目　录

第 5 章 字符串处理函数

第 6 章 数学计算函数

第 7 章 用户和组函数

字符测试函数

isalnum（测试字符是否为英文字母或数字）

相关函数：isalpha，isdigit，islower，isupper

表头文件：#include <ctype.h>

定义函数：int isalnum（int c）;

函数说明：检查参数 c 是否为英文字母或阿拉伯数字，在标准 C 中相当于使用（isalpha（c）|| isdigit（c））做测试。

返 回 值：若参数 c 为字母或数字，则返回 TRUE，否则返回 NULL（0）。

附加说明：此为宏定义，非真正函数。

范 例

```
/* 找出 str 字符串中为英文字母或数字的字符 */
#include <ctype.h>
main ()
{
  char str[]="123c@#FDsP[e?";
  int i;
  for (i = 0; str[i] != 0;i++)
    if (isalnum (str[i]) ) printf ("%c is an alphanumeric character\n", str[i]) ;
}
```

执行结果

```
1 is an alphabetic character
2 is an alphabetic character
3 is an alphabetic character
c is an alphabetic character
F is an alphabetic character
D is an alphabetic character
s is an alphabetic character
P is an alphabetic character
e is an alphabetic character
```

(1)

isalpha（测试字符是否为英文字母）

相关函数：isalnum，islower，isupper

表头文件：#include <ctype.h>

定义函数：int isalpha（int c）;

函数说明：检查参数 c 是否为英文字母，在标准 C 中相当于使用（isupper（c）‖islower（c））做测试。

返 回 值：若参数 c 为英文字母，则返回 TRUE，否则返回 NULL（0）。

附加说明：此为宏定义，非真正函数。

范 例

```
/* 找出 str 字符串中为英文字母的字符 */
#include <ctype.h>
main ()
{
  char str[]="123c@#FDsP[e?";
  int i;
  for (i = 0; str[i] != 0;i++)
  if (isalpha (str[i]) ) printf ("%c is an alphabetic character\n", str[i]) ;
}
```

执行结果

```
c is an alphabetic character
F is an alphabetic character
D is an alphabetic character
s is an alphabetic character
P is an alphabetic character
e is an alphabetic character
```

isascii（测试字符是否为 ASCII 码字符）

相关函数：iscntrl

表头文件：#include <ctype.h>

定义函数：int isascii（int c）;

函数说明：检查参数 c 是否为 ASCII 码字符，也就是判断 c 的范围是否在 0 到 127 之间。

返 回 值：若参数 c 为 ASCII 码字符，则返回 TRUE，否则返回 NULL（0）。

附加说明：此为宏定义，非真正函数。

范 例

```
/* 判断 int i 是否具有对映的 ASCII 码字符 */
#include <ctype.h>
main ()
{
   int i;
   for (i = 125; i < 130; i++)
    if (isascii (i) )
     printf ("%d is an ascii character : %c\n", i, i) ;
    else
     printf ("%d is not an ascii character\n", i) ;
}
```

执行结果

```
125 is an ascii character : }
126 is an ascii character : ~
127 is an ascii character :
128 is not an ascii character
129 is not an ascii character
```

isblank（测试字符是否为空格字符）

相关函数：isspace

表头文件：#include <ctype.h>

定义函数：int isblank（int c）;

函数说明：检查参数 c 是否为空格字符，也就是判断是否为空格（space）或是定位字符（tab）。空格（space）的 ASCII 码为 32，定位字符（tab）的 ASCII 码则为 9。

返 回 值：若参数 c 为空格字符，则返回 TRUE，否则返回 NULL（0）。

附加说明：此为宏定义，非真正函数。

范 例

```
/* 将字符串 str[] 中内含的空格字符找出，并显示空格字符的 ASCII 码*/
#include <ctype.h>
main ()
{
  char str[]="123c @# FD       sP[e?*;
  int i;
  for (i = 0; str[i] != 0;i++)
     if (isblank (str[i]) ) printf ("str[%d] is blank character: %d\n", i, str[i]) ;
}
```

执行结果

```
str[4] is blank character: 32
str[7] is blank character: 32
str[10] is blank character: 9
```

iscntrl（测试字符是否为 ASCII 码的控制字符）

相关函数：isascii

表头文件：#include <ctype.h>

定义函数：int iscntrl（int c）;

函数说明：检查参数 c 是否为 ASCII 控制码，也就是判断 c 的范围是否在 0 到 31 之间。

返 回 值：若参数 c 为 ASCII 控制码，则返回 TRUE，否则返回 NULL（0）。

附加说明：此为宏定义，非真正函数。

isdigit（测试字符是否为阿拉伯数字）

相关函数：isxdigit

表头文件：#include <ctype.h>

定义函数：int isdigit（int c）;

函数说明：检查参数 c 是否为阿拉伯数字 0 到 9。

返 回 值：若参数 c 为阿拉伯数字，则返回 TRUE，否则返回 NULL（0）。

附加说明：此为宏定义，非真正函数。

范 例

```
/* 找出 str 字符串中为阿拉伯数字的字符 */
#include <ctype.h>
main ()
{
 char str[]="123c@#FDsP[e?";
 int i;
 for (i = 0; str[i] != 0;i++)
    if (isdigit (str[i]) ) printf ("%c is an digit character\n", str[i]) ;
}
```

执行结果

```
1 is an digit character
```

```
2 is an digit character
3 is an digit character
```

isgraph（测试字符是否为可打印字符）

相关函数：isprint

表头文件：#include <ctype.h>

定义函数：int isgraph（int c）;

函数说明：检查参数 c 是否为可打印字符，若 c 所对映的 ASCII 码可打印，且非空格字符则返回 TRUE。

返 回 值：若参数 c 为可打印字符，则返回 TRUE，否则返回 NULL（0）。

附加说明：此为宏定义，非真正函数。

范 例

```
/* 判断 str 字符串中那些为可打印字符 */
#include <ctype.h>
main ()
{
  char str[]="a5 @;";
  int i;
  for (i = 0; str[i] != 0;i++)
    if (isgraph (str[i]) )
      printf ("str[%d] is printable character: %d\n", i, str[i]) ;
}
```

执行结果

```
str[0] is printable character: a
str[1] is printable character: 5
str[3] is printable character: @
str[4] is printable character: ;
```

islower（测试字符是否为小写英文字母）

相关函数：isalpha，isupper

表头文件：#include <ctype.h>

定义函数：int islower（int c）;

函数说明：检查参数 c 是否为小写英文字母。

返 回 值：若参数 c 为小写英文字母，则返回 TRUE，否则返回 NULL（0）。

附加说明：此为宏定义，非真正函数。

范 例

```
/* 找出 str 字符串中为小写英文字母的字符 */
#include <ctype.h>
main ()
{
  char str[]="123c@#FDsP[e?";
  int i;
  for (i = 0; str[i] != 0;i++)
    if (islower (str[i]) ) printf ("%c is a lower-case character\n", str[i]) ;
}
```

执行结果

```
c is a lower-case character
s is a lower-case character
e is a lower-case character
```

isprint（测试字符是否为可打印字符）

相关函数：isgraph

表头文件：#include <ctype.h>

定义函数：int isprint（int c）;

函数说明：检查参数 c 是否为可打印字符，若 c 所对映的 ASCII 码可打印，其中包含空格字符，则返回 TRUE。

返 回 值：若参数 c 为可打印字符，则返回 TRUE，否则返回 NULL（0）。

附加说明：此为宏定义，非真正函数。

范 例

```
/* 判断 str 字符串中那些为可打印字符（包含空格字符）*/
#include <ctype.h>
main ()
{
 char str[]="a5 @;";
 int i;
 for (i = 0; str[i] != 0;i++)
  if (isprint (str[i]) )
     printf ("str[%d] is printable character: %d\n", i, str[i]) ;
}
```

（可与 isgraph 范例的执行结果参照）

```
str[0] is printable character: a
str[1] is printable character: 5
str[2] is printable character:
str[3] is printable character: @
str[4] is printable character: ;
```

isspace（测试字符是否为空格字符）

相关函数：isgraph

表头文件：#include <ctype.h>

定义函数：isspace

函数说明：检查参数 c 是否为空格字符，也就是判断是否为空格（' '）、定位字符（'\t'）、CR（'\r'）、换行（'\n'）、垂直定位字符（'\v'）或翻页（'\f'）的情况。

返 回 值：若参数 c 为空格字符，则返回 TRUE，否则返回 NULL（0）。

附加说明：此为宏定义，非真正函数。

范 例

```
/* 将字符串 str[] 中内含的空格字符找出，并显示空格字符的 ASCII 码*/
#include <ctype.h>
main ()
{
 char str[]="123c @# FD\tsP[e?\n";
 int i;
 for (i = 0; str[i] != 0;i++)
  if (isspace (str[i]) )
   printf ("str[%d] is a white-space character: %d\n", i, str[i]) ;
}
```

执行结果

```
str[4] is a white-space character: 32
str[7] is a white-space character: 32
str[10] is a white-space character: 9   /* \t */
str[16] is a white-space character: 10/* \n */
```

ispunct（测试字符是否为标点符号或特殊符号）

相关函数：isspace，isdigit，isalpha

表头文件：#include <ctype.h>

定义函数：int ispunct（int c）;

函数说明：检查参数 c 是否为标点符号或特殊符号。返回 TRUE 也就是代表参数 c 为

非空格、非数字和非英文字母。

返 回 值：若参数 c 为标点符号或特殊符号，则返回 TRUE，否则返回 NULL（0）。

附加说明：此为宏定义，非真正函数。

范 例

```
/* 列出字符串 str 中的标点符号或特殊符号 */
#include <ctype.h>
main ()
{
  char str[]="123c@ #FDsP[e?";
  int i;
  for (i = 0; str[i] != 0;i++)
    if (ispunct (str[i]) ) printf ("%c\n", str[i]) ;
}
```

执行结果

```
@#[?
```

isupper（测试字符是否为大写英文字母）

相关函数：isalpha，islower

表头文件：#include <ctype.h>

定义函数：int isupper（int c）;

函数说明：检查参数 c 是否为大写英文字母。

返 回 值：若参数 c 为大写英文字母，则返回 TRUE，否则返回 NULL（0）。

附加说明：此为宏定义，非真正函数。

范 例

```
/* 找出 str 字符串中为大写英文字母的字符 */
```

```
#include <ctype.h>
main ()
{
 char str[]="123c@#FDsP[e?";
 int i;
 for (i = 0; str[i] != 0;i++)
  if (isupper (str[i]) ) printf ("%c is a an uppercase character\n", str[i]) ;
}
```

（可参照 islower 范例的执行结果）

```
F is a an uppercase character
D is a an uppercase character
P is a an uppercase character
```

isxdigit（测试字符是否为 16 进制数字）

相关函数：isalnum，isdigit

表头文件：#include <ctype.h>

定义函数：int isxdigit（int c）;

函数说明：检查参数 c 是否为 16 进制数字，只要 c 为下列其中一个情况则返回 TRUE。

16 进制数字：0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

返 回 值：若参数 c 为 16 进制数字，则返回 TRUE，否则返回 NULL（0）。

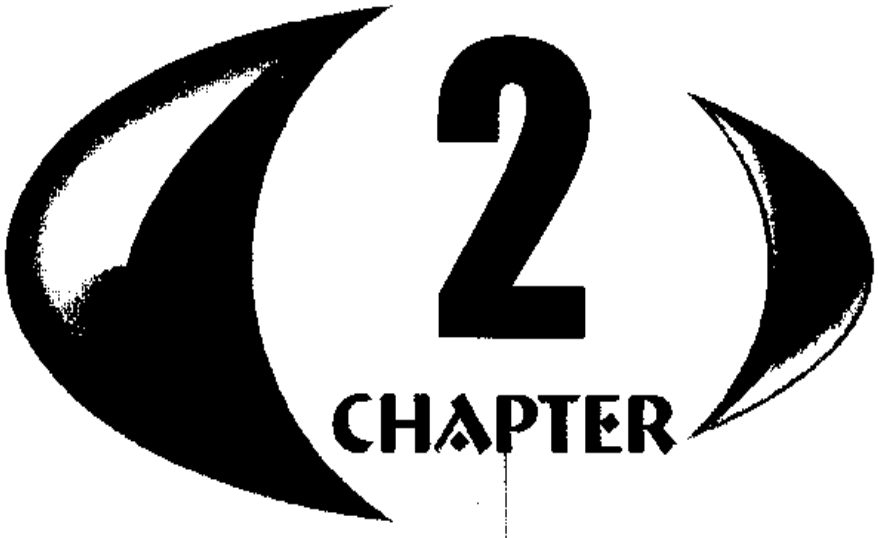
附加说明：此为宏定义，非真正函数。

范 例

```
/* 找出 str 字符串中为十六进制数字的字符 */
#include <ctype.h>
main ()
```

```
{
  char str[]="123c@#FDsP[e?";
  int i;
  for (i = 0; str[i] != 0;i++)
    if (isxdigit (str[i]) ) printf ("%c is a hexadecimal digits\n", str[i]) ;
}
```


```
1 is a hexadecimal digits
2 is a hexadecimal digits
3 is a hexadecimal digits
c is a hexadecimal digits
F is a hexadecimal digits
D is a hexadecimal digits
e is a hexadecimal digits
```



