPERM` 进程的有效用户识别码与欲修改权限的文件拥有者不同，而且也不具root权限。

`EACCESS` 参数`path`所指定的文件无法存取。

`EROFS` 欲写入权限的文件存在于只读文件系统内。

`EFAULT` 参数`path`指针超出可存取内存空间。

`EINVAL` 参数`mode`不正确

`ENAMETOOLONG` 参数`path`太长

`ENOENT` 指定的文件不存在

`ENOTDIR` 参数`path`路径并非一目录

`ENOMEM` 核心内存不足

`ELOOP` 参数`path`有过多符号连接问题。

`EIO` I/O 存取错误

范例 `/* 将/etc/passwd 文件权限设成S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH */`

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
main()
```

```
{
```

```
  chmod("/etc/passwd",S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
```

```
}
```

`chown`（改变文件的所有者）

相关函数 `fchown`，`lchown`，`chmod`

表头文件 `#include <sys/types.h>`

```
#include <unistd.h>
```

定义函数 `int chown(const char * path, uid_t owner,gid_t group);`

函数说明 `chown()`会将参数`path`指定文件的所有者变更为参数`owner`代表的用户，而将该文件的组变更为参数`group`组。如果参数`owner`或`group`为-1，对应的所有者或组不会有所改变。root与文件所有者皆可改变文件组，但所有者必须是参数`group`组的成员。当root用`chown()`改变文件所有者或组时，该文件若具有`S_ISUID`或`S_ISGID`权限，则会清除此权限位，此外如果具有`S_ISGID`权限但不具`S_IXGRP`位，则该文件会被强制锁定，文件模式会保留。

返回值 成功则返回0，失败返回-1，错误原因存于`errno`。

错误代码 参考`chmod`（）。

范例 `/* 将/etc/passwd 的所有者和组都设为root */`

```
#include <sys/types.h>
```

```
#include<unistd.h>
main()
{
chown("/etc/passwd",0,0);
}
```

chroot（改变根目录）

相关函数 chdir

表头文件 #include<unistd.h>

定义函数 int chroot(const char * path);

函数说明 chroot()用来改变根目录为参数path 所指定的目录。只有超级用户才允许改变根目录，子进程将继承新的根目录。

返回值 调用成功则返回0，失败则返-1，错误代码存于errno。

错误代码 EPERM 权限不足，无法改变根目录。

EFAULT 参数path指针超出可存取内存空间。

ENAMETOOLONG 参数path太长。

ENOTDIR 路径中的目录存在但却非真正的目录。

EACCESS 存取目录时被拒绝

ENOMEM 核心内存不足。

ELOOP 参数path有过多符号连接问题。

EIO I/O 存取错误。

范例 /* 将根目录改为/tmp,并将工作目录切换至/tmp */

```
#include<unistd.h>
main()
{
chroot("/tmp");
chdir("/");
}
```

closedir（关闭目录）

相关函数 opendir

表头文件 #include<sys/types.h>

#include<dirent.h>

定义函数 int closedir(DIR *dir);

函数说明 closedir()关闭参数dir所指的目录流。

返回值 关闭成功则返回0，失败返回-1，错误原因存于errno 中。

错误代码 EBADF 参数dir为无效的目录流

范例 参考readir()。

fchdir（改变当前的工作目录）

相关函数 getcwd, chroot

表头文件 #include<unistd.h>

定义函数 int fchdir(int fd);

函数说明 fchdir()用来将当前的工作目录改变成以参数fd 所指的描述词。

返回值 执行成功则返回0，失败返回-1，errno为错误代码。

附加说明

```
范例 #include<sys/types.h>
      #include<sys/stat.h>
      #include<fcntl.h>
      #include<unistd.h>
      main()
      {
      int fd;
      fd = open("/tmp",O_RDONLY);
      fchdir(fd);
      printf("current working directory : %s \n",getcwd(NULL,NULL));
      close(fd);
      }
```

执行 current working directory : /tmp

fchmod（改变文件的权限）

相关函数 chmod，stat，open，chown

表头文件 #include<sys/types.h>
#include<sys/stat.h>

定义函数 int fchmod(int fildes,mode_t mode);

函数说明 fchmod()会依参数mode权限来更改参数fildes所指文件的权限。参数fildes为已打开文件的文件描述词。参数mode请参考chmod（）。

返回值 权限改变成功则返回0，失败返回-1，错误原因存于errno。

错误原因 EBADF 参数fildes为无效的文件描述词。

EPERM 进程的有效用户识别码与欲修改权限的文件所有者不同，而且也不具root权限。

EROFS 欲写入权限的文件存在于只读文件系统内。

EIO I/O 存取错误。

```
范例 #include<sys/stat.h>
      #include<fcntl.h>
      main()
      {
      int fd;
      fd = open ("/etc/passwd",O_RDONLY);
      fchmod(fd,S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
      close(fd);
      }
```

fchown（改变文件的所有者）

相关函数 chown，lchown，chmod

表头文件 #include<sys/types.h>
#include<unistd.h>

定义函数 int fchown(int fd,uid_t owner,gid_t group);

函数说明 fchown()会将参数fd指定文件的所有者变更为参数owner代表的用户，而将该文件的组变更为参数group组。如果参数owner或group为-1，对映的所有者或组有所改变。参数fd 为已打开的文件描述词。当root用fchown()改变文件所有者或组时，该文件若具S_ISUID或S_ISGID权限，则会清除此权限位。

返回值 成功则返回0，失败则返回-1，错误原因存于errno。

错误代码 **EBADF** 参数fd文件描述词为无效的或该文件已关闭。

EPERM 进程的有效用户识别码与欲修改权限的文件所有者不同，而且也不具root权限，或是参数owner、group不正确。

EROFS 欲写入的文件存在于只读文件系统内。

ENOENT 指定的文件不存在

EIO I/O存取错误

范例

```
#include<sys/types.h>
#include<unistd.h>
#include<fcntl.h>
main()
{
    int fd;
    fd = open ("/etc/passwd",O_RDONLY);
    chown(fd,0,0);
    close(fd);
}
```

fstat（由文件描述词取得文件状态）

相关函数 **stat**, **lstat**, **chmod**, **chown**, **readlink**, **utime**

表头文件 **#include<sys/stat.h>**
#include<unistd.h>

定义函数 **int fstat(int fildes,struct stat *buf);**

函数说明 **fstat()**用来将参数fildes所指的文件状态，复制到参数buf所指的结构中(struct stat)。**Fstat()**与**stat()**作用完全相同，不同处在于传入的参数为已打开的文件描述词。详细内容请参考**stat()**。

返回值 执行成功则返回0，失败返回-1，错误代码存于**errno**。

范例

```
#include<sys/stat.h>
#include<unistd.h>
#include<fcntl.h>
main()
{
    struct stat buf;
    int fd;
    fd = open ("/etc/passwd",O_RDONLY);
    fstat(fd,&buf);
    printf("/etc/passwd file size +%d\n",buf.st_size);
}
```

执行 /etc/passwd file size = 705

ftruncate（改变文件大小）

相关函数 **open**, **truncate**

表头文件 **#include<unistd.h>**

定义函数 **int ftruncate(int fd,off_t length);**

函数说明 **ftruncate()**会将参数fd指定的文件大小改为参数length指定的大小。参数fd为已打开的文件描述词，而且必须是以写入模式打开的文件。如果原来的文件大小比参数length大，则超过的部分会被删去。

返回值 执行成功则返回0，失败返回-1，错误原因存于**errno**。

错误代码 **EBADF** 参数fd文件描述词为无效的或该文件已关闭。

EINVAL 参数fd 为一socket 并非文件，或是该文件并非以写入模式打开。

getcwd（取得当前的工作目录）

相关函数 get_current_dir_name, getwd, chdir

表头文件 #include<unistd.h>

定义函数 char * getcwd(char * buf,size_t size);

函数说明 getcwd()会将当前的工作目录绝对路径复制到参数buf所指的内存空间，参数size为buf的空间大小。在调用此函数时，buf所指的内存空间要足够大，若工作目录绝对路径的字符串长度超过参数size大小，则返回值NULL，errno的值则为ERANGE。倘若参数buf为NULL，getcwd()会依参数size的大小自动配置内存(使用malloc())，如果参数size也为0，则getcwd()会依工作目录绝对路径的字符串长度来决定所配置的内存大小，进程可以在使用完此字符串后利用free()来释放此空间。

返回值 执行成功则将结果复制到参数buf所指的内存空间，或是返回自动配置的字符串指针。失败返回NULL，错误代码存于errno。

```
范例 #include<unistd.h>
main()
{
    char buf[80];
    getcwd(buf,sizeof(buf));
    printf("current working directory : %s\n",buf);
}
```

执行 current working directory :/tmp

link（建立文件连接）

相关函数 symlink, unlink

表头文件 #include<unistd.h>

定义函数 int link (const char * oldpath,const char * newpath);

函数说明 link()以参数newpath指定的名称来建立一个新的连接(硬连接)到参数oldpath所指定的已存在文件。如果参数newpath指定的名称为一已存在的文件则不会建立连接。

返回值 成功则返回0，失败返回-1，错误原因存于errno。

附加说明 link()所建立的硬连接无法跨越不同文件系统，如果需要请改用symlink()。

错误代码 EXDEV 参数oldpath与newpath不是建立在同一文件系统。
EPERM 参数oldpath与newpath所指的文件系统不支持硬连接
EROFS 文件存在于只读文件系统内
EFAULT 参数oldpath或newpath 指针超出可存取内存空间。
ENAMETOLONG 参数oldpath或newpath太长
ENOMEM 核心内存不足
EEXIST 参数newpath所指的文件名已存在。
EMLINK 参数oldpath所指的文件已达最大连接数目。
ELOOP 参数pathname有过多符号连接问题
ENOSPC 文件系统的剩余空间不足。
EIO I/O 存取错误。

```
范例 /* 建立/etc/passwd 的硬连接为pass */
#include<unistd.h>
main()
{
    link("/etc/passwd","pass");
}
```

lstat（由文件描述词取得文件状态）

相关函数 **stat**, **fstat**, **chmod**, **chown**, **readlink**, **utime**

表头文件 **#include<sys/stat.h>**
#include<unistd.h>

定义函数 **int lstat (const char * file_name, struct stat * buf);**

函数说明 **lstat()**与**stat()**作用完全相同，都是取得参数**file_name**所指的文件状态，其差别在于，当文件为符号连接时，**lstat()**会返回该link本身的状态。详细内容请参考**stat()**。

返回值 执行成功则返回0，失败返回-1，错误代码存于**errno**。

范例 参考**stat()**。

opendir（打开目录）

相关函数 **open**, **readdir**, **closedir**, **rewinddir**, **seekdir**, **telldir**, **scandir**

表头文件 **#include<sys/types.h>**
#include<dirent.h>

定义函数 **DIR * opendir(const char * name);**

函数说明 **opendir()**用来打开参数**name**指定的目录，并返回**DIR***形态的目录流，和**open()**类似，接下来对目录的读取和搜索都要使用此返回值。

返回值 成功则返回**DIR*** 型态的目录流，打开失败则返回**NULL**。

错误代码 **EACCESS** 权限不足

EMFILE 已达到进程可同时打开的文件数上限。

ENFILE 已达到系统可同时打开的文件数上限。

ENOTDIR 参数**name**非真正的目录

ENOENT 参数**name** 指定的目录不存在，或是参数**name** 为一空字符串。

ENOMEM 核心内存不足。

readdir（读取目录）

相关函数 **open**, **opendir**, **closedir**, **rewinddir**, **seekdir**, **telldir**, **scandir**

表头文件 **#include<sys/types.h>**
#include<dirent.h>

定义函数 **struct dirent * readdir(DIR * dir);**

函数说明 **readdir()**返回参数**dir**目录流的下个目录进入点。

结构**dirent**定义如下

struct dirent

{

ino_t d_ino;

ff_t d_off;

signed short int d_reclen;

unsigned char d_type;

char d_name[256];

};

d_ino 此目录进入点的inode

d_off 目录文件开头至此目录进入点的位移

d_reclen d_name的长度，不包含**NULL**字符

d_type d_name 所指的文件类型

d_name 文件名

返回值 成功则返回下个目录进入点。有错误发生或读取到目录文件尾则返回**NULL**。

附加说明 EBADF参数dir为无效的目录流。

```
范例 #include<sys/types.h>
#include<dirent.h>
#include<unistd.h>
main()
{
    DIR * dir;
    struct dirent * ptr;
    int i;
    dir = opendir("/etc/rc.d");
    while((ptr = readdir(dir))!=NULL)
    {
        printf("d_name: %s\n",ptr->d_name);
    }
    closedir(dir);
}
```

执行 d_name:..
d_name:..
d_name:init.d
d_name:rc0.d
d_name:rc1.d
d_name:rc2.d
d_name:rc3.d
d_name:rc4.d
d_name:rc5.d
d_name:rc6.d
d_name:rc
d_name:rc.local
d_name:rc.sysinit

readlink（取得符号连接所指的文件）

相关函数 stat, lstat, symlink

表头文件 #include<unistd.h>

定义函数 int readlink(const char * path ,char * buf,size_t bufsiz);

函数说明 readlink()会将参数path的符号连接内容存到参数buf所指的内存空间，返回的内容不是以NULL作字符串结尾，但会将字符串的字符数返回。若参数bufsiz小于符号连接的内容长度，过长的内容会被截断。

返回值 执行成功则传符号连接所指的文件路径字符串，失败则返回-1，错误代码存于errno。

错误代码 EACCESS 取文件时被拒绝，权限不够

EINVAL 参数bufsiz 为负数

EIO I/O 存取错误。

ELOOP 欲打开的文件有过多符号连接问题。

ENAMETOOLONG 参数path的路径名称太长

ENOENT 参数path所指定的文件不存在

ENOMEM 核心内存不足

ENOTDIR 参数path路径中的目录存在但却非真正的目录。

remove（删除文件）

相关函数 link, rename, unlink

表头文件 `#include <stdio.h>`

定义函数 `int remove(const char * pathname);`

函数说明 `remove()`会删除参数`pathname`指定的文件。如果参数`pathname`为一文件，则调用`unlink()`处理，若参数`pathname`为一目录，则调用`rmdir()`来处理。请参考`unlink()`与`rmdir()`。

返回值 成功则返回0，失败则返回-1，错误原因存于`errno`。

错误代码 EROFS 欲写入的文件存在于只读文件系统内
EFAULT 参数`pathname`指针超出可存取内存空间
ENAMETOOLONG 参数`pathname`太长
ENOMEM 核心内存不足
ELOOP 参数`pathname`有过多符号连接问题
EIO I/O 存取错误。

`rename`（更改文件名称或位置）

相关函数 `link`, `unlink`, `symlink`

表头文件 `#include <stdio.h>`

定义函数 `int rename(const char * oldpath,const char * newpath);`

函数说明 `rename()`会将参数`oldpath`所指定的文件名称改为参数`newpath`所指的文件名称。若`newpath`所指定的文件已存在，则会被删除。

返回值 执行成功则返回0，失败返回-1，错误原因存于`errno`

范例 /* 设计一个DOS下的rename指令rename 旧文件名新文件名*/
`#include <stdio.h>`
`void main(int argc,char **argv)`
`{`
`if(argc<3){`
`printf("Usage: %s old_name new_name\n",argv[0]);`
`return;`
`}`
`printf("%s=>%s",argc[1],argv[2]);`
`if(rename(argv[1],argv[2]<0)`
`printf("error!\n");`
`else`
`printf("ok!\n");`
`}`

`rewinddir`（重设读取目录的位置为开头位置）

相关函数 `open`, `opendir`, `closedir`, `telldir`, `seekdir`, `readdir`, `scandir`

表头文件 `#include <sys/types.h>`

`#include <dirent.h>`

定义函数 `void rewinddir(DIR *dir);`

函数说明 `rewinddir()`用来设置参数`dir` 目录流目前的读取位置为原来开头的读取位置。

返回值

错误代码 EBADF `dir`为无效的目录流

范例 `#include <sys/types.h>`
`#include <dirent.h>`
`#include <unistd.h>`
`main()`
`{`

```

DIR * dir;
struct dirent *ptr;
dir = opendir("/etc/rc.d");
while((ptr = readdir(dir))!=NULL)
{
printf("d_name :%s\n",ptr->d_name);
}
rewinddir(dir);
printf("readdir again!\n");
while((ptr = readdir(dir))!=NULL)
{
printf("d_name: %s\n",ptr->d_name);
}
closedir(dir);
}

```

执行 d_name:..
d_name:..
d_name:init.d
d_name:rc0.d
d_name:rc1.d
d_name:rc2.d
d_name:rc3.d
d_name:rc4.d
d_name:rc5.d
d_name:rc6.d
d_name:rc
d_name:rc.local
d_name:rc.sysinit
readdir again!
d_name:..
d_name:..
d_name:init.d
d_name:rc0.d
d_name:rc1.d
d_name:rc2.d
d_name:rc3.d
d_name:rc4.d
d_name:rc5.d
d_name:rc6.d
d_name:rc
d_name:rc.local
d_name:rc.sysinit

seekdir（设置下回读取目录的位置）

相关函数 open, opendir, closedir, rewinddir, telldir, readdir, scandir

表头文件 #include<dirent.h>

定义函数 void seekdir(DIR * dir,off_t offset);

函数说明 seekdir()用来设置参数dir目录流目前的读取位置，在调用readdir()时便从此新位置开始读取。参数 offset 代表距离目录文件开头的偏移量。

返回值

错误代码 EBADF 参数dir为无效的目录流

范例 #include<sys/types.h>

```

#include<dirent.h>
#include<unistd.h>
main()
{
    DIR * dir;
    struct dirent * ptr;
    int offset,offset_5,i=0;
    dir=opendir("/etc/rc.d");
    while((ptr = readdir(dir))!=NULL)
    {
        offset = telldir(dir);
        if(++i == 5) offset_5 =offset;
        printf("d_name :%s offset :%d \n",ptr->d_name,offset);
    }
    seekdir(dir offset_5);
    printf("Readdir again!\n");
    while((ptr = readdir(dir))!=NULL)
    {
        offset = telldir(dir);
        printf("d_name :%s offset :%d\n",ptr->d_name.offset);
    }
    closedir(dir);
}

```

执行

```

d_name : . offset :12
d_name : .. offset:24
d_name : init.d offset 40
d_name : rc0.d offset :56
d_name :rc1.d offset :72
d_name:rc2.d offset :88
d_name:rc3.d offset 104
d_name:rc4.d offset:120
d_name:rc5.d offset:136
d_name:rc6.d offset:152
d_name:rc offset 164
d_name:rc.local offset :180
d_name:rc.sysinit offset :4096
readdir again!
d_name:rc2.d offset :88
d_name:rc3.d offset 104
d_name:rc4.d offset:120
d_name:rc5.d offset:136
d_name:rc6.d offset:152
d_name:rc offset 164
d_name:rc.local offset :180
d_name:rc.sysinit offset :4096

```

stat（取得文件状态）

相关函数 fstat, lstat, chmod, chown, readlink, utime

表头文件 #include<sys/stat.h>
#include<unistd.h>

定义函数 int stat(const char * file_name,struct stat *buf);

函数说明 stat()用来将参数file_name所指的文件状态，复制到参数buf所指的结构中。

下面是struct stat内各参数的说明