数据转换函数

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) ;
}
```

执行结果

```
c = -98.23
```

atoi（将字符串转换成整型数）

相关函数：atof，atol，strtod，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>
main ()
{
  char a[]="-100";
  char b[]=" 456";
  int c;
  c = atoi (a) + atoi (b) ;
  printf ("c = %d\n", c) ;
}
```

执行结果

```
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) ;
}
```

执行结果

```
c = 1234567890
```

ecvt（将浮点型数转换成字符串，取四舍五入）

相关函数：fcvt，gcvt，sprintf

表头文件：#include <stdlib.h>

定义函数：char *ecvt（double number，int ndigits，int *decpt，int *sign）；

函数说明：ecvt（）用来将参数 number 转换成 ASCII 码字符串，参数 ndigits 表示显示的位数。若转换成功，参数 decpt 指针所指的变量会返回数值中小数点的地址（从左至右算起），而参数 sign 指针所指的变量则代表数值正或负，若

数值为正，该返回值则为0，否则为1。

返 回 值：返回一字符串指针，此字符串声明为static，若再调用 ecvt（）或fcvt（），此字符串内容会被覆盖。

附加说明：请尽量改用 sprintf（）做转换。

范 例

```
#include <stdlib.h>
main ()
{
 double a =  123.45;
 double b = -1234.56;
 char *ptr;
 int decpt, sign;
 ptr = ecvt (a, 5, &decpt, &sign) ;
 printf ("decept = %d, sign = %d, a value = %s\n", decpt, sign, ptr) ;
 ptr = ecvt (b, 6, &decpt, &sign) ;
 printf ("decept = %d, sign = %d, b value = %s\n", decpt, sign, ptr) ;
}
```

执行结果

```
decept = 3, sign = 0, a value = 12345
decept = 4, sign = 1, b value = 123456
```

fcvt（将浮点型数转换为字符串，取四舍五入）

相关函数：ecvt，gcvt，sprintf

表头文件：#include <stdlib.h>

定义函数：char *fcvt（double number，int ndigits，int *decpt，int *sign）;

函数说明：fcvt（）用来将参数 number 转换成 ASCII 码字符串，参数 ndigits 表示小数点后显示的位数。若转换成功，参数 decpt 指针所指的变量会返回数值中

小数点的地址（从左至右算起），而参数 sign 指针所指的变量则代表数值正或负，若数值为正，该传回值则为 0，否则为 1。

返 回 值：返回一字符串指针，此字符串声明为 static，若再调用 ecvt（）或 fcvt（）此字符串内容会被覆盖。

附加说明：请尽量改用 sprintf（）做转换。

范 例

```
#include <stdlib.h>
main ()
{
  double a =  123.45;
  double b = -1234.567;
  char *ptr;
  int decpt, sign;
  ptr = fcvt (a, 2, &decpt, &sign) ; /* 小数点后显示 2 位数 */
  printf ("decept = %d, sign = %d, a value = %s\n", decpt, sign, ptr) ;
  ptr = fcvt (b, 3, &decpt, &sign) ; /* 小数点后显示 3 位数 */
  printf ("decept = %d, sign = %d, b value = %s\n", decpt, sign, ptr) ;
}
```

执行结果

```
decept = 3, sign = 0, a value = 12345
decept = 4, sign = 1, b value = 1234567
```

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。

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) ;
}
```

执行结果

```
a value = 123.45
b value = -1234.56
```

strtod（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 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，直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：请参考 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[]="1000000000";
  char b[]="1000000000";
  char c[]=" ffff";
  printf ("a = %d\n", strtol (a, NULL, 0) ) ;
  printf ("b = %d\n", strtol (b, NULL, 2) ) ;
  printf ("c = %d\n", strtol (c, NULL, 16) ) ;
}
```

执行结果

```
a = 1000000000
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[]="1000000000";
  char b[]="1000000000";
  char c[]=" ffff";
  printf ("a = %d\n", strtol (a, NULL, 0) ) ;
  printf ("b = %d\n", strtol (b, NULL, 2) ) ;
  printf ("c = %d\n", strtol (c, NULL, 16) ) ;
}
```

执行结果

```
a = 1000000000
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型整数。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 strtoul（）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

请参考 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) ;
}
```


```
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 <ctype.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。

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) ;
```

```
}
```

```
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）;

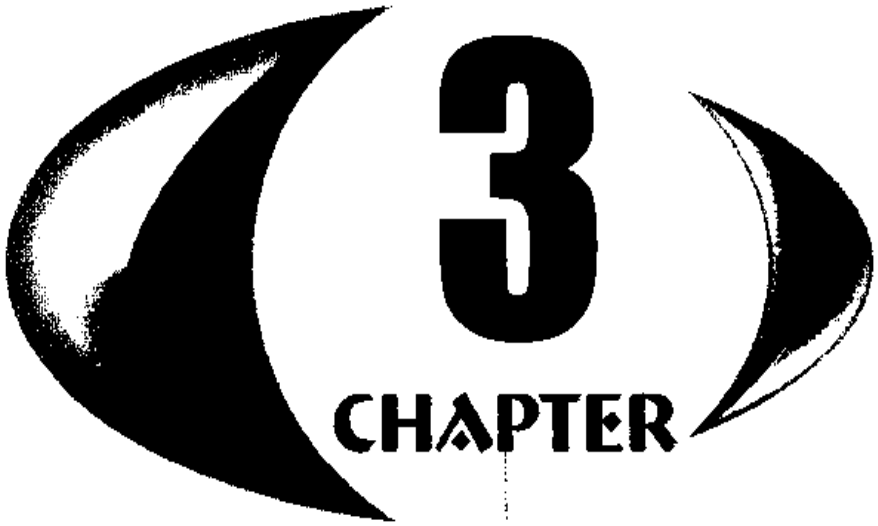
函数说明：若参数 c 为小写字母则将该对映的大写字母返回。

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。

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) ;
}
```

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



内存配置函数

alloca（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 *alloca（size_t size）;

函数说明：alloca（）用来配置 size 个字节的内存空间，然而和 malloc/calloc 不同的是，alloca（）是从堆栈空间（stack）中配置内存，因此在函数返回时会自动释放此空间。

返 回 值：若配置成功则返回一指针，失败则返回 NULL。

brk（改变数据字节的范围）

相关函数：malloc，calloc，free，realloc，sbrk

表头文件：#include <stdlib.h>

定义函数：int brk（void *end_data_segment）;

函数说明：brk（）用来依参数 end_data_segment 所指的数值设成新的数据字节范围。

返 回 值：若函数调用成功则返回 0。函数调用失败则返回 -1，将 errno 设为 ENOMEM。

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 4096 bytes。

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 offset）；

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 flag 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，不对地址作修正。通常不鼓励使用此旗标。
MAP_SHARED	对应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	对应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，而且映射区无法和其他进程共享。
MAP_DENYWRITE	只允许对应射区域的写入操作，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（）时必须要指定 MAP_SHARED 或 AP_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 */
    close (fd) ;                                    /* 关闭文件 */
}
```

执行结果

```
root: x: 0: 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: /:
xfs: 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并不会解除映射。

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

请参考 mmap（）。

realloc（更改已配置的内存空间）

相关函数：malloc，calloc，free，brk

表头文件：#include <stdlib.h>

定义函数：void *realloc（void *ptr，size_t size）;

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，而参数 size 为新配置的内存大小，其值可比原内存大或小。若参数 size 值较原配置空间较小，内存内容并不会改变，且返回的指针为原来内存启始地址;但若 size 值较原配置空间大，则 realloc（）不一定会返回原先的指针，原先的内容虽不会改变，但新多出的内存则未设初值。若是参数 ptr 指针为 NULL，此调用相当于 malloc（size）;若参数 size 值为 0，此调用相当于 free（ptr）;

返 回 值：若配置成功则返回一指针，失败则返回 NULL。

范 例

```
#include <stdlib.h>
main ()
{
 char *ptr1 = malloc (10) ;
 char *ptr2;
```

```
  memset (ptr1, 'A', 10) ;
  * (ptr1+10) = '\0';
  printf ("before realloc : ptr = %x [%s]\n", ptr1, ptr1) ;
  ptr2 = realloc (ptr1, 40960) ;
  printf ("after  realloc : ptr = %x [%s]\n", ptr2, ptr2) ;
}
```

执行结果

```
before realloc : ptr = 8049840 [AAAAAAAAAA]
after  realloc : ptr = 8049840 [AAAAAAAAAA]
```

sbrk（增加程序可用的数据空间）

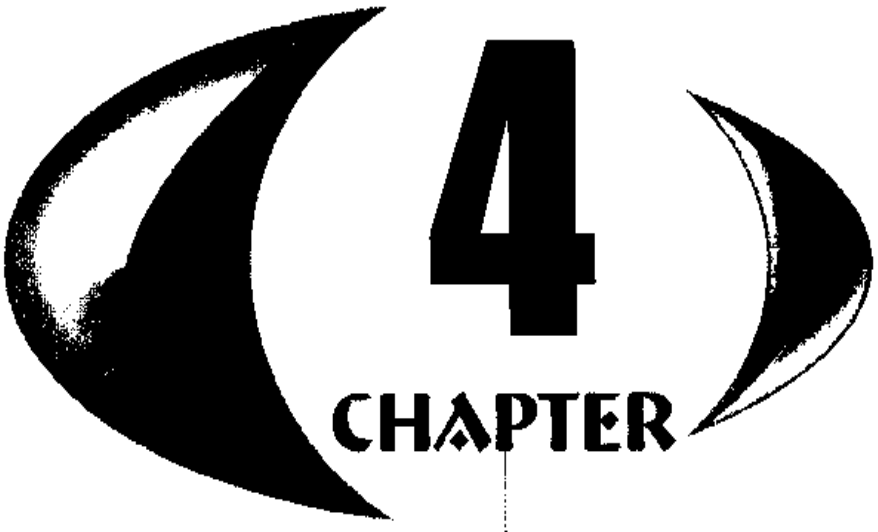
相关函数：malloc，calloc，free，realloc，brk

表头文件：#include <stdlib.h>

定义函数：void *sbrk（ptrdiff_t increment）;

函数说明：sbrk（）用来增加程序可用的数据空间，增加大小由参数 incre ment 决定。

返 回 值：若函数调用成功则返回一指针，指向新的内存空间。函数调用失败则返回 -1，将 errno 设为 ENOMEM。



时间函数

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) ) ) ;
}
```

执行结果

```
Sat Oct 28 02: 10: 06 2000
```

clock（取得进程占用 CPU 的大约时间）

相关函数：time

表头文件：#include <time.h>

定义函数：clock_t clock（void）

函数说明：clock（）用来取得进程所占用 CPU 的大约时间。

返 回 值：返回进程所占用 CPU 的大约时间。

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) ) ;
}
```

执行结果

```
Sat Oct 28 10：12：05 2000
```

difftime（计算时间差距）

相关函数：time，ctime，gmtime，localtime

表头文件：#include <time.h>

定义函数：double difftime（time_t time1，time_t time0）；

函数说明：difftime（）用来计算参数 time1 和 time0 所代表的时间差距，结果以 double 型精确值返回。两个参数的时间皆是以 1970 年 1 月 1 日零时零分零秒算起的 UTC 时间。

返 回 值：返回确精的时间差距秒数。

ftime（取得目前的时间和日期）

相关函数：time，ctime，gettimeofday

表头文件：#include <sys/timeb.h>

定义函数：int ftime（struct timeb *tp）；

函数说明：ftime（）将目前时间日期由 tp 所指的结构返回。tp 结构定义为：

```
struct    timeb {
    time_t    time;
    unsigned short millitm;
    short     timezone;
    short     dstflag;
};
```

millitm　为千分之一秒

time　为从公元 1970 年 1 月 1 日至今的秒数。

timezone　为目前时区和 Greenwich 相差的时间，单位为分钟。

dstflag　为日光节约时间的修正状态，如果为非 0 代表启用日光节

约时间修正。

返 回 值：无论成功或失败都返回 0。

范 例

```
#include <sys/timeb.h>
main ()
{
  struct timeb tp;
    ftime (&tp) ;
    printf ("time: %d\n", tp.time) ;
    printf ("millitm: %d\n", tp.millitm) ;
    printf ("timezone: %d\n", tp.timezone) ;
    printf ("dstflag: %d\n", tp.dstflag) ;
}
```

执行结果

```
time: 974857398
millitm: 215
timezone: -540
dstflag: 0
```

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（）会把目前的时间由 tv 所指的结构返回，当地时区的信息则放到由 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（Western European）*/
    DST_MET       /* 中欧（Middle European）*/
    DST_EET       /* 东欧（Eastern European）*/
    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。

错误代码：EFAULT 指针 tv 或 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) ;
}
```


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

gmtime（取得目前的时间和日期）

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;
};
```

tm_sec	代表目前秒数，正常范围为 0 - 59，但允许至 61 秒。
tm_min	代表目前分数，范围为 0 - 59。
tm_hour	从午夜算起的时数，范围为 0 - 23。
tm_mday	目前月份的日数，范围为 1 - 31。
tm_mon	代表目前月份，从一月算起，范围为 0 - 11。
tm_year	从 1900 年算起至今的年数。
tm_wday	一星期中的日数，从星期日算起，范围为 0 - 6。
tm_yday	从今年 1 月 1 日算起至今的天数，范围为 0 - 365。
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 代表目前的 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);
}
```

执行结果