```
struct stat
{
dev_t st_dev; /*device*/
ino_t st_ino; /*inode*/
mode_t st_mode; /*protection*/
nlink_t st_nlink; /*number of hard links */
uid_t st_uid; /*user ID of owner*/
gid_t st_gid; /*group ID of owner*/
dev_t st_rdev; /*device type */
off_t st_size; /*total size, in bytes*/
unsigned long st_blksize; /*blocksize for filesystem I/O */
unsigned long st_blocks; /*number of blocks allocated*/
time_t st_atime; /* time of lastaccess*/
time_t st_mtime; /* time of last modification */
time_t st_ctime; /* time of last change */
};
```

st_dev 文件的设备编号

st_ino 文件的i-node

st_mode 文件的类型和存取的权限

st_nlink 连到该文件的硬连接数目，刚建立的文件值为1。

st_uid 文件所有者的用户识别码

st_gid 文件所有者的组织识别码

st_rdev 若此文件为装置设备文件，则为其设备编号

st_size 文件大小，以字节计算

st_blksize 文件系统的I/O 缓冲区大小。

st_blocks 占用文件区块的个数，每一区块大小为512 个字节。

st_atime 文件最近一次被存取或被执行的时间，一般只有在用mknod、utime、read、write与truncate时改变。

st_mtime 文件最后一次被修改的时间，一般只有在用mknod、utime和write时才会改变

st_ctime i-node最近一次被更改的时间，此参数会在文件所有者、组、权限被更改时更新先前所描述的st_mode 则定义了下列数种情况

S_IFMT 0170000 文件类型的位遮罩

S_IFSOCK 0140000 socket

S_IFLNK 0120000 符号连接

S_IFREG 0100000 一般文件

S_IFBLK 0060000 区块装置

S_IFDIR 0040000 目录

S_IFCHR 0020000 字符装置

S_IFIFO 0010000 先进先出

S_ISUID 04000 文件的（set user-id on execution）位

S_ISGID 02000 文件的（set group-id on execution）位

S_ISVTX 01000 文件的sticky位

S_IRUSR（S_IREAD）00400 文件所有者具可读取权限

S_IWUSR（S_IWRITE）00200 文件所有者具可写入权限

S_IXUSR（S_IEXEC）00100 文件所有者具可执行权限

S_IRGRP 00040 用户组具可读取权限

S_IWGRP 00020 用户组具可写入权限

S_IXGRP 00010 用户组具可执行权限

S_IROTH 00004 其他用户具可读取权限

S_IWOTH 00002 其他用户具可写入权限

S_IXOTH 00001 其他用户具可执行权限

上述的文件类型在POSIX 中定义了检查这些类型的宏定义

S_ISLNK（st_mode）判断是否为符号连接

S_ISREG（st_mode）是否为一般文件

S_ISDIR（st_mode）是否为目录

S_ISCHR（st_mode）是否为字符装置文件

S_ISBLK (s3e) 是否为先进先出

S_ISSOCK (st_mode) 是否为socket

若一目录具有sticky 位 (S_ISVTX)，则表示在此目录下的文件只能被该文件所有者、此目录所有者或root来删除或改名。

返回值 执行成功则返回0，失败返回-1，错误代码存于errno

错误代码 ENOENT 参数file_name指定的文件不存在

ENOTDIR 路径中的目录存在但却非真正的目录

ELOOP 欲打开的文件有过多符号连接问题，上限为16符号连接

EFAULT 参数buf为无效指针，指向无法存在的内存空间

EACCESS 存取文件时被拒绝

ENOMEM 核心内存不足

ENAMETOOLONG 参数file_name的路径名称太长

范例 #include<sys/stat.h>

#include<unistd.h>

main()

{

struct stat buf;

stat ("/etc/passwd",&buf);

printf("/etc/passwd file size = %d \n",buf.st_size);

}

执行 /etc/passwd file size = 705

symlink (建立文件符号连接)

相关函数 link, unlink

表头文件 #include<unistd.h>

定义函数 int symlink(const char * oldpath,const char * newpath);

函数说明 symlink()以参数newpath指定的名称来建立一个新的连接(符号连接)到参数oldpath所指定的已存在文件。参数oldpath指定的文件不一定要存在，如果参数newpath指定的名称为一已存在的文件则不会建立连接。

返回值 成功则返回0，失败返回-1，错误原因存于errno。

错误代码 EPERM 参数oldpath与newpath所指的文件系统不支持符号连接

EROFS 欲测试写入权限的文件存在于只读文件系统内

EFAULT 参数oldpath或newpath指针超出可存取内存空间。

ENAMETOOLONG 参数oldpath或newpath太长

ENOMEM 核心内存不足

EEXIST 参数newpath所指的文件名已存在。

EMLINK 参数oldpath所指的文件已达到最大连接数目

ELOOP 参数pathname有过多符号连接问题

ENOSPC 文件系统的剩余空间不足

EIO I/O 存取错误

范例 #include<unistd.h>

main()

{

symlink("/etc/passwd","pass");

}

telldir (取得目录流的读取位置)

相关函数 open, opendir, closedir, rewinddir, seekdir, readdir, scandir

表头文件 `#include<dirent.h>`

定义函数 `off_t telldir(DIR *dir);`

函数说明 `telldir()`返回参数`dir`目录流目前的读取位置。此返回值代表距离目录文件开头的偏移量返回值返回下一个读取位置，有错误发生时返回-1。

错误代码 `EBADF`参数`dir`为无效的目录流。

```
范例 #include<sys/types.h>
#include<dirent.h>
#include<unistd.h>
main()
{
    DIR *dir;
    struct dirent *ptr;
    int offset;
    dir = opendir("/etc/rc.d");
    while((ptr = readdir(dir))!=NULL)
    {
        offset = telldir (dir);
        printf("d_name : %s offset :%d\n", ptr->d_name,offset);
    }
    closedir(dir);
}
```

```
执行 d_name : . offset :12
d_name : .. offset:24
d_name : init.d offset 40
d_name : rc0.d offset :56
d_name :rc1.d offset :72
d_name:rc2.d offset :88
d_name:rc3.d offset 104
d_name:rc4.d offset:120
d_name:rc5.d offset:136
d_name:rc6.d offset:152
d_name:rc offset 164
d_name:rc.local offset :180
d_name:rc.sysinit offset :4096
```

`truncate`（改变文件大小）

相关函数 `open`, `ftruncate`

表头文件 `#include<unistd.h>`

定义函数 `int truncate(const char * path,off_t length);`

函数说明 `truncate()`会将参数`path`指定的文件大小改为参数`length`指定的大小。如果原来的文件大小比参数`length`大，则超过的部分会被删去。

返回值 执行成功则返回0，失败返回-1，错误原因存于`errno`。

错误代码 `EACCESS` 参数`path`所指定的文件无法存取。

`EROFS` 欲写入的文件存在于只读文件系统内

`EFAULT` 参数`path`指针超出可存取内存空间

`EINVAL` 参数`path`包含不合法字符

`ENAMETOOLONG` 参数`path`太长

`ENOTDIR` 参数`path`路径并非一目录

`EISDIR` 参数`path` 指向一目录

`ETXTBUSY` 参数`path`所指的文件为共享程序，而且正被执行中

`ELOOP` 参数`path`有过多符号连接问题

`EIO` I/O 存取错误。

umask（设置建立新文件时的权限遮罩）

相关函数 creat, open

表头文件 #include<sys/types.h>
#include<sys/stat.h>

定义函数 mode_t umask(mode_t mask);

函数说明 umask()会将系统umask值设成参数mask&0777后的值，然后将先前的umask值返回。在使用open()建立新文件时，该参数mode并非真正建立文件的权限，而是(mode&~umask)的权限值。例如，在建立文件时指定文件权限为0666，通常umask值默认为022，则该文件的真正权限则为0666&~022=0644，也就是rw-r--r--。返回值此调用不会有错误值返回。返回值为原先系统的umask值。

unlink（删除文件）

相关函数 link, rename, remove

表头文件 #include<unistd.h>

定义函数 int unlink(const char * pathname);

函数说明 unlink()会删除参数pathname指定的文件。如果该文件名为最后连接点，但还有其他进程打开了此文件，则在所有关于此文件的文件描述词皆关闭后才会删除。如果参数pathname为一符号连接，则此连接会被删除。

返回值 成功则返回0，失败返回-1，错误原因存于errno

错误代码 EROFS 文件存在于只读文件系统内
EFAULT 参数pathname指针超出可存取内存空间
ENAMETOOLONG 参数pathname太长
ENOMEM 核心内存不足
ELOOP 参数pathname 有过多符号连接问题
EIO I/O 存取错误

utime（修改文件的存取时间和更改时间）

相关函数 utimes, stat

表头文件 #include<sys/types.h>
#include<utime.h>

定义函数 int utime(const char * filename,struct utimbuf * buf);

函数说明 utime()用来修改参数filename文件所属的inode存取时间。

结构utimbuf定义如下

```
struct utimbuf{
    time_t actime;
    time_t modtime;
};
```

返回值 如果参数buf为空指针(NULL)，则该文件的存取时间和更改时间全部会设为目前时间。执行成功则返回0，失败返回-1，错误代码存于errno。

错误代码 EACCESS 存取文件时被拒绝，权限不足
ENOENT 指定的文件不存在。

utimes（修改文件的存取时间和更改时间）

相关函数 `utime`，`stat`

表头文件 `#include <sys/types.h>`
`#include <utime.h>`

定义函数 `int utimes(char * filename, struct timeval *tvp);`

函数说明 `utimes()`用来修改参数`filename`文件所属的inode存取时间和修改时间。

结构`timeval`定义如下

```
struct timeval {  
    long tv_sec;  
    long tv_usec; /* 微妙*/  
};
```

返回值 参数`tvp` 指向两个`timeval` 结构空间，和`utime（）`使用的`utimbuf`结构比较，`tvp[0].tv_sec` 则为`utimbuf.actime`，`tvp[1].tv_sec` 为`utimbuf.modtime`。

执行成功则返回0。失败返回-1，错误代码存于`errno`。

错误代码 `EACCESS` 存取文件时被拒绝，权限不足

`ENOENT` 指定的文件不存在

alarm (设置信号传送闹钟)

相关函数 signal, sleep

表头文件 #include<unistd.h>

定义函数 unsigned int alarm(unsigned int seconds);

函数说明 alarm()用来设置信号SIGALRM在经过参数seconds指定的秒数后传送给目前的进程。如果参数seconds 为0, 则之前设置的闹钟会被取消, 并将剩下的时间返回。

返回值 返回之前闹钟的剩余秒数, 如果之前未设闹钟则返回0。

```
范例 #include<unistd.h>
#include<signal.h>
void handler() {
    printf("hello\n");
}
main()
{
    int i;
    signal(SIGALRM,handler);
    alarm(5);
    for(i=1;i<7;i++){
        printf("sleep %d ...\n",i);
        sleep(1);
    }
}
```

```
执行 sleep 1 ...
sleep 2 ...
sleep 3 ...
sleep 4 ...
sleep 5 ...
hello
sleep 6 ...
```

kill (传送信号给指定的进程)

相关函数 raise, signal

表头文件 #include<sys/types.h>

#include<signal.h>

定义函数 int kill(pid_t pid,int sig);

函数说明 kill()可以用来送参数sig指定的信号给参数pid指定的进程。参数pid有几种情况:

pid>0 将信号传给进程识别码为pid 的进程。

pid=0 将信号传给和目前进程相同进程组的所有进程

pid=-1 将信号广播传送给系统内所有的进程

pid<0 将信号传给进程组识别码为pid绝对值的所有进程

参数sig代表的信号编号可参考附录D

返回值 执行成功则返回0, 如果有错误则返回-1。

错误代码 EINVAL 参数sig 不合法

ESRCH 参数pid 所指定的进程或进程组不存在

EPERM 权限不够无法传送信号给指定进程

```

范例 #include<unistd.h>
      #include<signal.h>
      #include<sys/types.h>
      #include<sys/wait.h>
      main()
      {
        pid_t pid;
        int status;
        if(!(pid= fork())){
          printf("Hi I am child process!\n");
          sleep(10);
          return;
        }
        else{
          printf("send signal to child process (%d) \n",pid);
          sleep(1);
          kill(pid ,SIGABRT);
          wait(&status);
          if(WIFSIGNALED(status))
            printf("chile process receive signal %d\n",WTERMSIG(status));
        }
      }

```

执行 sen signal to child process(3170)
 Hi I am child process!
 child process receive signal 6

pause（让进程暂停直到信号出现）

相关函数 kill, signal, sleep

表头文件 #include<unistd.h>

定义函数 int pause(void);

函数说明 pause()会令目前的进程暂停（进入睡眠状态），直到被信号(signal)所中断。

返回值 只返回-1。

错误代码 EINTR 有信号到达中断了此函数。

sigaction（查询或设置信号处理方式）

相关函数 signal, sigprocmask, sigpending, sigsuspend

表头文件 #include<signal.h>

定义函数 int sigaction(int signum,const struct sigaction *act ,struct sigaction *oldact);

函数说明 sigaction()会依参数signum指定的信号编号来设置该信号的处理函数。参数signum可以指定SIGKILL和SIGSTOP以外的所有信号。

如参数结构sigaction定义如下

```

struct sigaction
{
  void (*sa_handler) (int);
  sigset_t sa_mask;
  int sa_flags;
  void (*sa_restorer) (void);
}

```

sa_handler此参数和signal()的参数handler相同，代表新的信号处理函数，其他意义请参考signal()。

sa_mask 用来设置在处理该信号时暂时将sa_mask 指定的信号搁置。

sa_restorer 此参数没有使用。

sa_flags 用来设置信号处理的其他相关操作，下列的数值可用。

OR 运算 (|) 组合

A_NOCLDSTOP: 如果参数sigum为SIGCHLD，则当子进程暂停时并不会通知父进程

SA_ONESHOT/SA_RESETHAND:当调用新的信号处理函数前，将此信号处理方式改为系统预设的方式。

SA_RESTART:被信号中断的系统调用会自行重启

SA_NOMASK/SA_NODEFER:在处理此信号未结束前不理睬此信号的再次到来。

如果参数oldact不是NULL指针，则原来的信号处理方式会由此结构sigaction 返回。

返回值 执行成功则返回0，如果有错误则返回-1。

错误代码 EINVAL 参数sigum 不合法，或是企图拦截SIGKILL/SIGSTOPSIGKILL信号

EFAULT 参数act，oldact指针地址无法存取。

EINTR 此调用被中断

```
范例 #include<unistd.h>
#include<signal.h>
void show_handler(struct sigaction * act)
{
    switch (act->sa_flags)
    {
        case SIG_DFL:printf("Default action\n");break;
        case SIG_IGN:printf("Ignore the signal\n");break;
        default: printf("0x%x\n",act->sa_handler);
    }
}
main()
{
    int i;
    struct sigaction act,oldact;
    act.sa_handler = show_handler;
    act.sa_flags = SA_ONESHOT|SA_NOMASK;
    sigaction(SIGUSR1,&act,&oldact);
    for(i=5;i<15;i++)
    {
        printf("sa_handler of signal %2d =".i);
        sigaction(i,NULL,&oldact);
    }
}
```

执行 sa_handler of signal 5 = Default action
sa_handler of signal 6= Default action
sa_handler of signal 7 = Default action
sa_handler of signal 8 = Default action
sa_handler of signal 9 = Default action
sa_handler of signal 10 = 0x8048400
sa_handler of signal 11 = Default action
sa_handler of signal 12 = Default action
sa_handler of signal 13 = Default action
sa_handler of signal 14 = Default action

sigaddset（增加一个信号至信号集）

相关函数 sigemptyset, sigfillset, sigdelset, sigismember

表头文件 #include<signal.h>

定义函数 `int sigaddset(sigset_t *set,int signum);`

函数说明 `sigaddset()`用来将参数`signum` 代表的信号加入至参数`set` 信号集里。

返回值 执行成功则返回0，如果有错误则返回-1。

错误代码 `EFAULT` 参数`set`指针地址无法存取

`EINVAL` 参数`signum`非合法的信号编号

`sigdelset`（从信号集里删除一个信号）

相关函数 `sigemptyset`, `sigfillset`, `sigaddset`, `sigismember`

表头文件 `#include<signal.h>`

定义函数 `int sigdelset(sigset_t * set,int signum);`

函数说明 `sigdelset()`用来将参数`signum`代表的信号从参数`set`信号集里删除。

返回值 执行成功则返回0，如果有错误则返回-1。

错误代码 `EFAULT` 参数`set`指针地址无法存取

`EINVAL` 参数`signum`非合法的信号编号

`sigemptyset`（初始化信号集）

相关函数 `sigaddset`, `sigfillset`, `sigdelset`, `sigismember`

表头文件 `#include<signal.h>`

定义函数 `int sigemptyset(sigset_t *set);`

函数说明 `sigemptyset()`用来将参数`set`信号集初始化并清空。

返回值 执行成功则返回0，如果有错误则返回-1。

错误代码 `EFAULT` 参数`set`指针地址无法存取

`sigfillset`（将所有信号加入至信号集）

相关函数 `sigempty`, `sigaddset`, `sigdelset`, `sigismember`

表头文件 `#include<signal.h>`

定义函数 `int sigfillset(sigset_t * set);`

函数说明 `sigfillset()`用来将参数`set`信号集初始化，然后把所有的信号加入到此信号集里。

返回值 执行成功则返回0，如果有错误则返回-1。

附加说明 `EFAULT` 参数`set`指针地址无法存取

`sigismember`（测试某个信号是否已加入至信号集里）

相关函数 `sigemptyset`, `sigfillset`, `sigaddset`, `sigdelset`

表头文件 `#include<signal.h>`

定义函数 `int sigismember(const sigset_t *set,int signum);`

函数说明 `sigismember()`用来测试参数`signum` 代表的信号是否已加入至参数`set`信号集里。如果信号集里已有该信号则返回1，否则返回0。

返回值 信号集已有该信号则返回1，没有则返回0。如果有错误则返回-1。

错误代码 `EFAULT` 参数`set`指针地址无法存取

`EINVAL` 参数`signum` 非合法的信号编号

signal（设置信号处理方式）

相关函数 sigaction, kill, raise

表头文件 #include<signal.h>

定义函数 void (*signal(int signum,void(* handler)(int)))(int);

函数说明 signal()会依参数signum 指定的信号编号来设置该信号的处理函数。当指定的信号到达时就会跳转到参数handler指定的函数执行。如果参数handler不是函数指针，则必须是下列两个常数之一：

SIG_IGN 忽略参数signum指定的信号。

SIG_DFL 将参数signum 指定的信号重设为核心预设的信号处理方式。

关于信号的编号和说明，请参考附录D

返回值 返回先前的信号处理函数指针，如果有错误则返回SIG_ERR(-1)。

附加说明 在信号发生跳转到自定的handler处理函数执行后，系统会自动将此处理函数换回原来系统预设的处理方式，如果要改变此操作请改用sigaction()。

范例 参考alarm()或raise()。

sigpending（查询被搁置的信号）

相关函数 signal, sigaction, sigprocmask, sigsuspend

表头文件 #include<signal.h>

定义函数 int sigpending(sigset_t *set);

函数说明 sigpending()会将搁置的信号集合由参数set指针返回。

返回值 执行成功则返回0，如果有错误则返回-1。

错误代码 EFAULT 参数set指针地址无法存取

EINTR 此调用被中断。

sigprocmask（查询或设置信号遮罩）

相关函数 signal, sigaction, sigpending, sigsuspend

表头文件 #include<signal.h>

定义函数 int sigprocmask(int how,const sigset_t *set,sigset_t * oldset);

函数说明 sigprocmask()可以用来改变目前的信号遮罩，其操作依参数how来决定

SIG_BLOCK 新的信号遮罩由目前的信号遮罩和参数set 指定的信号遮罩作联集

SIG_UNBLOCK 将目前的信号遮罩删除掉参数set指定的信号遮罩

SIG_SETMASK 将目前的信号遮罩设成参数set指定的信号遮罩。

如果参数oldset不是NULL指针，那么目前的信号遮罩会由此指针返回。

返回值 执行成功则返回0，如果有错误则返回-1。

错误代码 EFAULT 参数set, oldset指针地址无法存取。

EINTR 此调用被中断

sleep（让进程暂停执行一段时间）

相关函数 signal, alarm

表头文件 #include<unistd.h>

定义函数 unsigned int sleep(unsigned int seconds);

函数说明 `sleep()` 会令目前的进程暂停，直到达到参数 `seconds` 所指定的时间，或是被信号所中断。

返回值 若进程暂停到参数 `seconds` 所指定的时间则返回0，若有信号中断则返回剩余秒数。

`ferror`（检查文件流是否有错误发生）

相关函数 `clearerr`, `perror`

表头文件 `#include <stdio.h>`

定义函数 `int ferror(FILE *stream);`

函数说明 `ferror()` 用来检查参数 `stream` 所指定的文件流是否发生了错误情况，如有错误发生则返回非0值。

返回值 如果文件流有错误发生则返回非0值。

`perror`（打印出错误原因信息字符串）

相关函数 `strerror`

表头文件 `#include <stdio.h>`

定义函数 `void perror(const char *s);`

函数说明 `perror()` 用来将上一个函数发生错误的原因输出到标准错误(`stderr`)。参数 `s` 所指的字符串会先打印出，后面再加上错误原因字符串。此错误原因依照全局变量 `errno` 的值来决定要输出的字符串。

返回值

范例 `#include <stdio.h>`

```
main()
{
    FILE *fp;
    fp = fopen("/tmp/noexist", "r+");
    if(fp == NULL) perror("fopen");
}
```

执行 `$./perror`

`fopen : No such file or diretory`

`strerror`（返回错误原因的描述字符串）

相关函数 `perror`

表头文件 `#include <string.h>`

定义函数 `char * strerror(int errnum);`

函数说明 `strerror()` 用来依参数 `errnum` 的错误代码来查询其错误原因的描述字符串，然后将该字符串指针返回。

返回值 返回描述错误原因的字符串指针。

范例 `/* 显示错误代码0 至9 的错误原因描述*/`