```
2000/10/28 Sat 8: 15: 38
```

localtime（取得当地目前的时间和日期）

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区。

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) ;
}
```

执行结果

```
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) ;
}
```

执行结果

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

settimeofday（设置目前的时间）

相关函数：time，ctime，ftime，gettimeofday

表头文件：#include <sys/time.h>

#include <unistd.h>

函数说明：int settimeofday（const struct timeval *tv，const struct timezone *tz）；

函数说明：settimeofday（）会把目前的时间设成由 tv 所指的结构信息，当地时区的信息则设成 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。

strftime（格式化日期和时间）

相关函数：time，asctime，gmtime，localtime

表头文件：#include <time.h>

定义函数：size_t strftime（char *s，size_t max，const char *format，const struct tm *tm）；

函数说明：strftime（）会将参数 tm 的时间结构，依照参数 format 所指定的字符串格式做转换，转换后的字符串内容将复制到参数 s 所指的字符串数组中，该字符串的最大长度为参数 max 所控制。

参数 format 的格式和一般格式化字符串相当类似，都是以 '%' 字符做控制，下面是其格式指令：

%a　当地星期日期的名称缩写，如 Sun。
%A　当地星期日期的完整名称，如 Sunday。
%b　当地月份的缩写。
%B　当地月份的完整名称。
%c　当地适当的日期与时间表示法。
%C　以 year/100 表示年份。
%d　月里的天数，表示法为 01 - 31。
%D　相当于 "%m/%d/%y" 格式。
%e　如同 %d 为一个月里的天数，表示法为 1 - 31。
%h　和 %b 相同。
%H　以 24 小时制表示小时数（00 - 23）。
%I　以 12 小时制表示小时数（01 - 12）。
%j　一年中的天数（001 - 366）。
%k　如同 %H，但表示法为 0 - 23。

%l 如同 %I，但表示法为 1 - 12。
%m 月份（01 - 12）。
%M 分数（00 - 59）。
%n 同 \n 。
%p 显示对应的 AM 或 PM 表示。
%P 和 %p 相同，但是用小写的 am 和 pm 表示。
%r 相当于使用 "%I：%M：%S %p" 格式。
%R 相当于使用 "%H：%M" 格式。
%s 从 1970-01-01 00：00：00 UTC 算起迄今的秒数。
%S 秒数（00 - 61）。
%t 同 \t。
%T 24 小时时间表示，相当于使用 "%H：%M：%S" 格式。
%u 一星期中的星期日期，范围 1 - 7，星期一从 1 开始。
%U 一年中的星期数（00 - 53），一月第一个星期日开始为 01。
%w 一星期中的星期日期，范围 0 - 6，星期日从 0 开始。
%W 一年中的星期数（00 - 53），一月第一个星期一开始为 01。
%x 当地适当的日期表示。
%X 当地适当的时间表示。
%y 一世纪中的年份（00 - 99）。
%Y 完整的公元年份表示。
%Z 使用的时区名称。
%% '%' 符号。

返 回 值：返回复制到参数 s 所指的字符串数组的总字符数，不包括字符串结束符号。如果返回 0，表示未复制字符串到参数 s 内，但不表示一定有错误发生。

附加说明：环境变量 TZ 和 TC_TIME 会影响此函数结果。

范 例

```
#include <time.h>
main ()
{
  char *format[] = { "%I：%M：%S %p %m/%d  %a ",
```

```
                        "%x %X %Y ",
                        NULL
                        };
  char buf[30];
  int i;
  time_t clock;
  struct tm *tm;
  time (&clock) ;
  tm = gmtime (&clock) ;
  for ( i = 0; format[i] != NULL; i++)
  {
   strftime (buf, sizeof (buf) , format[i], tm) ;
   printf ("%s => %s\n", format[i], buf) ;
  }
}
```


```
%I: %M: %S %p %m/%d  %a  => 02: 13: 15 AM 10/28  Sat
%x %X %Y  => 10/28/00 02: 13: 15 2000
```

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);

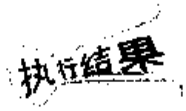
函数说明：此函数会返回从公元 1970 年 1 月 1 日的 UTC 时间从 0 时 0 分 0 秒算起到现在所经过的秒数。如果 t 并非空指针的话，此函数也会将返回值存到 t 指针所指的内存。

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) ;
}
```




```
972699100
```

tzset（设置时区以供时间转换）

相关函数：ctime，ftime，gettimeofday

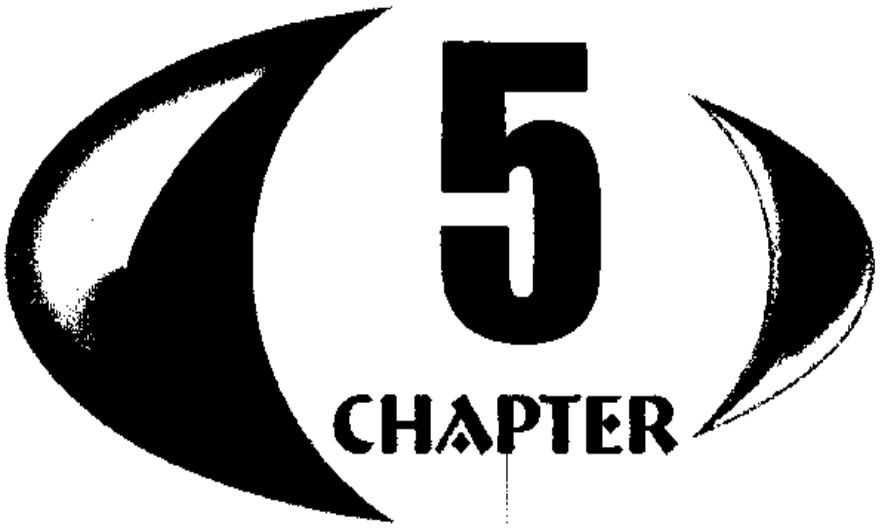
表头文件：#include <time.h>

定义函数：void tzset（void）；

extern char *tzname[2];

函数说明：tzset（）用来将环境变量 TZ 设给全局变量 tzname，也就是从环境变量取得目前当地的时区。时间转换的函数会自动调用此函数。若环境变量 TZ 未设置，全局变量 tzname 会依照/etc/localtime 找出最接近当地的时区。如果环境变量 TZ 的值为 NULL，或是无法判认，则使用 UTC 时区。

返 回 值：此函数总是调用成功，并且初始化全局变量 tzname。



字符串处理函数

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

请参考 memcmp（）。

bcopy（拷贝内存内容）

相关函数：memccpy，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：无

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 指针放在前 */
  printf ("\nmemcpy () : ") ;
  for (i = 0; i< 30; i++)
      printf ("%c ", dest[i]) ;
}
```


```
bcopy () : string string
memcpy () : string string
```

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_t n）;

返 回 值：无

附加说明：建议使用 memset（）取代

请参考 memset（）。

ffs（在一整型数中查找第一个值为真的位）

相关函数：无

表头文件：#include <string.h>

定义函数：int ffs（int i）;

函数说明：ffs（）会由低位至高位，判断参数 i 字节中每一位，将最先出现位的值为真（1）的位置返回。

返 回 值：返回值 1 到 32 之间，表示位的值最先出现真值的位置。返回值若为 0，即表示没有一个位的值为真，该字节值为零。

范 例

```
#include <string.h>

main ()
{
  int i[] = {0, 1, 2, 4, 8, 16, 32, 64};
  int j;
  for (j = 0; j < 8; j++)
      printf ("%d : %d \n", i[j], ffs (i[j]) ) ;
}
```

执行结果

```
0 : 0
1 : 1
2 : 2
4 : 3
8 : 4
16 : 5
32 : 6
64 : 7
```

index（查找字符串中第一个出现的指定字符）

相关函数：rindex，strchr，strrchr

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) ;
}
```

执行结果

```
5678901234567890123456789O
```

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 0 表示在 src 所指的内存前 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) ;
}
```

执行结果

```
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 = "01234567890123456789012345678901234567890";
 char *p;
 p = memchr (s, '5', 10) ;
 printf ("%s\n", p) ;
}
```


```
5678901234567890123456789012345678901234567890
```

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，此顺序亦为字符的值。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 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) ) ;
}
```

执行结果

```
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 b, 返回 -1 */
memcmp (a , d) : 0      /* 字符串 a = 字符串 b, 返回  0 */
```

memcpy（拷贝内存内容）

相关函数：bcopy，memccpy，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 所指的地址上。与 strcpy（）不同的是，memcpy（）会完整的复制 n 个字节，不会因遇到字符串结束 '\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]) ;
}
```

执行结果

```
strcpy () : string a )
memcpy () : string string
```

memfrob（对内存区域编码）

相关函数：strfry

表头文件：#include <string.h>

定义函数：void *memfrob（void *s，size_t n);

函数说明：memfrob（）用来将参数 s 所指的内存空间前 n 个字符与 42 作 XOR 运算，用途是可以隐藏一特定字符串内容，只要再用相同的参数调用 memfrob（）即可将内容还原。

返 回 值：返回编码成功后的内存空间地址。

附加说明：此函数为 Linux 特有。

范 例

```
#include <string.h>
main ()
```

```
{
  char a[]="This_is_memfrob";
  printf ("Before first memfrob () : %s \n", a) ;
  memfrob (a, strlen (a) ) ;
  printf ("After first memfrob ()  : %s \n", a) ;
  memfrob (a, strlen (a) ) ;
  printf ("After second memfrob () : %s \n", a) ;
}
```

执行结果

```
Before first memfrob () : This_is_memfrob
After first memfrob ()  : ~BCYuCYuGOGLXEH
After second memfrob () : This_is_memfrob
```

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（）仍然可以正确地处理，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

请参考 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';     /* 字符串结尾补上 '\0' 结束字符 */
  printf ("%s\n", s) ;
}
```

执行结果

```
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地址，否则返回 0。

范 例

```
#include <string.h>
main ()
{
  char *s = "01234567890123456789012345678901234567890";
  char *p;

  p = rindex (s, '5') ;
  printf ("%s\n", p) ;
}
```

执行结果

```
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 长度若大于 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) ;
}
```

执行结果

```
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) ) ;
}
```


```
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，strpbrk，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 = "01234567890123456789012345678890";
  char *p;

  p = strchr (s, '5') ;
  printf ("%s\n", p) ;

}
```


```
5678901234567890123456789
```

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) ) ;

}
```

执行结果

```
strcmp (a , b) : 32       /* 字符串 a > 字符串 b，返回 'a' 和 'A' 的差值 32 */
strcmp (a , c) : -31      /* 字符串 a < 字符串 c，返回 'B' 和 'a' 的差值 -31 */
```

```
strcmp (a , d) : 0        /* 字符串 a = 字符串 d, 值回 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参数 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

请参考 strcmp（）。

strcpy（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 *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) ) ;
}
```

执行结果

```
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") ) ;
}
```


```
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。


```
#include <string.h>

main ()
{
  char a[]="strdup";
  char *b;
```

```
  b = strdup (a) ;
  printf (*b[] = \"%s\"\n", b) ;
}
```


```
b[] = "strdup"
```

strfry（随机重组字符串内的字符）

相关函数：memfrob

表头文件：#include <string.h>

定义函数：char *strfry（char *string）;

函数说明：strfry（）会利用 rand（）来随机重新分配参数 string 字符串内的字符，然后返回指向参数 string 字符串的指针。

返 回 值：返回经随机重组后的字符串指针。

附加说明：由于 strfry（）会改变参数 string 字符串内容，因此参数 string 的字符串指针必须指向可写入的内存地址。

范 例

```
#include <string.h>

main ()
{
 char a[]="strfry";
 int i;
 for (i=0;i<4;i++)
    printf ("a[] = \"%s\"\n", strfry (a) ) ;

}
```

```
a[] = "ysrrtf"
a[] = "frsrty"
a[] = "ftyrsr"
a[] = "tsfryr"
```

strlen（返回字符串长度）

相关函数：无

表头文件：#include <string.h>

定义函数：size_t strlen（const char *s）;

函数说明：strlen（）用来记算指定的字符串 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) ) ;
}
```

```
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 (!strcasecmp (a, b) )
    printf ("%s = %s\n", a, b) ;
}
```

执行结果

```
aBcDeF = AbCdEf
```

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

函数说明：strcat（）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 strcat () : %s\n", a) ;
  printf ("after  strcat () : %s\n", strncat (a, b, 6) ) ;
}
```


```
before strcat () : string (1)
after  strcat () : string (1) string
```

strncmp（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 strncmp（const char *s1，const char *s2，size_t n）;

函数说明：strncmp（）用来将参数 s1 中前 n 个字符和参数 s2 字符串作比较。

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

请参考 strcmp（）。

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 strcpy () : %s\n", a) ;
  printf ("after  strcpy () : %s\n", strcpy (a, b) ) ;
}
```

执行结果

```
before strcpy () : string (1)
after  strcpy () : 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 = "01234567890123456789012345678901";
  char *p;

  p = strpbrk (s, "a1 839") ; /* 1 会最先在 s 字符串中找到 */
  printf ("%s\n", p) ;
  p = strpbrk (s, "4398") ;   /* 3 会最先在 s 字符串中找到 */
  printf ("%s\n", p) ;