```
#include <string.h>
main()
{
    int i;
    for(i=0; i<10; i++)
        printf("%d : %s\n", i, strerror(i));
}
```

执行 `0 : Success`

`1 : Operation not permitted`

2 : No such file or directory
3 : No such process
4 : Interrupted system call
5 : Input/output error
6 : Device not configured
7 : Argument list too long
8 : Exec format error
9 : Bad file descriptor

mkfifo (建立具名管道)

相关函数 pipe, popen, open, umask

表头文件 #include<sys/types.h>
#include<sys/stat.h>

定义函数 int mkfifo(const char * pathname,mode_t mode);

函数说明 mkfifo()会依参数pathname建立特殊的FIFO文件，该文件必须不存在，而参数mode为该文件的权限（mode%~umask），因此umask值也会影响到FIFO文件的权限。Mkfifo()建立的FIFO文件其他进程都可以用读写一般文件的方式存取。当使用open()来打开FIFO文件时，O_NONBLOCK旗标会有影响
1、当使用O_NONBLOCK 旗标时，打开FIFO 文件来读取的操作会立刻返回，但是若还没有其他进程打开FIFO 文件来读取，则写入的操作会返回ENXIO 错误代码。
2、没有使用O_NONBLOCK 旗标时，打开FIFO 来读取的操作会等到其他进程打开FIFO文件来写入才正常返回。同样地，打开FIFO文件来写入的操作会等到其他进程打开FIFO 文件来读取后才正常返回。

返回值 若成功则返回0，否则返回-1，错误原因存于errno中。

错误代码 EACCESS 参数pathname所指定的目录路径无可执行的权限
EEXIST 参数pathname所指定的文件已存在。
ENAMETOOLONG 参数pathname的路径名称太长。
ENOENT 参数pathname包含的目录不存在
ENOSPC 文件系统的剩余空间不足
ENOTDIR 参数pathname路径中的目录存在但却非真正的目录。
EROFS 参数pathname指定的文件存在于只读文件系统内。

范例 #include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
main()
{
char buffer[80];
int fd;
unlink(FIFO);
mkfifo(FIFO,0666);
if(fork()>0){
char s[] = "hello!\n";
fd = open (FIFO,O_WRONLY);
write(fd,s,sizeof(s));
close(fd);
}
else{
fd= open(FIFO,O_RDONLY);
read(fd,buffer,80);
printf("%s",buffer);
close(fd);
}
}

执行 hello!

pclose（关闭管道I/O）

相关函数 popen

表头文件 #include<stdio.h>

定义函数 int pclose(FILE * stream);

函数说明 pclose()用来关闭由popen所建立的管道及文件指针。参数stream为先前由popen()所返回的文件指针。

返回值 返回子进程的结束状态。如果有错误则返回-1，错误原因存于errno中。

错误代码 ECHILD pclose()无法取得子进程的结束状态。

范例 参考popen()。

pipe（建立管道）

相关函数 mkfifo, popen, read, write, fork

表头文件 #include<unistd.h>

定义函数 int pipe(int filed[2]);

函数说明 pipe()会建立管道，并将文件描述词由参数filed数组返回。filed[0]为管道里的读取端，filed[1]则为管道的写入端。

返回值 若成功则返回零，否则返回-1，错误原因存于errno中。

错误代码 EMFILE 进程已用完文件描述词最大量。

ENFILE 系统已无文件描述词可用。

EFAULT 参数filed数组地址不合法。

范例 /* 父进程借管道将字符串"hello!\n"传给子进程并显示*/

```
#include <unistd.h>
main()
{
    int filed[2];
    char buffer[80];
    pipe(filed);
    if(fork()>0){
        /* 父进程*/
        char s[] = "hello!\n";
        write(filed[1],s,sizeof(s));
    }
    else{
        /*子进程*/
        read(filed[0],buffer,80);
        printf("%s",buffer);
    }
}
```

执行 hello!

popen（建立管道I/O）

相关函数 pipe, mkfifo, pclose, fork, system, fopen

表头文件 #include<stdio.h>

定义函数 `FILE * popen(const char * command,const char * type);`

函数说明 `popen()`会调用`fork()`产生子进程，然后从子进程中调用`/bin/sh -c`来执行参数`command`的指令。参数`type`可使用“r”代表读取，“w”代表写入。依照此`type`值，`popen()`会建立管道连到子进程的标准输出设备或标准输入设备，然后返回一个文件指针。随后进程便可利用此文件指针来读取子进程的输出设备或是写入到子进程的标准输入设备中。此外，所有使用文件指针(`FILE*`)操作的函数也都可以使用，除了`fclose()`以外。

返回值 若成功则返回文件指针，否则返回`NULL`，错误原因存于`errno`中。

错误代码 `EINVAL`参数`type`不合法。

注意事项 在编写具`SUID/SGID`权限的程序时请尽量避免使用`popen()`，`popen()`会继承环境变量，通过环境变量可能会造成系统安全的问题。

范例 `#include <stdio.h>`
`main()`
`{`
`FILE * fp;`
`char buffer[80];`
`fp=popen("cat /etc/passwd","r");`
`fgets(buffer,sizeof(buffer),fp);`
`printf("%s",buffer);`
`pclose(fp);`
`}`

执行 `root :x:0 0: root: /root: /bin/bash`

accept（接受socket连线）

相关函数 socket, bind, listen, connect

表头文件 `#include<sys/types.h>`
`#include<sys/socket.h>`

定义函数 `int accept(int s,struct sockaddr * addr,int * addrlen);`

函数说明 `accept()`用来接受参数s的socket连线。参数s的socket必需先经`bind()`、`listen()`函数处理过，当有连线进来时`accept()`会返回一个新的socket处理代码，往后的数据传送与读取就是经由新的socket处理，而原来参数s的socket能继续使用`accept()`来接受新的连线要求。连线成功时，参数addr所指的结构会被系统填入远程主机的地址数据，参数addrlen为sockaddr的结构长度。关于结构sockaddr的定义请参考`bind()`。

返回值 成功则返回新的socket处理代码，失败返回-1，错误原因存于errno中。

错误代码 EBADF 参数s 非合法socket处理代码。

EFAULT 参数addr指针指向无法存取的内存空间。

ENOTSOCK 参数s为一文件描述词，非socket。

EOPNOTSUPP 指定的socket并非SOCK_STREAM。

EPERM 防火墙拒绝此连线。

ENOBUFS 系统的缓冲内存不足。

ENOMEM 核心内存不足。

范例 参考listen()。

bind（对socket定位）

相关函数 socket, accept, connect, listen

表头文件 `#include<sys/types.h>`
`#include<sys/socket.h>`

定义函数 `int bind(int sockfd,struct sockaddr * my_addr,int addrlen);`

函数说明 `bind()`用来设置给参数sockfd的socket一个名称。此名称由参数my_addr指向一sockaddr结构，对于不同的socket domain定义了一个通用的数据结构

struct sockaddr

```
{  
    unsigned short int sa_family;  
    char sa_data[14];  
};
```

sa_family 为调用socket（）时的domain参数，即AF_xxxx值。

sa_data 最多使用14个字符长度。

此sockaddr结构会因使用不同的socket domain而有不同结构定义，例如使用AF_INET domain，其socketaddr结构定义便为

struct socketaddr_in

```
{  
    unsigned short int sin_family;  
    uint16_t sin_port;  
    struct in_addr sin_addr;  
    unsigned char sin_zero[8];  
};
```

struct in_addr

```
{  
    uint32_t s_addr;
```

```
};  
sin_family 即为sa_family  
sin_port 为使用的port编号  
sin_addr.s_addr 为IP 地址  
sin_zero 未使用。
```

参数 addrlen为sockaddr的结构长度。

返回值 成功则返回0，失败返回-1，错误原因存于errno中。

错误代码 EBADF 参数sockfd 非合法socket处理代码。

EACCESS 权限不足

ENOTSOCK 参数sockfd为一文件描述词，非socket。

范例 参考listen()

connect（建立socket连线）

相关函数 socket, bind, listen

表头文件 #include<sys/types.h>

#include<sys/socket.h>

定义函数 int connect (int sockfd,struct sockaddr * serv_addr,int addrlen);

函数说明 connect()用来将参数sockfd 的socket 连至参数serv_addr 指定的网络地址。结构sockaddr请参考bind()。参数addrlen为sockaddr的结构长度。

返回值 成功则返回0，失败返回-1，错误原因存于errno中。

错误代码 EBADF 参数sockfd 非合法socket处理代码

EFAULT 参数serv_addr指针指向无法存取的内存空间

ENOTSOCK 参数sockfd为一文件描述词，非socket。

EISCONN 参数sockfd的socket已是连线状态

ECONNREFUSED 连线要求被server端拒绝。

ETIMEDOUT 企图连线的操作超过限定时间仍未有响应。

ENETUNREACH 无法传送数据包至指定的主机。

EAFNOSUPPORT sockaddr结构的sa_family不正确。

EALREADY socket为不可阻断且先前的连线操作还未完成。

范例 /* 利用socket的TCP client

此程序会连线TCP server，并将键盘输入的字符串传送给server。

TCP server范例请参考listen（）。

*/

```
#include<sys/stat.h>
```

```
#include<fcntl.h>
```

```
#include<unistd.h>
```

```
#include<sys/types.h>
```

```
#include<sys/socket.h>
```

```
#include<netinet/in.h>
```

```
#include<arpa/inet.h>
```

```
#define PORT 1234
```

```
#define SERVER_IP "127.0.0.1"
```

```
main()
```

```
{
```

```
int s;
```

```
struct sockaddr_in addr;
```

```
char buffer[256];
```

```
if((s = socket(AF_INET,SOCK_STREAM,0))<0){
```

```
perror("socket");
```

```
exit(1);
```

```
}
```

```
/* 填写sockaddr_in结构*/
```



```

bzero(&addr,sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port=htons(PORT);
addr.sin_addr.s_addr = inet_addr(SERVER_IP);
/* 尝试连线*/
if(connect(s,&addr,sizeof(addr))<0){
perror("connect");
exit(1);
}
/* 接收由server端传来的信息*/
recv(s,buffer,sizeof(buffer),0);
printf("%s\n",buffer);
while(1){
bzero(buffer,sizeof(buffer));
/* 从标准输入设备取得字符串*/
read(STDIN_FILENO,buffer,sizeof(buffer));
/* 将字符串传给server端*/
if(send(s,buffer,sizeof(buffer),0)<0){
perror("send");
exit(1);
}
}
}

```

执行 \$./connect

```

Welcome to server!
hi I am client! /*键盘输入*/
/*<Ctrl+C>中断程序*/

```

endprotoent（结束网络协议数据的读取）

相关函数 getprotoent, getprotobyname, getprotobynumber, setprotoent

表头文件 #include<netdb.h>

定义函数 void endprotoent(void);

函数说明 endprotoent()用来关闭由getprotoent()打开的文件。

返回值

范例 参考getprotoent()

endservent（结束网络服务数据的读取）

相关函数 getservent, getservbyname, getservbyport, setservent

表头文件 #include<netdb.h>

定义函数 void endservent(void);

函数说明 endservent()用来关闭由getservent()所打开的文件。

返回值

范例 参考getservent()。

getsockopt（取得socket状态）

相关函数 setsockopt

表头文件 `#include<sys/types.h>`
`#include<sys/socket.h>`

定义函数 `int getsockopt(int s,int level,int optname,void* optval,socklen_t* optlen);`

函数说明 `getsockopt()`会将参数s所指定的socket状态返回。参数optname代表欲取得何种选项状态，而参数optval则指向欲保存结果的内存地址，参数optlen则为该空间的大小。参数level、optname请参考 `setsockopt()`。

返回值 成功则返回0，若有错误则返回-1，错误原因存于errno

错误代码 `EBADF` 参数s 并非合法的socket处理代码
`ENOTSOCK` 参数s为一文件描述词，非socket
`ENOPROTOOPT` 参数optname指定的选项不正确
`EFAULT` 参数optval指针指向无法存取的内存空间

范例 `#include<sys/types.h>`
`#include<sys/socket.h>`
`main()`
`{`
`int s,optval,optlen = sizeof(int);`
`if((s = socket(AF_INET,SOCK_STREAM,0))<0) perror("socket");`
`getsockopt(s,SOL_SOCKET,SO_TYPE,&optval,&optlen);`
`printf("optval = %d\n",optval);`
`close(s);}`

执行 `optval = 1 /*SOCK_STREAM的定义正是此值*/`

`htonl`（将32位主机字符顺序转换成网络字符顺序）

相关函数 `htons`, `ntohl`, `ntohs`

表头文件 `#include<netinet/in.h>`

定义函数 `unsigned long int htonl(unsigned long int hostlong);`

函数说明 `htonl()`用来将参数指定的32位hostlong 转换成网络字符顺序。

返回值 返回对应的网络字符顺序。

范例 参考 `getservbyport()`或 `connect()`。

`htons`（将16位主机字符顺序转换成网络字符顺序）

相关函数 `htonl`, `ntohl`, `ntohs`

表头文件 `#include<netinet/in.h>`

定义函数 `unsigned short int htons(unsigned short int hostshort);`

函数说明 `htons()`用来将参数指定的16位hostshort转换成网络字符顺序。

返回值 返回对应的网络字符顺序。

范例 参考 `connect()`。

`inet_addr`（将网络地址转成二进制的数字）

相关函数 `inet_aton`, `inet_ntoa`

表头文件 `#include<sys/socket.h>`
`#include<netinet/in.h>`
`#include<arpa/inet.h>`

定义函数 `unsigned long int inet_addr(const char *cp);`

函数说明 `inet_addr()`用来将参数`cp`所指的网路地址字符串转换成网路所使用的二进制数字。网路地址字符串是以数字和点组成的字符串，例如：“163.13.132.68”。

返回值 成功则返回对应的网路二进制的数字，失败返回-1。

`inet_pton`（将网路地址转成网路二进制的数字）

相关函数 `inet_addr`，`inet_ntoa`

表头文件 `#include<sys/socket.h>`
`#include<netinet/in.h>`
`#include<arpa/inet.h>`

定义函数 `int inet_pton(const char * cp,struct in_addr *inp);`

函数说明 `inet_pton()`用来将参数`cp`所指的网路地址字符串转换成网路使用的二进制的数字，然后存于参数`inp`所指的`in_addr`结构中。

结构`in_addr`定义如下

```
struct in_addr
{
    unsigned long int s_addr;
};
```

返回值 成功则返回非0值，失败则返回0。

`inet_ntop`（将网路二进制的数字转换成网路地址）

相关函数 `inet_addr`，`inet_pton`

表头文件 `#include<sys/socket.h>`
`#include<netinet/in.h>`
`#include<arpa/inet.h>`

定义函数 `char * inet_ntop(struct in_addr in);`

函数说明 `inet_ntop()`用来将参数`in`所指的网路二进制的数字转换成网路地址，然后将指向此网路地址字符串的指针返回。

返回值 成功则返回字符串指针，失败则返回NULL。

`listen`（等待连接）

相关函数 `socket`，`bind`，`accept`，`connect`

表头文件 `#include<sys/socket.h>`

定义函数 `int listen(int s,int backlog);`

函数说明 `listen()`用来等待参数`s`的`socket`连线。参数`backlog`指定同时能处理的最大连接要求，如果连接数目达此上限则客户端将收到ECONNREFUSED的错误。`listen()`并未开始接收连线，只是设置`socket`为`listen`模式，真正接收客户端连线的是`accept()`。通常`listen()`会在`socket()`，`bind()`之后调用，接着才调用`accept()`。

返回值 成功则返回0，失败返回-1，错误原因存于`errno`

附加说明 `listen()`只适用SOCK_STREAM或SOCK_SEQPACKET的`socket`类型。如果`socket`为AF_INET则参数`backlog`最大值可设至128。

错误代码 EBADF 参数`sockfd`非合法`socket`处理代码

EACCESS 权限不足

EOPNOTSUPP 指定的`socket`并未支援`listen`模式。

范例 `#include<sys/types.h>`

```

#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<unistd.h>
#define PORT 1234
#define MAXSOCKFD 10
main()
{
int sockfd,newsockfd,is_connected[MAXSOCKFD],fd;
struct sockaddr_in addr;
int addr_len = sizeof(struct sockaddr_in);
fd_set readfds;
char buffer[256];
char msg[ ] ="Welcome to server!";
if ((sockfd = socket(AF_INET,SOCK_STREAM,0))<0){
perror("socket");
exit(1);
}
bzero(&addr,sizeof(addr));
addr.sin_family =AF_INET;
addr.sin_port = htons(PORT);
addr.sin_addr.s_addr = htonl(INADDR_ANY);
if(bind(sockfd,&addr,sizeof(addr))<0){
perror("connect");
exit(1);
}
if(listen(sockfd,3)<0){
perror("listen");
exit(1);
}
for(fd=0;fd<MAXSOCKFD;fd++)
is_connected[fd]=0;
while(1){
FD_ZERO(&readfds);
FD_SET(sockfd,&readfds);
for(fd=0;fd<MAXSOCKFD;fd++)
if(is_connected[fd]) FD_SET(fd,&readfds);
if(!select(MAXSOCKFD,&readfds,NULL,NULL,NULL))continue;
for(fd=0;fd<MAXSOCKFD;fd++)
if(FD_ISSET(fd,&readfds)){
if(sockfd == fd){
if((newsockfd = accept (sockfd,&addr,&addr_len))<0)
perror("accept");
write(newsockfd,msg,sizeof(msg));
is_connected[newsockfd] =1;
printf("connect from %s\n",inet_ntoa(addr.sin_addr));
}else{
bzero(buffer,sizeof(buffer));
if(read(fd,buffer,sizeof(buffer))<=0){
printf("connect closed.\n");
is_connected[fd]=0;
close(fd);
}else
printf("%s",buffer);
}
}
}
}

```

```
}
```

执行 `$./listen`

```
connect from 127.0.0.1
```

```
hi I am client
```

```
connected closed.
```

`ntohl`（将32位网络字符顺序转换成主机字符顺序）

相关函数 `htonl`, `htons`, `ntohs`

表头文件 `#include<netinet/in.h>`

定义函数 `unsigned long int ntohl(unsigned long int netlong);`

函数说明 `ntohl()`用来将参数指定的32位`netlong`转换成主机字符顺序。

返回值 返回对应的主机字符顺序。

范例 参考`getservent()`。

`ntohs`（将16位网络字符顺序转换成主机字符顺序）

相关函数 `htonl`, `htons`, `ntohl`

表头文件 `#include<netinet/in.h>`

定义函数 `unsigned short int ntohs(unsigned short int netshort);`

函数说明 `ntohs()`用来将参数指定的16位`netshort`转换成主机字符顺序。

返回值 返回对应的主机顺序。

范例 参考`getservent()`。

`recv`（经`socket`接收数据）

相关函数 `recvfrom`, `recvmsg`, `send`, `sendto`, `socket`

表头文件 `#include<sys/types.h>`

`#include<sys/socket.h>`

定义函数 `int recv(int s,void *buf,int len,unsigned int flags);`

函数说明 `recv()`用来接收远端主机经指定的`socket`传来的数据，并把数据存到由参数`buf`指向的内存空间，参数`len`为可接收数据的最大长度。

参数 `flags`一般设0。其他数值定义如下：

`MSG_OOB` 接收以`out-of-band`送出的数据。

`MSG_PEEK` 返回来的数据并不会在系统内删除，如果再调用`recv()`会返回相同的数据内容。

`MSG_WAITALL`强迫接收到`len`大小的数据后才能返回，除非有错误或信号产生。

`MSG_NOSIGNAL`此操作不愿被`SIGPIPE`信号中断返回值成功则返回接收到的字符数，失败返回-1，错误原因存于`errno`中。

错误代码 `EBADF` 参数`s`非合法的`socket`处理代码

`EFAULT` 参数中有一指针指向无法存取的内存空间

`ENOTSOCK` 参数`s`为一文件描述词，非`socket`。

`EINTR` 被信号所中断

`EAGAIN` 此动作会令进程阻断，但参数`s`的`socket`为不可阻断

`ENOBUFS` 系统的缓冲内存不足。

`ENOMEM` 核心内存不足

`EINVAL` 传给系统调用的参数不正确。

范例 参考`listen()`。

recvfrom（经socket接收数据）

相关函数 recv, recvmsg, send, sendto, socket

表头文件 #include<sys/types.h>
#include<sys/socket.h>

定义函数 int recvfrom(int s,void *buf,int len,unsigned int flags ,struct sockaddr *from ,int *fromlen);

函数说明 recv()用来接收远程主机经指定的socket 传来的数据，并把数据存到由参数buf 指向的内存空间，参数len 为可接收数据的最大长度。参数flags 一般设0，其他数值定义请参考recv()。参数from用来指定欲传送的网络地址，结构sockaddr 请参考bind()。参数fromlen为sockaddr的结构长度。

返回值 成功则返回接收到的字符数，失败则返回-1，错误原因存于errno中。

错误代码 EBADF 参数s非合法的socket处理代码

EFAULT 参数中有一指针指向无法存取的内存空间。

ENOTSOCK 参数s为一文件描述词，非socket。

EINTR 被信号所中断。

EAGAIN 此动作会令进程阻断，但参数s的socket为不可阻断。

ENOBUFFS 系统的缓冲内存不足

ENOMEM 核心内存不足

EINVAL 传给系统调用的参数不正确。

范例 /*利用socket的UDP client

此程序会连线UDP server，并将键盘输入的字符串传给server。

UDP server 范例请参考sendto（）。

*/

#include<sys/stat.h>

#include<fcntl.h>

#include<unistd.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<arpa/inet.h>

#define PORT 2345

#define SERVER_IP "127.0.0.1"

main()

{

int s,len;

struct sockaddr_in addr;

int addr_len =sizeof(struct sockaddr_in);

char buffer[256];

/* 建立socket*/

if((s = socket(AF_INET,SOCK_DGRAM,0))<0){

perror("socket");

exit(1);

}

/* 填写sockaddr_in*/

bzero(&addr,sizeof(addr));

addr.sin_family = AF_INET;

addr.sin_port = htons(PORT);

addr.sin_addr.s_addr = inet_addr(SERVER_IP);

while(1){

bzero(buffer,sizeof(buffer));

/* 从标准输入设备取得字符串*/

len =read(STDIN_FILENO,buffer,sizeof(buffer));

/* 将字符串传送给server端*/

sendto(s,buffer,len,0,&addr,addr_len);