}
```

执行结果

```
12345678901234567890123456789
345678901234567890123456789
```

strrchr（查找字符串中最后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 *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 = "01234567890123456789012345678901234567890";
 char *p;

 p = strrchr (s, '5') ;
 printf ("%s\n", p) ;

}
```

执行结果

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，strpbrk，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。简单地说，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";
  char *t2="abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ/- ";
  char *t3="abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ/-.
  12345678";
  printf ("%d\n", strspn (str, t1) ) ;
  printf ("%d\n", strspn (str, t2) ) ;
  printf ("%d\n", strspn (str, t3) ) ;
}
```

执行结果

```
5   /* 计算大小写字母，不包含 " "，所以返回 "Linux" 的长度 */
30  /* 包含大小写字母和 "/"、"-" 和 " "，返回到 "3" 之前的长度 */
48  /* 包含字符串 str 所有出现的字符，所以返回整个字符串长度 */
```

strstr（在一字符串中查找指定的字符串）

相关函数：index，memchr，rindex，strchr，strpbrk，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 = "01234567890123456789012345678901234567890";
  char *p;

  p = strstr (s, "901") ;
  printf ("%s\n", p) ;

}
```

执行结果

```
9012345678901234567890
```

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。

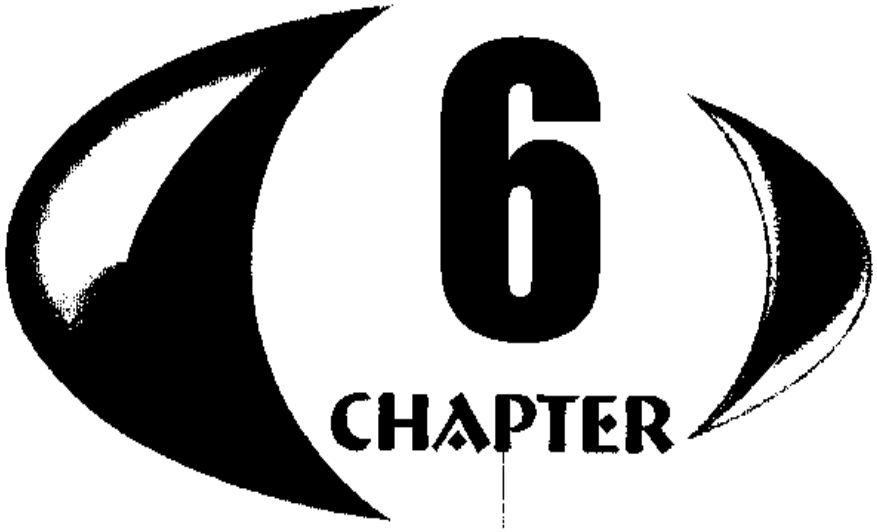
返 回 值：返回下一个分割后的字符串指针，如果已无从分割则返回 NULL。

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") ;
}
```


```
ab cd ef;gh i jkl;mnop;qrs tu vwx y;z  /* - 与 ; 字符已被 \0 字符取代 */
```



数学计算函数

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 的绝对值计算结果。

范 例

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

执行结果

```
|-12| = 12
```

acos（取反余弦函数值）

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) ;
}
```

执行结果

```
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) ;
}
```

执行结果

```
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) ;
}
```

```
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) ;
}
```

```
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]) ) ;
}
```

执行结果

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

cos（取余弦函数值）

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

执行结果

```
cos (0.5) = 0.877583
```

cosh（取双曲线余弦函数值）

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 的双曲线余弦值。

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) ;
}
```

```
cosh (0.5) = 1.127626
```

div（取得两整型数相除后的商及余数）

相关函数：ldiv

表头文件：#include <stdlib.h>

定义函数：div_t div（int numer，int denom）；

函数说明：div（）函数会计算参数 numer/denom，然后将相除后的商及余数由 div_t 结构返回。div_t 结构定义如下：

```
typedef struct
    {
        int quot;   /* 商数 */
        int rem;        /* 余数 */
    } div_t;
```

返 回 值：返回 div_t 结构，包含商数及余数。

范 例

```
/* 计算 67/4 的商及余数 */
#include <stdlib.h>
main ()
{
  div_t answer;
  answer = div (67, 4) ;
  printf ("Quotient = %d, remainder = %d\n", answer.quot, answer.rem) ;
}
```

```
Quotient = 16, remainder = 3
```

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 ex 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) ;
}
```

执行结果

```
e^10 = 22026.465795
```

fabs（计算浮点型数的绝对值）

相关函数：abs，labs

表头文件：#include <math.h>

定义函数：double fabs（double x）;

函数说明：fabs（）用来计算浮点型数 x 的绝对值，然后将结果返回。

返 回 值：返回参数 x 的绝对值计算结果。

范 例

```
#include <math.h>
main ()
{
 double answer;
 answer = fabs (-3.141592) ;
 printf ("|-3.141592| = %f\n", answer) ;
}
```

执行结果

```
|-3.141592| = 3.141592
```

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) ;
}
```

执行结果

```
exp = 11
fraction = 0.500000     /* 0.5* (2^11) = 1024 */
```

hypot（计算直角三角形斜边长）

相关函数：sqrt

表头文件：#include <math.h>

定义函数：double hypot（double x，double y）;

函数说明：hypot（）是调用 sqrt（x*x + y*y）然后将计算结果返回。通常用来计算直角三角形斜边长，参数 x 和 y 为两边边长，或是计算原点到点（x，y）的距离。

返 回 值：返回（x*x + y*y）的平方根值。

附加说明：使用 GCC 编译时请加入 -lm。

范 例

```
/* 计算点 (3, 4) 至原点的距离 */

#include <math.h>

main ()
{
  double distance;
```

```
  distance = hypot ( 3, 4) ;
  printf ("distance of the point (3, 4) from the origin is %f\n", distance) ;

}
```

执行结果

```
distance of the point (3, 4) from the origin is 5.000000
```

labs（计算长整型数的绝对值）

相关函数：abs，fabs

表头文件：#include <stdlib.h>

定义函数：long int labs（long int j）;

函数说明：labs（）用来计算参数 j 的绝对值，然后将结果返回。

返 回 值：返回 j 的绝对值计算结果。

附加说明：使用 GCC 编译时请加入 -lm。

范 例

```
#include <stdlib.h>

main ()
{
  long int answer;

  answer = labs (-2000) ;
  printf ("|-2000| = %d\n", answer) ;
}
```

执行结果

```
|-2000| = 2000
```

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 乘上 2 的 exp 次方值，即 x * 2exp。

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) ;
}
```

执行结果

```
3*2^ (2)   = 12.000000
```

ldiv（取得两长整数相除后的商及余数）

相关函数：div

表头文件：#include <stdlib.h>

定义函数：ldiv_t ldiv（long int numer，long int denom）；

函数说明：ldiv（）函数会计算参数 numer/denom，然后将相除后的商及余数由 ldiv_t 结构返回。ldiv_t 结构定义如下：

```
typedef struct
{
        long int quot;  /* 商数  */
        long int rem,   /* 余数  */
} ldiv_t;
```

返 回 值：返回 ldiv_t 结构，包含商数及余数。

范 例

```
/* 计算 2653589/79323 的商及余数 */

#include <stdlib.h>

main ()
{
  ldiv_t answer;
  answer = ldiv (2653589, 79323) ;
  printf ("Quotient = %d, remainder = %d\n", answer.quot, answer.rem) ;
}
```


```
Quotient = 33, remainder = 35930
```

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) ;
}
```

执行结果

```
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) ;
}
```

执行结果

```
log10 (100) = 2.000000
```

modf（将浮点型数分解成整数与小数）

相关函数：frexp

表头文件：#include <math.h>

定义函数：double modf（double x，double *iptr）;

函数说明：modf（）用来将参数 x 的浮点型数分解成整数和小数。小数部分直接返回，整数部分则借参数 iptr 指针返回。

返 回 值：返回参数 x 的小数部分，整数部分则存于 iptr 指针所指的地址。

附加说明：使用 GCC 编译时请加入 -lm。

范 例

```
/* 分解 3.141592 的整数与小数部分 */
#include <math.h>
```

```
main ()
{
 double integral;
 double fractional;

 fractional = modf (3.141592, &integral) ;
 printf ("integral   = %f\n", integral) ;
 printf ("fractional = %f\n", fractional) ;
}
```

执行结果

```
integral   = 3.000000
fractional = 0.141592
```

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 xy 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) ;
}
```

执行结果

```
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) ;
}
```

执行结果

```
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) ;
}
```

执行结果

```
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 为负值。

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) ;
}
```

执行结果

```
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) ;
}
```

执行结果

```
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。

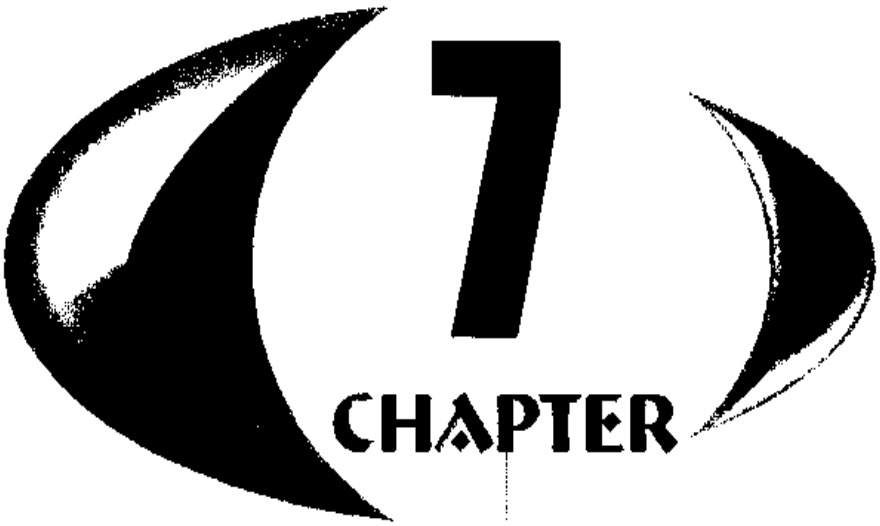
附加说明：使用 GCC 编译时请加入 -lm。

范 例

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

执行结果

```
tanh (0.5) = 0.462117
```



用户和组函数

cuserid（取得用户帐号名称）

相关函数：getlogin

表头文件：#include <stdio.h>

定义函数：char * cuserid（char *string）;

函数说明：cuserid（）会将用户帐号名称复制到参数 string 所指的字符串数组中，如果参数 string 为空指针（NULL），cuserid（）会自动配置一静态的字符串数组，然后将指向此字符串的指针返回，此自动配置的空间大小由定义在 stdio.h 中的 L_cuserid 值决定（POSIX 定义为 9）。注意，cuserid（）自动配置的字符串数组会由调用 getlogin（）或再调用 cuserid（）时所覆盖。

返 回 值：请尽量使用 getpwuid（geteuid（））；来取代此函数。

附加说明：返回指向用户帐号名称的字符串指针。

范 例

```
#include <stdio.h>
main ()
{
  printf ("I am %s.\n", cuserid (0) ) ;
}
```

执行结果

```
I am root.  /* 当使用 root 身份执行范例程序时 */
```

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：无

附加说明：无

范 例

请参考 getgrent（）或 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：无

附加说明：无

范 例

请参考 getpwent（）或 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：无

附加说明：无

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 *fgetgrent（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) ;
}
```


```
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) ;
  }
}
```

执行结果

```
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
```

getegid（取得有效的组识别码）

相关函数：getgid，setgid，setregid

表头文件：#include <unistd.h>

#include <sys/types.h>

定义函数：gid_t getegid（void）;

函数说明：getegid()用来取得执行目前进程有效的组识别码。有效的组识别码（effective group ID）用来决定进程执行时组的权限。

返 回 值：返回有效的组识别码。

范 例

```
main ()
{
  printf ("egid is %d\n", getegid () ) ;
```

```
}
```

执行结果

```
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（effective user ID）用来决定进程执行的权限，借由改变此值，进程可以获得额外的权限。倘若执行文件的set ID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 set ID（SUID）位，而当任何用户在执行 passwd 时其有效的用户识别码则会被设成 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 () ) ;
}
```

执行结果

```
euid is 0 /* 当使用 root 身份执行范例程序时 */
```

getgid（取得真实的组识别码）

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：getuid（）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 () ) ;
}
```

执行结果

```
gid is 0 / * 当使用 root 身份执行范例程序时 */
```

getgrent（从组文件文件中取得帐号的数据）

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;        /* 组成员帐号 */
        };
```

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。

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。

错误代码：ENOMEM　内存不足，无法配置 group 结构。

范 例

```
/* 列出所有组数据 */
#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 () ;
}
```

执行结果

```
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 指定的组识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->gr_mem[i++]) ;
  printf ("\n") ;
}
```

执行结果

```
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，getgrent，getgruid

表头文件：#include <grp.h>

#include <sys/types.h>

定义函数：struct group *getgrnam（const char *name）;

函数说明：getgrnam（）用来逐一搜索参数 name 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 ()
{

  struct 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") ;

}
```

执行结果

```
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：getgroups（）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]) ;
}
```


```
0 : 0
1 : 1
2 : 2
3 : 3
4 : 4
5 : 6
6 : 10
```

getlogin（取得登录的用户帐号名称）

相关函数： cuserid

表头文件： #include <unistd.h>

定义函数： char * getlogin（void）;