```

/* 接收server端返回的字符串*/
len = recvfrom(s,buffer,sizeof(buffer),0,&addr,&addr_len);
printf("receive: %s",buffer);
}
}

```

执行 (先执行udp server 再执行udp client)

```

hello /*从键盘输入字符串*/
receive: hello /*server端返回来的字符串*/

```

recvmsg (经socket接收数据)

相关函数 recv, recvfrom, send, sendto, sendmsg, socket

表头文件 #include<sys/types.h>
#include<sys/socket.h>

定义函数 int recvmsg(int s,struct msghdr *msg,unsigned int flags);

函数说明 recvmsg()用来接收远程主机经指定的socket传来的数据。参数s为已建立好连线的socket, 如果利用UDP协议则不需经过连线操作。参数msg指向欲连线的数据结构内容, 参数flags一般设0, 详细描述请参考send()。关于结构msghdr的定义请参考sendmsg()。

返回值 成功则返回接收到的字符数, 失败则返回-1, 错误原因存于errno中。

错误代码 EBADF 参数s非合法的socket处理代码。

EFAULT 参数中有一指针指向无法存取的内存空间

ENOTSOCK 参数s为一文件描述词, 非socket。

EINTR 被信号所中断。

EAGAIN 此操作会令进程阻断, 但参数s的socket为不可阻断。

ENOBUFFS 系统的缓冲内存不足

ENOMEM 核心内存不足

EINVAL 传给系统调用的参数不正确。

范例 参考recvfrom()。

send (经socket传送数据)

相关函数 sendto, sendmsg, recv, recvfrom, socket

表头文件 #include<sys/types.h>
#include<sys/socket.h>

定义函数 int send(int s,const void * msg,int len,unsigned int flags);

函数说明 send()用来将数据由指定的socket 传给对方主机。参数s为已建立好连接的socket。参数msg指向欲连线的数据内容, 参数len则为数据长度。参数flags一般设0, 其他数值定义如下

MSG_OOB 传送的数据以out-of-band 送出。

MSG_DONTROUTE 取消路由表查询

MSG_DONTWAIT 设置为不可阻断运作

MSG_NOSIGNAL 此动作不愿被SIGPIPE 信号中断。

返回值 成功则返回实际传送出去的字符数, 失败返回-1。错误原因存于errno

错误代码 EBADF 参数s 非合法的socket处理代码。

EFAULT 参数中有一指针指向无法存取的内存空间

ENOTSOCK 参数s为一文件描述词, 非socket。

EINTR 被信号所中断。

EAGAIN 此操作会令进程阻断, 但参数s的socket为不可阻断。

ENOBUFFS 系统的缓冲内存不足

ENOMEM 核心内存不足

EINVAL 传给系统调用的参数不正确。

范例 参考connect()

sendmsg (经socket传送数据)

相关函数 send, sendto, recv, recvfrom, recvmsg, socket

表头文件 #include <sys/types.h>
#include <sys/socket.h>

定义函数 int sendmsg(int s, const struct msghdr *msg, unsigned int flags);

函数说明 sendmsg()用来将数据由指定的socket传给对方主机。参数s为已建立好连线的socket, 如果利用UDP协议则不需经过连线操作。参数msg 指向欲连线的数据结构内容, 参数flags一般默认为0, 详细描述请参考send()。

结构msghdr定义如下

```
struct msghdr
{
    void *msg_name; /*Address to send to /receive from . */
    socklen_t msg_namelen; /* Length of address data */
    struct iovec * msg_iov; /* Vector of data to send/receive into */
    size_t msg_iovlen; /* Number of elements in the vector */
    void * msg_control; /* Ancillary data */
    size_t msg_controllen; /* Ancillary data buffer length */
    int msg_flags; /* Flags on received message */
};
```

返回值 成功则返回实际传送出去的字符数, 失败返回-1, 错误原因存于errno

错误代码 EBADF 参数s 非合法的socket处理代码。

EFAULT 参数中有一指针指向无法存取的内存空间

ENOTSOCK 参数s为一文件描述词, 非socket。

EINTR 被信号所中断。

EAGAIN 此操作会令进程阻断, 但参数s的socket为不可阻断。

ENOBUFS 系统的缓冲内存不足

ENOMEM 核心内存不足

EINVAL 传给系统调用的参数不正确。

范例 参考sendto()。

sendto (经socket传送数据)

相关函数 send, sendmsg, recv, recvfrom, socket

表头文件 #include < sys/types.h >
#include < sys/socket.h >

定义函数 int sendto (int s , const void * msg, int len, unsigned int flags, const struct sockaddr * to , int tolen);

函数说明 sendto() 用来将数据由指定的socket传给对方主机。参数s为已建好连线的socket,如果利用UDP协议则不需经过连线操作。参数msg指向欲连线的数据内容, 参数flags 一般设0, 详细描述请参考send()。参数to用来指定欲传送的网络地址, 结构sockaddr请参考bind()。参数tolen为sockaddr的结果长度。

返回值 成功则返回实际传送出去的字符数, 失败返回-1, 错误原因存于errno 中。

错误代码 EBADF 参数s非合法的socket处理代码。

EFAULT 参数中有一指针指向无法存取的内存空间。

WNOTSOCK 参数s为一文件描述词, 非socket。

EINTR 被信号所中断。

EAGAIN 此动作会令进程阻断, 但参数s的socket为可阻断的。

ENOBUFS 系统的缓冲内存不足。

EINVAL 传给系统调用的参数不正确。

范例