函数说明： getlogin（）会从 /var/run/utmp 中查找登录目前终端机的用户帐号名称，找不到相关数据就返回一空指针（NULL），如果找到帐号名称就自动配置一字符串数组，把帐号名称复制到此数组，最后将指向此字符串的指针返回。注意，getlogin（）自动配置的字符串数组会由调用 cuserid 或再调用 getlogin（）时所覆盖。

附加说明： getlogin（）会有潜在的安全性问题，使用时请留意！环境变量 LOGNAME 同样也是取得登录的用户帐号名称。

返 回 值： 返回指向用户帐号名称的字符串指针。

范 例

```
#include <unistd.h>
main ()
{
  printf ("I am %s.\n", getlogin () ) ;
}
```

执行结果

```
I am root.        /* 当使用 root 身份执行范例程序时 */
```

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 所指定的用户帐号数据，找不到相关数据就返回 -1。所返回的 buf 字符串格式如下：

帐号：密码：用户识别码（uid）：组识别码（gid）：全名：根目录：shell

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"。

返 回 值：返回 0 表示成功，有错误发生时返回 -1。

范 例

```
#include <pwd.h>
#include <sys/types.h>
main ()
```

```
{
  char buffer[80];
  getpw (0, buffer) ;
  printf ("%s\n", buffer) ;
}
```


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

getpwent（从密码文件中取得帐号的数据）

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;        /* 用户帐号 */
    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 */
};
```

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。

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。

错误代码：ENOMEM 内存不足，无法配置 passwd 结构。

范 例

```
/* 列出所有帐号数据 */
#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 () ;
}
```

执行结果

```
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帐号的数据）

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 指定的帐号名称，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 帐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) ;
}
```

执行结果

```
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 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 uid=6 的帐号名称、识别码和根目录 */
#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) ;
}
```

执行结果

```
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 () ) ;
}
```

执行结果

```
uid is 0    /* 当使用 root 身份执行范例程序时 */
```

getutent（从 utmp 文件中取得帐号登录数据）

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];  /* 登录帐号  */
   char ut_host[UT_HOSTSIZE];  /* 登录帐号的远程主机名称  */
   struct exit_status ut_exit; /* 当类型为 DEAD_PROCESS 时进程的结束
                                  状态  */
   long int ut_session;        /* Session 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 有下列几种类型：

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 的进程。
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
```

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。

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 *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 () ;
}
```

执行结果

```
/* 表示有三个 root 帐号分别登录 /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，或 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) ;
  }
}
```

执行结果

```
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

USER_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) ;
  }
}
```

执行结果

```
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：initgroupst（）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，错误代码存于 errno。

logwtmp（将一登录数据记录到 wtmp 文件）

相关函数：updtmp，getutent

表头文件：#include <utmp.h>

定义函数：void logwtmp（const char *line，const char *name，const char *host）；

函数说明：logwtmp（）会依参数 line、name 和 host 所指定的装置名、帐号和远程主机等数据自动建立一 utmp 结构，然后调用 updwtmp（）来将此记录写入到 wtmp 文件（/var/log/wtmp）。

附加说明：需要有 wtmp 文件的写入权限。

返 回 值：无

范 例

```
#include <utmp.h>
main ()
{
  logwtmp ("pts/2", "kids", "www.gnu.org") ;
}
```


```
/* 表示有三个 root 帐号分别登录 /dev/pts/0，/dev/pts/1，/dev/pts/2 */
7 root pts/0
7 root pts/1
7 root pts/2
```

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，如果没找到相符的记录则会加入到 utmp 文件尾。utmp 结构请参考 getutent（）。

附加说明：需要有写入 /var/run/utmp 的权限。

返 回 值：无

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) ;
}
```

执行结果

```
/* 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
```

setegid（设置有效的组识别码）

相关函数：setgid，setregid，setfsgid

表头文件：#include <unistd.h>

定义函数：int setegid（gid_t egid）;

函数说明：setegid（）用来重新设置执行目前进程的有效组识别码。在 Linux 下，setegid（egid）相当于 setregid（-1，egid）。

返 回 值：执行成功则返回 0，失败则返回 -1，错误代码存于 errno。

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识别码。一般情况下，文件系统的组识别码（fsgid）与有效的组识别码（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。

错误代码：EPERM　权限不够，无法完成设置。

附加说明：此函数为 Linux 特有。

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（euid）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。

错误代码：EPERM　权限不够，无法完成设置。

附加说明：此函数为 Linux 特有。

setgid（设置真实的组识别码）

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识别码（real gid）设成参数 gid 值。如果是以超级用户身份执行此调用，则 real、effective 与 saved gid 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>

定义函数：#include <sys/types.h>
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：无

附加说明：请参考 setpwent（）。

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）。

附加说明：只有 root 权限才能使用此函数。

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）。

setpwent（从头读取密码文件中的帐号数据）

相关函数：getpwent，endpwent

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：无

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 () ;      /* 从头开始读取 */
 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 () ;
}
```

执行结果

```
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   /* 因为 setpwent () 所以从头开始 */
```

setregid（设置真实及有效的组识别码）

相关函数：setgid，setegid，setfsgid

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识别码，将参数 egid 设置为目前进程的有效组识别码。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（effective-userid）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，如果只是想暂时抛弃 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，进程的 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。

返 回 值：无

范 例

请参考 setpwent（）或 setgrent（）。

updwtmp（将一登录数据记录到 wtmp 文件）

相关函数：logwtmp，pututline

表头文件：#include <utmp.h>

定义函数：void updwtmp（const char *wtmp_file，const struct utmp *ut);

函数说明：updwtmp（）用来将先前 logwtmp（）所建立的 utmp 结构写入到文件内。参数 ut 为 logwtmp（）建立的 utmp 结构数据，参数 wtmp_file 则为欲写入的 wtmp 文件（/var/log/wtmp)。

附加说明：需要有 wtmp 文件的写入权限。

返 回 值：无

范 例

```
#include <unistd.h>
#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") ;
   updwtmp ("/var/log/wtmp", &ut) ;
}
```

执行结果

```
/* 利用指令 last 来观察 */
kids    pts/1    www.gnu.org    Mon Dec 11 10: 31   still logged in
```

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：无

数据加密函数

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) ) ;
```

```
}
```


```
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
```

getpass（取得一密码输入）

相关函数：crypt

表头文件：#include <unistd.h>

定义函数：char *getpass（const char * prompt）;

函数说明：getpass（）会显示参数 prompt 所指的字符串，然后从 /dev/tty 中读取所输入的密码，若无法从 /dev/tty 中读取则会转从标准输入设备中读取密码。所输入的密码长度限制在 128 个字符，包含结束字符 NULL，超过长度的字符及换行字符（'\n'）将会被忽略。在输入密码时 getpass（）会关闭字符回应，并忽略一些信号如 CTRL-C 或 CTRL-Z 所产生的信号。

返 回 值：返回一个指向以 NULL 结尾的密码字符串。

附加说明：为了系统安全考虑，一般在使用 getpass（）输入密码后，该密码最好尽快处理完毕，然后将该密码字符串清除。

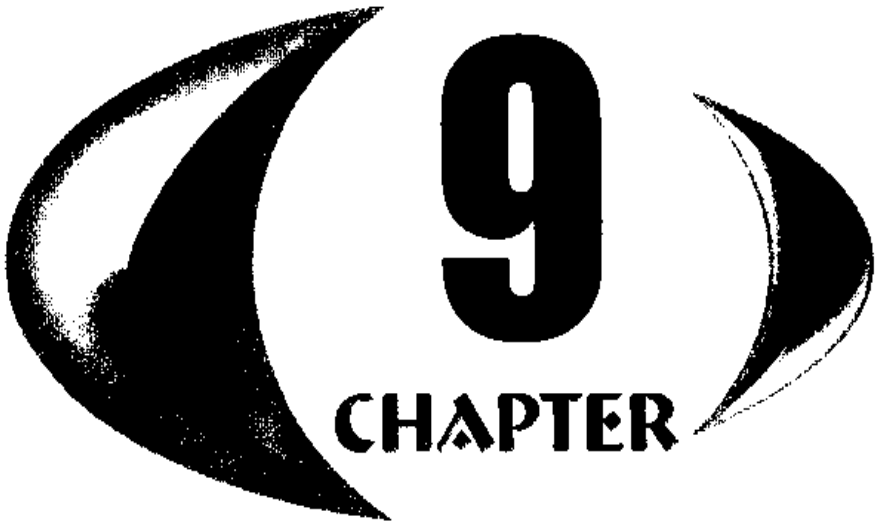
范 例

```
#include <unistd.h>
main ()
{
 char passwd[] = "password";
 char *ptr;
 ptr = getpass ("Input Password : ") ;
```

```
  if (!strcmp (passwd, ptr) )
     printf ("Correct\n") ;
  else
     printf ("Incorrect\n") ;

}
```


```
Input Password:      /* 输入 test，不会显示 */
Incorrect            /* 密码错误 */
Input Password :     /* 输入 password */
Correct              /* 密码正确 */
```



数据结构函数

bsearch（二元搜索）

相关函数：qsort

定义函数：#include <stdlib.h>

函数说明：void *bsearch（const void *key，const void *base，size_t nmemb，size_t size，int（*compar）（const void *，const void *））；

返 回 值：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 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 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) ;
      }
    }
}
```

执行结果

```
>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                          /* 再输入一次 hello 字符串 */
found : hello                   /* 这次找到了 */
```

hcreate（建立哈希表）

相关函数：hsearch，hdestroy

表头文件：#include <search.h>

定义函数：int hcreate（unsigned nel）；

函数说明：hcreate（）用来建立哈希表，参数 nel 为哈希表最大量的估计值。

返 回 值：如果无法成功建立哈希表则返回 NULL。

hdestory（删除哈希表）

相关函数：hcreate，hsearch

表头文件：#include <search.h>

定义函数：void hdestroy（void）；

函数说明：hdestory（）用来删除目前使用的哈希表，如果想建立新的哈希表，则必须先利用此函数删除目前的哈希表。

返 回 值：无

hsearch（哈希表搜索）

相关函数：hcreate，hdestroy

表头文件：#include <search.h>

定义函数：ENTRY *hsearch（ENTRY item，ACTION action）；

函数说明：hsearch（）会从目前的哈希表中查找参数 item 结构中的 key 关键数据，如果参数 action 为 FIND，则找到数据后就会将关键数据的地址返回，找

不到则返回 NULL。如果参数 action 为 ENTER，找不到关键数据时会主动把该数据加入哈希表中。

结构 ENTRY 定义如下：

```
typedef struct entry
{
    char *key;
    char *data;
} ENTRY;
```

key 为一字符串指针，指向以 NULL 为结尾的字符串，用来搜索使用。data 指针则指向与 key 相关的数据地址。

返 回 值：找到关键数据则返回找到的地址，如果在哈希表中找不到关键数据则返回 NULL。

insque（加入一项目至队列中）

相关函数：remque

表头文件：#include <stdlib.h>

定义函数：void insque（struct qelem *elem，struct qelem *prev）；

函数说明：insque（）和 remque（）用来操作一双向链接串连的队列，

结构 struct qelem 定义如下：

```
struct qelem {
  struct    qelem *q_forw;
  struct    qelem *q_back;
  char      q_data[1];
};
```

insque（）将参数 elem 所指的项目插到队列中参数 prev 所指的项目后面，此参数 prev 不可为空指针（NULL）。

返 回 值：无

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。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

请参考 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) ) ;  /* 利用 strcmp* () 比较字符串 */
}
main ()
{
 char data[NMEMB][SIZE] = { "linux", "freebsd", "solaris", "sunos", "windows" };
 char key[80], *base, *offset;
 int i, nmemb = NMEMB, size = SIZE;
 for (i = 1; i < 5; i++) {
  fgets (key, sizeof (key) , stdin) ;   /* 输入关键字 */
  key[strlen (key) -1] = '\0';/* 去掉 \n 字符 */
  base = data[0]; /* 设置从 data 数组开头搜索 */
  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) ;
  }
```

```
  }
}
```


```
linux                    /* 输入字符串 linux */
found : linux            /* 找到 linux 字符串 */
os/2                     /* 输入字符串 os/2 */
os/2 not found!          /* 找不到 os/2 字符串 */
Add os/2 to data array   /* 将 os/2 字符串加入到 data 数组 */
os/2                     /* 再输入 os/2 */
found : os/2             /* 这次找到了 */
```

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：无

范 例

```
#define nmemb   7   /* 数组中有 7 个元素 */
#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 };  /* 有 7 个整数 */
  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") ;

}
```

执行结果

```
3 102 5 -2 98 52 18     /* 原始未排序的数据 */
-2 3 5 18 52 98 102     /* 经 qsort () 排序过后的数据 */
```

remque（从队列中删除一项目）

相关函数：insque

表头文件：#include <stdlib.h>

定义函数：void remque（struct qelem *elem）;

函数说明：insque（）和 remque（）用来操作一双向链接串连的队列，结构 struct qelem 定义如下：

```
struct qelem {
        struct  qelem *q_forw;
        struct  qelem *q_back;
```

```
        char    q_data[1];
};
```

remque（）用来将参数 elem 所指的项目从队列中删除。

返 回 值：无

tdelete（从二叉树中删除数据）

相关函数：tsearch，tfind，twalk

表头文件：#include <search.h>

定义函数：void *tdelete（const void *key，void **rootp，int （*compar）（const void *，const void *））；

函数说明：tdelete（）用来将参数 key 指向的关键数据，从参数 rootp 指向的二叉树内移走。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：找到关键数据予以删除，并将其父结点指针返回。如果在二叉树中找不到关键数据，则返回 NULL。

tfind（搜索二叉树）

相关函数：tsearch，tdelete，twalk

表头文件：#include <search.h>

定义函数：void *tfind（const void *key，const void **rootp，int（*compar）（const void *，const void *））；

函数说明：tfind（）和 tsearch（）非常类似。参数 key 指向欲查找的关键数据，参数 rootp 指向二叉树的根节点地址，与 tsearch（）不同的是，如果在二叉数中找不到关键数据，则不会加入此数据至二叉树内。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：找到关键数据则返回找到的地址，如果在二叉树中找不到关键数据，则返回 NULL。

tsearch（二叉树）

相关函数：qsort，bsearch，hsearch，lsearch，tfind，tdelete，twalk

表头文件：#include <search.h>

定义函数：void *tsearch（const void *key，void **rootp，int（*compar）（const void *，const void *））；

函数说明：tsearch（）用来建利二叉树或搜索二叉树。参数 key 指向欲查找的关键数据，参数 rootp 指向二叉树的根节点地址，如果此参数为 NULL，就会造出新的二叉树。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：找到关键数据则返回找到的地址，如果在二叉树中找不到关键数据，则会将此数据加入到二叉树中，然后将新加入的地址返回。

twalk（走访二叉树）

相关函数：tsearch，tfind，tdelete

表头文件：#include <search.h>

定义函数：void twalk（const void *root，void（*action）（const void *nodep，const VISIT which，const int depth））；

函数说明：twalk()用来走访已建立好的二叉树。参数 root 指向二叉树的根节点地址，参数 action 为一函数指针，这个函数为处理函数，每次进入一个节点都会调用此函数。传给 action 函数的第一个参数 nodep 为目前节点的指针，第二个参数 which 为一整数值，代表四种走访方式：

- **preorder**：先走节点，其次走访左节点，最后走访右节点。
- **postorder**：先走访左节点，其次走访节点，最后走访右节点。
- **endorder**：先走访左节点，其次走访右节点，最后走访节点。
- **leaf**：节点为唯一被走访的，没有其他子节点。

第三个参数代表目前节点的深度，如果是根节点则为 0。

返 回 值：无

随机数函数

drand48（产生一个正的浮点型随机数）

相关函数：rand，erand48，lrand48，nrand48，mrand48，jrand48，srand48

表头文件：#include <stdlib.h>

定义函数：double drand48（void）;

函数说明：drand48（）会返回一正的浮点型随机数，范围在 0.0 至 1.0 间。

返 回 值：返回 0.0 至 1.0 之间的随机数。

范 例

```
#include <stdlib.h>
main ()
{
  int i;
  for (i = 0; i<10; i++)
    printf ("%f\n", drand48 () ) ;

}
```

执行结果

```
0.000000
0.000985
0.041631
0.176643
0.364602
0.091331
0.092298
0.487217
0.526750
0.454433
```

erand48（产生一个正的浮点型随机数）

相关函数：rand，drand48，lrand48，nrand48，mrand48，jrand48，srand48

表头文件：#include <stdlib.h>

定义函数：double erand48（unsigned short int xsubi[3]）；

函数说明：erand48（）会返回一正的浮点型随机数值，范围在 0.0 至 1.0 间。参数 xsubi 数组存放随机数方程序所需之初值，详见 seed48（）。

返 回 值：返回 0.0 至 1.0 之间的随机数。

范 例

```
#include <stdlib.h>
main ()
{
  unsigned short int xsubi[3] = { 1, 2, 3 };
  int i, j;
  for (i = 0; i<10; i++) {
    for (j=0; j<3; j++) printf ("%6d ", xsubi[j]) ;
    printf ("random number : %f\n", erand48 (xsubi) ) ;
  }
}
```

执行结果

```
/* 可看出初值与随机数的变化 */
     1       2       3     random  number : 0.441996
 59000   43974   28966     random  number : 0.263128
 61731   23903   17244     random  number : 0.654138
  7666   39619   42869     random  number : 0.420649
 11285   43283   27567     random  number : 0.023469
 41724    4128    1538     random  number : 0.764511
 52567   63651   50102     random  number : 0.992443
 38934   50798   65040     random  number : 0.053484
```

```
34153   8706    3505    random number : 0.117352
9152    51643   7690    random number : 0.437919
```

initstate（建立随机数状态数组）

相关函数：random，setstate

表头文件：#include <stdlib.h>

定义函数：char *initstate（unsigned int seed，char *state，int n）；

函数说明：initstate（）用来初始化 random（）所使用的数组，参数 n 为数组大小，参数 seed 为初始化用的随机数种子。

返 回 值：返回调用 initstate（）前 random（）所使用的数组。

附加说明：EINVAL 参数 state 数组大小不足 8 个字符长。

jrand48（产生一个长整型数随机数）

相关函数：rand，erand48，drand48，nrand48，lrand48，mrand48，srand48

表头文件：#include <stdlib.h>

定义函数：long int jrand48（unsigned short int xsubi[3]）；

函数说明：jrand48（）会返回一个长整型数随机数，范围在-231 到 231 之间。参数 xsubi 数组存放随机数方程序所需的初值，详见 seed48（）。

返 回 值：返回 -231 至 231 之间的随机数。

范 例

```
#include <stdlib.h>
main ()
{
```

```
  unsigned short int xsubi[3] = { 1, 2, 3 };
  int i, j;
  for (i = 0; i<10; i++) {
    for (j=0; j<3; j++) printf ("%6d ", xsubi[j]) ;
   printf ("random number : %d\n", jrand48 (xsubi) ) ;
  }
}
```


```
/* 可看出初值与随机数的变化 */

    1       2       3     random  number : 1898359750
 59000   43974   28966    random  number : 1130126687
 61731   23903   17244    random  number : -662018755
  7666   39619   42869    random  number : 1806674195
 11285   43283   27567    random  number : 100798496
 41724    4128    1538    random  number : -1136064675
 52567   63651   50102    random  number : -2115028590
 38934   50798   65040    random  number : 229712386
 34153    8706    3505    random  number : 504023483
  9152   51643    7690    random  number : 1880849356
```

lcong48（设置 48 位运算的随机数种子）

相关函数：srand48，seed48，drand48，lrand48，mrand48

表头文件：#include <stdlib.h>

定义函数：void lcong48（unsigned short int param[7]）;

函数说明：48 位运算的随机数函数是根据线性调和（linear congruential）演算法来产生随机数：

```
Xn+1 = (aXn + c) mod m ,  (n >= 0)
```

其中常数 a = 0x5DEECE66D，c = 0xB，m = 248。

seed48 ( ) 可以用来改变 Xi 的初值 (详见 seed48 ( )), 但无法改变常数值。lcong48 ( ) 除了可改变初值, 亦可改变常数 a 与 c。参数 param 声明成七个 unsigned short int 的数组, param[0-2]为上述随机数方程序的初值 (相当于 seed48 ( ) 的参数 seed16v[3]), param[3-5] 代表常数 a 值, param[6]则为 c 值。

返 回 值: 无

附加说明: 如果之后调用了 srand48 ( ) 或 seed48 ( ), 则常数 a 与 c 的数值会恢复上述的默认值。

范 例

```
#include <stdlib.h>
main ()
{
 unsigned short int param[7] = { 1, 2, 3, 11, 22, 12, 20 };
 int i;
 lcong48 (param) ;
 for (i = 0; i < 10; i++) {
  printf ("%d\n", lrand48 () ) ;
 }
```

执行结果

```
}
2916374     /* 可与 seed48 () 范例比较 */
75989575
1818405008
36170157
839018925
1994498779
761662626
```

lrand48（产生一个正的长整型随机数）

相关函数：rand，erand48，drand48，nrand48，mrand48，jrand48，srand48

表头文件：#include <stdlib.h>

定义函数：long int lrand48（void）;

函数说明：lrand48（）会返回一个正的长整型随机数值，范围在 0 到 231 之间。

返 回 值：返回 0 至 231 之间的随机数。

范 例

```
#include <stdlib.h>
main ()
{
  int i;
  for (i = 0; i<10; i++)
    printf ("%d\n", lrand48 () ) ;
}
```

执行结果

```
0
2116118
89401895
379337186
782977366
196130996
198207689
1046291021
1131187612
975888346
```

60

mrand48（产生一个长整型随机数）

相关函数：rand，erand48，drand48，nrand48，lrand48，jrand48，srand48

表头文件：#include <stdlib.h>

定义函数：long int mrand48（void）;

函数说明：mrand48（）会返回一个长整型随机数值，范围在 -231 到 231 之间。

返 回 值：返回 -231 至 231 之间的随机数。

范 例

```
#include <stdlib.h>
main ()
{
  int i;
  for (i = 0; i<10; i++)
    printf ("%d\n", lrand48 () ) ;
}
```

执行结果

```
0
4232237
178803790
758674372
1565954732
392261992
396415378
2092582042
-114891576
1951776693
```

nrand48（产生一个正的长整型随机数）

相关函数： rand，erand48，drand48，lrand48，mrand48，jrand48，srand48

表头文件： #include <stdlib.h>

定义函数： long int nrand48（unsigned short int xsubi[3]）；

函数说明： nrand48（）会返回一个正的长整型随机数值，范围在 0 到 231 之间。参数 xsubi 数组存放随机数方程序所需的初值，详见 seed48（）。

返 回 值： 返回 0 至 231 之间的随机数。

范 例

```
#include <stdlib.h>
main ()
{
  unsigned short int xsubi[3] = { 1, 2, 3 };
  int i, j;
  for (i = 0; i<10; i++) {
    for (j=0; j<3; j++) printf ("%6d ", xsubi[j]) ;
    printf ("random number : %d\n", nrand48 (xsubi) ) ;
  }
}
```

执行结果

```
/* 可看出初值与随机数的变化 */

   1     2      3     random  number : 949179875
59000  43974  28966   random  number : 565063343
61731  23903  17244   random  number : 1404751201
7666   39619  42869   random  number : 903337097
11285  43283  27567   random  number : 50399248
41724  4128   1538    random  number : 1641774161
52567  63651  50102   random  number : 2131256119
```

```
38934  50798   65040   random number : 114856193
34153  8706    3505    random number : 252011741
9152   51643   7690    random number : 940424678
```

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，此范列未设随机数种子，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) ;
  }
}
```


```
9 4 8 8 10 2 4 8 3 6    /* 第一次执行 */
9 4 8 8 10 2 4 8 3 6    /* 由于未设随机数种子，故产生之随机数固定 */
```

random（产生随机数）

相关函数：rand，srandom

表头文件：#include <stdlib.h>

定义函数：long int random（void）;

函数说明：random（）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 srandom（）设好随机数种子，如果未设随机数种子，random（）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 srandom（）。

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

请参考 rand（）。

seed48（设置 48 位运算的随机数种子）

相关函数：srand48，lcong48，drand48，lrand48，mrand48

表头文件：#include <stdlib.h>

定义函数：unsigned short int *seed48（unsigned short int seed16v[3]）;

函数说明：48 位运算的随机数函数是根据线性调和（linear congruential）演算法来产生随机数：

Xn+1 = (aXn + c) mod m , (n >= 0)

其中常数 a = 0x5DEECE66D，c = 0xB，m = 248。参数 seed16v 声明成三个 unsigned short int 的数组，长度即为 48 位，此传入的数值为上述随机数方程序的初值。seed48 ( ) 的返回值为和 seed16v 相同声明的数组，数组内容为调用 seed48 ( ) 前方程序采用的初值。

返 回 值：调用 seed48 ( ) 前随机数方程序采用的初值。

范 例

```
#include <stdlib.h>
main ()
{
 unsigned short int *p, seed16v[3] = { 1, 2, 3 }; /* 初值设 1, 2, 3 */
 int i = 0;
 p = seed48 (seed16v) ;          /* 原始初值存于 *p */
 for (i = 0; i < 3; i++) printf ("%d", * (p+i) ) ;
 printf ("\n") ;
 for (i = 0; i < 10; i++) {
   printf ("%d\n", lrand48 () ) ;
 }
}
```

执行结果

```
000          /* 调用 seed48 () 前方程序所采用的初值 */
949179875
565063343
1404751201
903337097
50399248
1641774161
2131256119
114856193
252011741
```

setstate（建立随机数状态数组）

相关函数：random，initstate

表头文件：#include <stdlib.h>

定义函数：char *setstate（char *state）;

函数说明：setstate（）用来建立 random（）所使用的随机数状态数组。参数 state 指向新的随机数数组，此数组会供产生随机数的函数使用。

返 回 值：返回调用 setstate（）前 random（）所使用的数组。

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 必需是个整数，通常可以利用 getpid（）或 time（0）的返回值来当作 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：无

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) ) ;      /* 利用 time (0) 当作随机数子 */
```

```
    for (i = 0; i < 10; i++)
    {
        j=1+ (int) (10.0*rand () / (RAND_MAX+1.0) ) ;
        printf ("%d ", j) ;
    }
}
```