```
#include < sys/types.h >
#include < sys/socket.h >
# include <netinet.in.h>
#include <arpa/inet.h>
#define PORT 2345 /*使用的port*/
main(){
int sockfd,len;
struct sockaddr_in addr;
char buffer[256];
/*建立socket*/
if(sockfd=socket (AF_INET,SOCK_DGRAM,0)<0){
perror ("socket");
exit(1);
}
/*填写sockaddr_in 结构*/
bzero ( &addr, sizeof(addr) );
addr.sin_family=AF_INET;
addr.sin_port=htons(PORT);
addr.sin_addr=hton1(INADDR_ANY) ;
if (bind(sockfd, &addr, sizeof(addr))<0){
perror("connect");
exit(1);
}
while(1){
bezro(buffer,sizeof(buffer));
len = recvfrom(socket,buffer,sizeof(buffer), 0 , &addr &addr_len);
/*显示client端的网络地址*/
printf("receive from %s\n " , inet_ntoa( addr.sin_addr));
/*将字符串返回给client端*/
sendto(sockfd,buffer,len,0,&addr,addr_len);"
}
}
```

执行 请参考recvfrom()

setprotoent（打开网络协议的数据文件）

相关函数 getprotobyname, getprotobynumber, endprotoent

表头文件 #include <netdb.h>

定义函数 void setprotoent (int stayopen);

函数说明 setprotoent()用来打开/etc/protocols， 如果参数stayopen值为1，则接下来的getprotobyname()或getprotobynumber()将不会自动关闭此文件。

setservent（打开主机网络服务的数据文件）

相关函数 getservent, getservbyname, getservbyport, endservent

表头文件 #include < netdb.h >

定义函数 void setservent (int stayopen);

函数说明 setservent()用来打开/etc/services， 如果参数stayopen值为1，则接下来的getservbyname()或getservbyport()将补回自动关闭文件。

setsockopt（设置socket状态）

相关函数 getsockopt

表头文件 #include<sys/types.h>
#include<sys/socket.h>

定义函数 int setsockopt(int s,int level,int optname,const void * optval,,socklen_toptlen);

函数说明 setsockopt()用来设置参数s所指定的socket状态。参数level代表欲设置的网络层，一般设成 SOL_SOCKET以存取socket层。参数optname代表欲设置的选项，有下列几种数值：
SO_DEBUG 打开或关闭排错模式
SO_REUSEADDR 允许在bind（）过程中本地地址可重复使用
SO_TYPE 返回socket形态。
SO_ERROR 返回socket已发生的错误原因
SO_DONTROUTE 送出的数据包不要利用路由设备来传输。
SO_BROADCAST 使用广播方式传送
SO_SNDBUF 设置送出的暂存区大小
SO_RCVBUF 设置接收的暂存区大小
SO_KEEPALIVE 定期确定连线是否已终止。
SO_OOBINLINE 当接收到OOB数据时会马上送至标准输入设备
SO_LINGER 确保数据安全且可靠的传送出去。

参数 optval代表欲设置的值，参数optlen则为optval的长度。

返回值 成功则返回0，若有错误则返回-1，错误原因存于errno。

附加说明 EBADF 参数s并非合法的socket处理代码
ENOTSOCK 参数s为一文件描述词，非socket
ENOPROTOOPT 参数optname指定的选项不正确。
EFAULT 参数optval指针指向无法存取的内存空间。

范例 参考getsockopt()。

shutdown（终止socket通信）

相关函数 socket, connect

表头文件 #include<sys/socket.h>

定义函数 int shutdown(int s,int how);

函数说明 shutdown()用来终止参数s所指定的socket连线。参数s是连线中的socket处理代码，参数how有下列几种情况：
how=0 终止读取操作。
how=1 终止传送操作
how=2 终止读取及传送操作

返回值 成功则返回0，失败返回-1，错误原因存于errno。

错误代码 EBADF 参数s不是有效的socket处理代码
ENOTSOCK 参数s为一文件描述词，非socket
ENOTCONN 参数s指定的socket并未连线

socket（建立一个socket通信）

相关函数 accept, bind, connect, listen

表头文件 #include<sys/types.h>
#include<sys/socket.h>

定义函数 int socket(int domain,int type,int protocol);

函数说明 `socket()`用来建立一个新的socket，也就是向系统注册，通知系统建立一通信端口。参数domain 指定使用何种的地址类型，完整的定义在/usr/include/bits/socket.h 内，底下是常见的协议:

PF_UNIX/PF_LOCAL/AF_UNIX/AF_LOCAL UNIX 进程通信协议

PF_INET/AF_INET Ipv4网络协议

PF_INET6/AF_INET6 Ipv6 网络协议

PF_IPX/AF_IPX IPX-Novell协议

PF_NETLINK/AF_NETLINK 核心用户接口装置

PF_X25/AF_X25 ITU-T X.25/ISO-8208 协议

PF_AX25/AF_AX25 业余无线AX.25协议

PF_ATMPVC/AF_ATMPVC 存取原始ATM PVCs

PF_APPLETALK/AF_APPLETALK appletalk (DDP) 协议

PF_PACKET/AF_PACKET 初级封包接口

参数 type有下列几种数值:

SOCK_STREAM 提供双向连续且可信赖的数据流，即TCP。支持

OOB 机制，在所有数据传送前必须使用connect()来建立连线状态。

SOCK_DGRAM 使用不连续不可信赖的数据包连接

SOCK_SEQPACKET 提供连续可信赖的数据包连接

SOCK_RAW 提供原始网络协议存取

SOCK_RDM 提供可信赖的数据包连接

SOCK_PACKET 提供和网络驱动程序直接通信。

protocol用来指定socket所使用的传输协议编号，通常此参考不用管它，设为0即可。

返回值 成功则返回socket处理代码，失败返回-1。

错误代码 EPROTONOSUPPORT 参数domain指定的类型不支持参数type或protocol指定的协议

ENFILE 核心内存不足，无法建立新的socket结构

EMFILE 进程文件表溢出，无法再建立新的socket

EACCESS 权限不足，无法建立type或protocol指定的协议

ENOBUFS/ENOMEM 内存不足

EINVAL 参数domain/type/protocol不合法

范例 参考connect()。

getenv（取得环境变量内容）

相关函数 putenv, setenv, unsetenv

表头文件 #include<stdlib.h>

定义函数 char * getenv(const char *name);

函数说明 getenv()用来取得参数name环境变量的内容。参数name为环境变量的名称，如果该变量存在则会返回指向该内容的指针。环境变量的格式为name=value。

返回值 执行成功则返回指向该内容的指针，找不到符合的环境变量名称则返回NULL。

```
范例 #include<stdlib.h>
      main()
      {
        char *p;
        if((p = getenv("USER")))
          printf("USER=%s\n",p);
      }
```

执行 USER = root

putenv（改变或增加环境变量）

相关函数 getenv, setenv, unsetenv

表头文件 #include<stdlib.h>

定义函数 int putenv(const char * string);

函数说明 putenv()用来改变或增加环境变量的内容。参数string的格式为name=value，如果该环境变量原先存在，则变量内容会依参数string改变，否则此参数内容会成为新的环境变量。

返回值 执行成功则返回0，有错误发生则返回-1。

错误代码 ENOMEM 内存不足，无法配置新的环境变量空间。

```
范例 #include<stdlib.h>
      main()
      {
        char *p;
        if((p = getenv("USER")))
          printf("USER =%s\n",p);
        putenv("USER=test");
        printf("USER+5s\n",getenv("USER"));
      }
```

执行 USER=root

USER=root

setenv（改变或增加环境变量）

相关函数 getenv, putenv, unsetenv

表头文件 #include<stdlib.h>

定义函数 int setenv(const char *name,const char * value,int overwrite);

函数说明 setenv()用来改变或增加环境变量的内容。参数name为环境变量名称字符串。

参数 **value** 则为变量内容，参数 **overwrite** 用来决定是否要改变已存在的环境变量。如果 **overwrite** 不为0，而该环境变量原已有内容，则原内容会被改为参数 **value** 所指的变量内容。如果 **overwrite** 为0，且该环境变量已有内容，则参数 **value** 会被忽略。

返回值 执行成功则返回0，有错误发生时返回-1。

错误代码 **ENOMEM** 内存不足，无法配置新的环境变量空间

范例 `#include <stdlib.h>`

```
main()
{
    char * p;
    if((p=getenv("USER")))
        printf("USER = %s\n",p);
    setenv("USER","test",1);
    printf("USER=%s\n",getenv("USER"));
    unsetenv("USER");
    printf("USER=%s\n",getenv("USER"));
}
```

执行 `USER = root`

`USER = test`

`USER = (null)`

getopt (分析命令行参数)

相关函数

表头文件 `#include <unistd.h>`

定义函数 `int getopt(int argc, char * const argv[], const char * optstring);`

函数说明 `getopt()`用来分析命令行参数。参数`argc`和`argv`是由`main()`传递的参数个数和内容。参数`optstring` 则代表欲处理的选项字符串。此函数会返回在`argv` 中下一个的选项字母，此字母会对应参数`optstring` 中的字母。如果选项字符串里的字母后接着冒号“:”，则表示还有相关的参数，全域变量`optarg` 即会指向此额外参数。如果`getopt()`找不到符合的参数则会印出错信息，并将全域变量`optopt`设为“?”字符，如果不希望`getopt()`印出错信息，则只要将全域变量`opterr`设为0即可。

返回值 如果找到符合的参数则返回此参数字母，如果参数不包含在参数`optstring` 的选项字母则返回“?”字符，分析结束则返回-1。

范例 `#include <stdio.h>`
`#include <unistd.h>`
`int main(int argc, char **argv)`
`{`
`int ch;`
`opterr = 0;`
`while((ch = getopt(argc, argv, "a:bcde")) != -1)`
`switch(ch)`
`{`
`case 'a':`
`printf("option a: %s\n", optarg);`
`break;`
`case 'b':`
`printf("option b: %b\n");`
`break;`
`default:`
`printf("other option: %c\n", ch);`
`}`
`printf("optopt + %c\n", optopt);`
`}`

执行 `./getopt -b`
`option b:b`
`./getopt -c`
`other option:c`
`./getopt -a`
`other option :?`
`./getopt -a12345`
`option a:'12345'`

isatty (判断文件描述词是否是终端机)

相关函数 `ttyname`

表头文件 `#include <unistd.h>`

定义函数 `int isatty(int desc);`

函数说明 如果参数`desc`所代表的文件描述词为一终端机则返回1，否则返回0。

返回值 如果文件为终端机则返回1，否则返回0。

范例 参考ttyname()。

select (I/O多工机制)

表头文件 #include<sys/time.h>
#include<sys/types.h>
#include<unistd.h>

定义函数 int select(int n,fd_set * readfds,fd_set * writefds,fd_set * exceptfds,struct timeval * timeout);

函数说明 select()用来等待文件描述词状态的改变。参数n代表最大的文件描述词加1，参数readfds、writefds和exceptfds 称为描述词组，是用来回传该描述词的读，写或例外的状况。底下的宏提供了处理这三种描述词组的方式:

FD_CLR(inr fd,fd_set* set); 用来清除描述词组set中相关fd 的位

FD_ISSET(int fd,fd_set *set); 用来测试描述词组set中相关fd 的位是否为真

FD_SET (int fd,fd_set*set); 用来设置描述词组set中相关fd的位

FD_ZERO (fd_set *set); 用来清除描述词组set的全部位

参数 timeout为结构timeval，用来设置select()的等待时间，其结构定义如下

```
struct timeval
{
    time_t tv_sec;
    time_t tv_usec;
};
```

返回值 如果参数timeout设为NULL则表示select () 没有timeout。

错误代码 执行成功则返回文件描述词状态已改变的个数，如果返回0代表在描述词状态改变前已超过timeout时间，当有错误发生时则返回-1，错误原因存于errno，此时参数readfds，writefds，exceptfds和timeout的值变成不可预测。

EBADF 文件描述词为无效的或该文件已关闭

EINTR 此调用被信号所中断

EINVAL 参数n 为负值。

ENOMEM 核心内存不足

范例 常见的程序片段:fs_set readset;

FD_ZERO(&readset);

FD_SET(fd,&readset);

select(fd+1,&readset,NULL,NULL,NULL);

if(FD_ISSET(fd,readset){.....}

ttyname (返回一终端机名称)

相关函数 Isatty

表头文件 #include<unistd.h>

定义函数 char * ttyname(int desc);

函数说明 如果参数desc所代表的文件描述词为一终端机，则会将此终端机名称由一字符串指针返回，否则返回NULL。

返回值 如果成功则返回指向终端机名称的字符串指针，有错误情况发生时则返回NULL。

范例 #include<unistd.h>

#include<sys/types.h>

#include <sys/stat.h>

#include<fcntl.h>

main()

{

int fd;

```
char * file = "/dev/tty";  
fd = open (file,O_RDONLY);  
printf("%s",file);  
if(isatty(fd)){  
printf("is a tty.\n");  
printf("ttyname = %s \n",ttyname(fd));  
}  
else printf(" is not a tty\n");  
close(fd);  
}
```

执行 /dev/tty is a tty
ttyname = /dev/tty