```
5 8 8 8 10 2 10 8 9 9   /* 第一次执行 */
2 9 7 4 10 3 2 10 8 7   /* 第二次执行 */
```

srand48（设置48位运算的随机数种子）

相关函数：seed48，lcong48，drand48，lrand48，mrand48

表头文件：#include <stdlib.h>

定义函数：void srand48（long int seedval）;

函数说明：srand48（）用来设置 drand48（）/lrand48（）/mrand48（）产生随机数时的随机数种子。参数 seed 必需是个整数，通常可以利用 getpid（）或 time（0）的返回值来当作 seed。如果每次 seed 都设相同值，所产生的随机数值每次就会一样。

返 回 值：无

范 例

请参考 lrand48（）

srandom（设置随机数种子）

相关函数：rand，random

表头文件：#include <stdlib.h>

定义函数：void srandom（unsigned int seed）;

函数说明：srandom（）用来设置 random（）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 getpid（）或 time（0）的返回值来当作 seed。如果每次 seed 都设相同值，random（）所产生的随机数值每次就会一样。

返 回 值：无

范 例

请参考 srand（）。

初级I/O函数

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，而 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

请参考 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_tmode）；

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) ,
```

```
    mode_t mode) ;
```

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（file descriptor），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（device file），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跟着改变。

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。

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：dup2（）相当于调用 fcntl（oldfd，F_DUPFD，newfd）；请参考 fcntl()

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，有下列几种情况：

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;   /* 锁定的型态 */
 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_ype 有三种型态：

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段。

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 作 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。

下列是较特别的使用方式：

（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符。若有错误则返回 -1，errno 会存放错误代码。

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。

ESPIPE　参数 fildes 所指的文件为一管道（pipe）、Socket 或 FIFO。

EINVAL　参数 whence 非有效值。

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

请参考本函数说明。

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方式

```
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) ;
}
```

执行结果

```
template = template-1gZcbo     /* 文件 template-1gZcbo 已被建立 */
```

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，且欲打开的文件为符号连接（symbol link），则会开文件失败。
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（synchronous）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（symbol link），则会令开文件失败。
O_DIRECTORY	如果参数 pathname 所指的文件并非为一目录，则会令

开文件失败。此为 Linux 2.2 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-umask)。

S_IRWXU00700	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（denied）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 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（）的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];
  /* 打开 /tmp/temp 作写入，如果该文件不存在则建立该文件 */
  fd = open ("/tmp/temp", O_WRONLY | O_CREAT) ;
  write (fd, s, sizeof (s) ) ;
  close (fd) ;
  /* 打开 /tmp/temp 准备作读取动作 */
  fd = open ("/tmp/temp", O_RDONLY) ;
  size = read (fd, buffer, sizeof (buffer) ) ;
  close (fd) ;
  printf ("%s", buffer) ;

}
```

执行结果

```
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

定义函数：#include <unistd.h>

函数说明：ssize_t read（int fd，void *buf，size_t count）；

返 回 值：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，若返回的字节数比所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

请参考 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。

EBADF　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（）。

标准I/O函数

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：无

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

```
/* 将标准输入设备的文件描述词 0，转为可写入的文件指针 */
#include <stdio.h>
main ()
{
  FILE *fp = fdopen (0, "w+") ;
  fprintf (fp, "%s\n", "hello!") ;
  fclose (fp) ;
}
```

执行结果

```
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，参数 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：返回非零值代表已达到文件尾。

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：fopen，fread，fscanf，getc

表头文件：#include <stdio.h>

定义函数：int fgetc（FILE *stream）；

函数说明：fgetc（）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。

返 回 值：fgetc（）会返回读取到的字符，若返回 EOF 则表示到了文件尾。

范 例

```
/*
以 r 形态打开文件 exist，然后将文件内容一个一个字符读出、显示，直到文件结束后关闭文件。
注意：文件 exist 必须存在，不然会有错误产生。
```

```
*/
#include <stdio.h>
main ()
{
 FILE *fp;
 int c;
 fp = fopen ("exist", "r") ;
 while ( (c = fgetc (fp) ) != EOF)
  printf ("%c", c) ;
 fclose (fp) ;
}
```

fgetpos（取得文件流的读取位置）

相关函数：fseek，rewind，ftell，fsetpos

表头文件：#include <stdio.h>

定义函数：int fgetpos（FILE *stream，fpos_t *pos);

函数说明：fgetpos（）和 ftell（）一样都是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。读写位置会利用参数 pos 指针返回。

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。

错误代码：EBADF　参数 stream 非有效或可移动读写位置的文件流。

范 例

请参考 fseek（）。

fgets（由文件中读取一字符串）

相关函数：fopen，fread，fscanf，getc

表头文件：#include <stdio.h>

定义函数：char *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：fgets（）若成功则返回 s 指针，返回 NULL 则表示有有错误发生。

范 例

```
#include <stdio.h>
main ()
{
  char s[80];
  fputs (fgets (s, 80, stdin) , stdout) ;
}
```

执行结果

```
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) ;
}
```

执行结果

```
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零，即该文件内容会消失。若文件不存在则建立该文件。

w+ 打开可读写的文件，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+ 以附加的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

```
/* 以 a+ 形态打开文件 noexist */
#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

```
/* 将小写英文字母 a 到 z，一个字一个字地写入到文件 noexist 中 */
#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。

范 例

```
/* 读取由 fwrite () 范例的执行结果 */
#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);
}
```

执行结果

```
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。若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") ;      /* 打开 /etc/passwd */
 fp = freopen ("/etc/group", "r", fp) ;         /* 改为打开 /etc/group*/
 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）欲将读写位置移到文件开头时：fseek（FILE *stream，0，SEEK_SET）；

（2）欲将读写位置移到文件尾时：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。

错误代码：EBADF 参数 stream 非有效或可移动读写位置的文件流。EINVAL 参 数 whence 非有效值。

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) ;        /* 用 fgetpos () 取得读写位置 */
 printf ("offset = %d\n", pos) ;
```

```
  pos = 10;
  fsetpos (stream, &pos) ;        /* 用 fsetpos () 设置读写位置 */
  printf ("offset = %d\n", ftell (stream) ) ;
  fclose (stream) ;
}
```

执行结果

```
offset = 5      /* fseek (stream, 5, SEEK_SET) ; */
offset = 0      /* rewind (stream) ; */
offset = 10     /* pos = 10, fsetpos (stream, &pos) ; */
```

fsetpos（移动文件流的读写位置）

相关函数：rewind，ftell，fgetpos，fseek

表头文件：#include <stdio.h>

定义函数：int fsetpos（FILE *stream，fpos_t *pos);

函数说明：fsetpos（）和 fseek（）一样都是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 pos 为欲移动到的读写位置。

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。

错误代码：EBADF　参数 stream 非有效或可移动读写位置的文件流。

范 例

请参考 fseek（）。

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

请参考 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) ;
}
```

执行结果

请参考 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

请参考 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（如键盘）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) ;
  }
}
```

执行结果

```
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，fgets

表头文件：#include <stdio.h>

定义函数：char *gets（char *s）;

函数说明：gets（）用来从标准输入设备（如键盘）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（buffer overflow）的安全性问题。建议使用 fgets（）取代。

范 例

请参考 fgets（）。

mktemp（产生唯一的临时文件文件名）

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。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：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) ;
}
```

执行结果

```
/* 每次产生的文件名均不同 */
template = template-ipc7Bj
template = template-o5d4Mp
template = template-uMmYzU
```

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

请参考 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（如屏幕）。

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

请参考 getchar（）。

puts（将指定的字符串写到标准输出设备）

相关函数：fopen，fread，fscanf，fputs

表头文件：#include <stdio.h>

定义函数：int puts（const char *s）;

函数说明：puts（）用来将参数 s 字符串写到标准输出设备（如屏幕）。

返 回 值：puts（）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（）。

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：无

范 例

请参考 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：无

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：无

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：无

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：无

tmpfile（建立临时文件）

相关函数：mktemp mkstemp

表头文件：#include <stdio.h>

定义函数：FILE *tmpfile（void）;

函数说明：tmpfile（）用来建立临时文件文件。临时文件建立的目录以 usr/ include/ stdio.h 定义的 P_tmpdir（/tmp）来决定。该临时文件顺利建立后会以可读写模式打开该文件，并将文件指针返回。

返 回 值：临时文件顺利建立后，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。

错误代码：EACCES 无法建立临时文件。
EEXIST 临时文件已存在。
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。

范 例

```
#include <stdio.h>
main ()
{
 FILE *fp;
 fp = tmpfile () ;
 fclose (fd) ;
}
```

ungetc（将一指定字符写回文件流中）

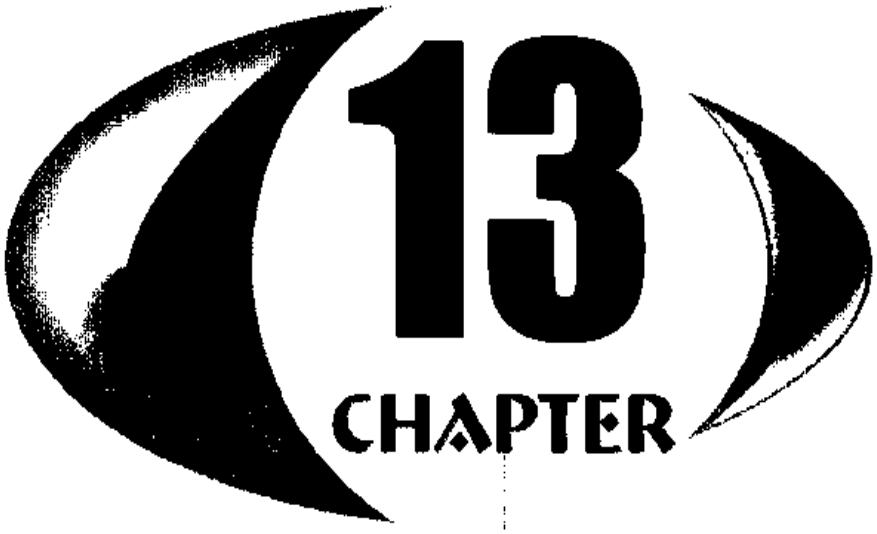
相关函数：fputc，getchar，getc

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。



进程及流程控制

abort（以异常方式结束进程）

相关函数：exit，_exit，assert

表头文件：#include <stdlib.h>

定义函数：void abort（void）;

函数说明：abort（）将引起进程异常的终止，此时所有已打开的文件流会自动关闭，所有的缓冲区数据也会自动写回。

返 回 值：无

assert（若测试的条件不成立则终止进程）

相关函数：abort

表头文件：#include <assert.h>

定义函数：void assert（int expression）;

函数说明：assert（）会判断参数 expression 是否成立，若不成立则会显示错误和排错信息，然后调用 abort（）来终止进程。

返 回 值：无

范 例

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <assert.h>
main ()
{
 assert (open ("/tmp/noexist", O_RDONLY) >=0) ;
 printf ("open ok\n") ;
}
```

```
assert: assert.c: 9: main: Assertion `open ("/tmp/noexist", 00) >=0' failed.
Aborted (core dumped)
```

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) ;
}
```

```
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。

错误代码：请参考 execve（）。

范 例

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

执行结果

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

execle（执行文件）

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

表头文件：#include <unistd.h>

定义函数：int execle（const char *path，const char *arg，……，char* const envp[]）；

函数说明：execle（）用来执行参数 path 字符串所代表的文件路径，然后将第二个以后的参数当作该文件的 argv[0]、argv[1]……，最后一个参数必须指向一新的环境变量数组，此新的环境变量数组即成为新执进程序的环境变量。

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

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

执行结果

```
-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作该文件的 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：请参考 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) ;
}
```

执行结果

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

execv（执行文件）

相关函数：fork，execl，execle，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，第二个参数系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) ;
}
```

执行结果

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

execve（执行文件）

相关函数：fork，execl，execle，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 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) ;
}
```

执行结果

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

execvp（执行文件）

相关函数：fork，execl，execle，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) ;
}
```

执行结果

```
-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：无

范 例

请参考 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，并且会传送 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：无

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（）。

fork（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 fork（void）;

函数说明：fork（）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，因此编写程序时要留意死锁（deadlock）或竞争条件（race condition）的发生。

返 回 值：如果 fork 成功则在父进程会返回新建立的子进程代码（PID），而在新建立的子进程中则返回 0。如果 fork 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 ( fork () == 0 ) {
    printf ("This is the child process\n") ;
  } else {
    printf ("This is the parent process\n") ;
  }
}
```

执行结果

```
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) ) ;
}
```

执行结果

```
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识别码。

范 例

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


```
my gid = 29546
```

getpid（取得进程识别码）

相关函数：fork，kill，getppid

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 () ) ;
}
```


```
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 () ) ;
}
```

执行结果

```
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。最好在调用此函数前先清除 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。

longjmp（跳转到原先 setjmp 保存的堆栈环境）

相关函数：setjmp，siglongjmp，sigsetjmp

表头文件：#include <setjmp.h>

定义函数：void longjmp（jmp_buf env，int val）;

函数说明：longjmp()会还原之前由 setjmp()保存的堆栈环境，然后跳转到 setjmp()之后继续程序的流程。参数 env 为 setjmp（）所保存的堆栈环境，参数 val 是提供 setjmp（）的返回值，此值不可为 0，若为 0 系统会自动以 1 来取代。

返 回 值：无

附加说明：longjmp（）和 siglongjmp（）会令程序不易令人理解，请尽量不要使用。

范 例

```
#include <setjmp.h>
jmp_buf env;
test ()
{
  printf ("Before longjmp () \n") ;
```

```
  longjmp (env, 1111) ; /* 设置 setjmp () 返回值为 1111 */
  printf ("After longjmp () \n") ;
}
main ()
{
  int val;
  int i = 1234;
  if ( (val =setjmp (env) ) != 0) {
    printf ("longjmp call!\n") ;
    printf ("val = %d, i = %d\n", val, i) ;
    return;   /* 从 main () 中返回，结束程序 */
  }
  i = 5678;
  test () ;
}
```

执行结果

```
Before longjmp ()
longjmp call!
val = 1111, i = 5678
```

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。

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) ; /* str 指针将传给 my_exit 的 arg 参数 */
  exit (1234) ; /* 此结束状态 1234 将传给 my_exit 的 status 参数 */
}
```

执行结果

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

ptrace（进程追踪）

相关函数：fork，wait，signal

表头文件：#include <sys/ptrace.h>

定义函数：int ptrace（int request，int pid，int addr，int data）；

函数说明：ptrace（）提供数种服务让父进程来对子进程作追踪。被追踪的子进程处于停止状态时，父进程可以存取子进程的内存空间。参数 request 用来要求系统提供的服务：

PTRACE_TRACEME	此进程将由父进程追踪。
PTRACE_PEEKTEXT	从 pid 子进程的内存地址 addr 中读取一个 word。
PTRACE_PEEKDATA	同 PTRACE_PEEKTEXT。
PTRACE_PEEKUSR	从 pid 子进程的 USER 区域内地址 addr 中读取一个 word。
PTRACE_POKETEXT	将 data 写入 pid 子进程的内存地址 addr 中。
PTRACE_POKEDATA	同 PTRACE_POKETEXT。
PTRACE_POKEUSR	将 data 写入 pid 子进程的 USER 区域内地址 addr 中。
PTRACE_SYSCALL	继续 pid 子进程的执行。
PTRACE_CONT	同 PTRACE_SYSCALL。
PTRACE_SINGLESTEP	设置 pid 子进程的单步追踪旗标。
PTRACE_ATTACH	附带（attach）pid 子进程。
PTRACE_DETACH	分离（detach）pid 子进程。

返 回 值：如果执行成功则返回 0，执行失败则返回 -1，失败原因存于 errno 中。

错误代码：

EPERM	无法追踪较特别的进程（如 init）或指定的进程已处追踪模式。
ESRCH	参数 pid 所指定的进程不存在。
EIO	参数 request 不合法。

范 例

```
/* 用来追踪子进程，拦截 write () 系统调用并显示寄存器值 */
```

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <sys/ptrace.h>
#include <asm/ptrace.h>

#define attach (pid)          ptrace (PTRACE_ATTACH, pid, (char *) 1, 0)
#define detach (pid)          ptrace (PTRACE_DETACH, pid, (char *) 1, 0)
#define trace_sys (pid)       ptrace (PTRACE_SYSCALL, pid, (char *) 1, 0)
#define trace_me ()           ptrace (PTRACE_TRACEME, 0, (char *) 1, 0)
long get_regs (int pid, int reg)
{
  long val;
  val = ptrace (PTRACE_PEEKUSER, pid, (char *) (4*reg) , 0) ;
  if (val == -1)
  perror ("ptrace (PTRACE_PEEKUSER, ...) ") ;
  return val;
}
void print_regs (pid)
{
 struct pt_regs Regs;
 Regs.orig_eax = get_regs (pid, ORIG_EAX) ;
 Regs.eax = get_regs (pid, EAX) ;
 Regs.eip = get_regs (pid, EIP) ;
 if (Regs.orig_eax != 0x4) return;
 printf ("ORIG_EAX = 0x%x , EAX = 0x%x, ", Regs.orig_eax, Regs.eax) ;
 printf ("EIP = 0x%x\n", Regs.eip) ;
}
int trace_syscall (pid)
{
 int value;
 value = trace_sys (pid) ;
 if (value < 0) perror ("ptrace") ;
 return value;
}
main (int argc, char *argv[])
{
```

```
  int pid;
  int p, status;

  if (argc < 2) { printf ("usage: %s program\n", argv[0]) ; return; }
  if ( ! (pid = fork () ) ) {
    if (trace_me () <0) perror ("ptrace") ;
    execl (argv[1], "tracing", 0) ;
  } else {
    printf ("ptrace %s (pid = %d) ...\n", argv[1], pid) ;
    sleep (1) ;
    do {
      trace_syscall (pid) ;
      p = wait (&status) ;
      if (WIFEXITED (status) ) { printf ("child exit () \n") ; exit (1) ; }
      if (WIFSIGNALED (status) ) {
        printf ("child exit () , because a signal\n") ; exit (1) ;
      }
      print_regs (pid) ;
    } while (1) ;
  }
}
```

执行结果

```
[kids@localhost ptrace]$ ./ptrace /bin/uname
ptrace /bin/uname (pid = 6996) ...
ORIG_EAX = 0x4 , EAX = 0xfffffffda, EIP = 0x400bbdf4
Linux
ORIG_EAX = 0x4 , EAX = 0x6, EIP = 0x400bbdf4
child exit ()
/* EIP 值可看出由 glibc 调用系统, ORIG_EAX = 0x4 为 write () 系统
   调用的编号, EAX = 0x6 代表 write () 输出的字符数 */
```

setjmp（保存目前堆栈环境）

相关函数：longjmp，siglongjmp，sigsetjmp

表头文件：#include <setjmp.h>

定义函数：int setjmp（jmp_buf env）;

函数说明：setjmp（）会保存目前堆栈环境，然后将目前的地址做一个记号，而在程序其他地方调用 longjmp（）时便会直接跳越到这个记号位置，然后还原堆栈环境，继续程序执行。参数 env 为用来保存目前堆栈环境，一般声明为全局变量。当 setjmp（）返回 0 时代表已经作好记号，若返回非 0 值代表由 longjmp（）跳转回来，详细情况请参考 longjmp（）。

返 回 值：返回 0 代表已保存好目前堆栈环境，随时可供 longjmp（）调用，若返回非 0 值则代表由 longjmp（）返回。

附加说明：setjmp（）和 sigsetjmp（）会令程序不易令人理解，请尽量不要使用。

范 例

请参考 longjmp（）。

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 指定进程所属的组识别码设为参数 pgid 指定的组识别码。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 pigd 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，而找不到符合的行程。
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。

siglongjmp（跳转到原先 sigsetjmp 保存的堆栈环境）

相关函数：setjmp，longjmp，sigsetjmp

表头文件：#include <setjmp.h>

定义函数：void siglongjmp（sigjmp_buf env，int val）；

函数说明：siglongjmp（）会还原之前由 sigsetjmp（）保存的堆栈环境，然后跳转到 sigsetjmp（）之后继续程序的流程。参数 env 为 sigsetjmp（）所保存的堆栈环境，参数 val 是提供 sigsetjmp（）的返回值，此值不可为 0，若为 0 系统会自动以 1 来取代。

返 回 值：无

附加说明：longjmp()和 siglongjmp()会令程序不易令人理解，请尽量不要使用。

范 例

请参考 longjmp（）。

sigsetjmp（保存目前堆栈环境）

相关函数：longjmp，siglongjmp，setjmp

表头文件：#include <setjmp.h>

定义函数：int sigsetjmp（sigjmp_buf env，int savesigs）；

函数说明：sigsetjmp（）会保存目前堆栈环境，然后将目前的地址作一个记号，而在程序其他地方调用 siglongjmp（）时便会直接跳越到这个记号位置，然后还原堆栈环境，继续程序执行。参数 env 为用来保存目前堆栈环境，一般声明为全局变量。参数 savesigs 若为非 0 值则代表搁置的信号集合也会一块保存。当 sigsetjmp（）返回 0 时代表已经作好记号，若返回非 0 值代表由 siglongjmp（）跳转回来，详细情况请参考 longjmp（）。

返 回 值：返回 0 代表已保存好目前堆栈环境，随时可供 siglongjmp（）调用，若返回非 0 值则代表由 siglongjmp（）返回。

附加说明：setjmp（）和 sigsetjmp（）会令程序不易令人理解，请尽量不要使用。

范 例

请参考 longjmp（）。

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，其他失败原因则返回 -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 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") ;
}
```

执行结果

```
-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>

定义函数：#include <sys/wait.h>

函数说明：pid_t wait（int *status）;

返 回 值：wait（）会暂停目前进程的执行，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块返回。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（）。

附加说明：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

```
#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) ;
   }
}
```

执行结果

```
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（）会暂停目前进程的执行，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块返回。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行程。

pid >0　等待任何子进程识别码为 pid 的子进程。

参数 options 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）　取得子进程由 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。

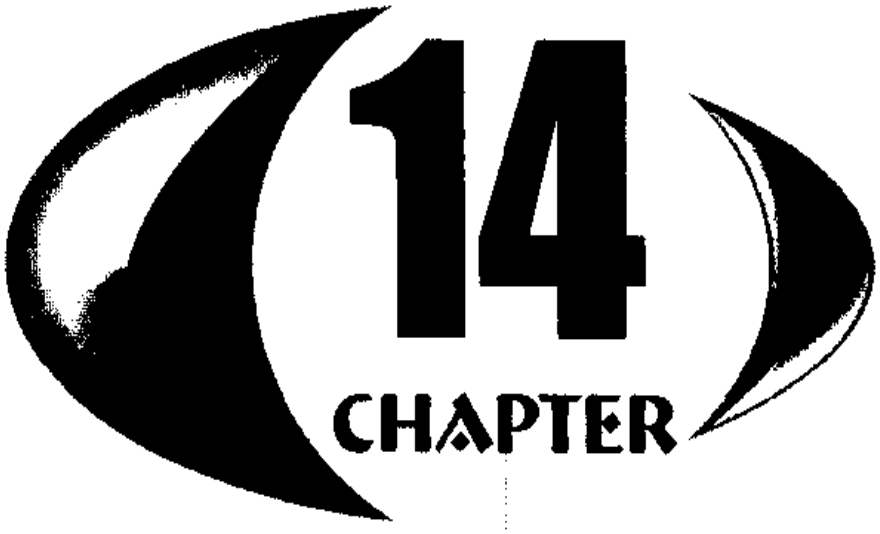
WIFSTOPPED（status）　如果子进程处于暂停执行情况则此宏值为真。一般只有使用 WUNTRACED 时才会有此情况。

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

请参考 wait（）。



格式化输入输出函数

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

```
/* fprintf (stdout, ......) 对标准输出设备 (stdout) 做输出 */
#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) ;
}
```

执行结果

```
-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

```
/* fscanf (stdin, ......) 从标准输入设备 (stdin) 读取数据 */
#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) ;
}
```

执行结果

```
10 0x1b aaaaaaaa bbbbbbb          /* 从键盘输入 */
10 27 aaaaa                       /* 0x1b 经转换成 27，忽略 b 字符串*/
```

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 整型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 整型数的参数会被转成一无符号的十六进制数字，并以大写 ABCD 表示。

✣ **浮点型数：**

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

%e 将 double 型的参数以指数（科学记号）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 dobule 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 和 %g 作用相同，唯一区别在以指数型态打印时会选择%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 * 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：

- 此旗标会将一数值向左对齐。

+ 一般在打印负数时，printf ( ) 会加印一个负号，正数则不加任何符号。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 0。而在类型为 x 之前（如 %#x）则会在打印十六进制数值前多印 '0x'，如果类型为 X 则是多打印 '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，无数字的参数将补上 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 同义 */
}
```

执行结果

```
-100 3.141590 96       /* 96 为 150 的十六进制值 */
150 150
```

scanf（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（unsigned int）。

%u 输入的数据会被转换成一无符号的正整数（unsigned int）。

%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) ;
}
```

执行结果

```
10 0x1b aaaaaaaa bbbbbbb
10 27 aaaaa              /* 0x1b 经转换成 27，忽略 b 字符串*/
```

snprintf（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 snprintf（char *str，size_t size，const char *format，...）；

函数说明：snprintf（）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'）或达到参数 size 指定的大小为止。

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。

范 例

```
#include <stdio.h>
main ()
{
  char a[] = "This is string A!";
  char buf[80];
  snprintf (buf, sizeof (buf) , ">>> %s <<<\n", a) ;
  printf ("%s", buf) ;

}
```

执行结果

```
>>> This is string A! <<<
```

sprintf（格式化字符串复制）

相关函数：printf，snprintf

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（buffer overflow），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) ;

}
```

执行结果

```
>>> 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

```
/* 可参考 scanf () , 这里把标准输出设备改为内定字符串 input[] */
#include <stdio.h>
main ()
{
  int i;
```

```
    unsigned int j;
    char input[] = "10 0x1b aaaaaaaa bbbbbbb";
    char s[5];
    sscanf (input, "%d %x %5[a-z] %*s %f", &i, &j, s, s) ;
    printf ("%d %d %s\n", i, j, s) ;
}
```

执行结果

```
10 27 aaaaa              /* 0x1b 经转换成 27，忽略 b 字符串*/
```

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

请参考 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（）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 fscanf（）及 vprintf（）。

vprintf（格式化输出数据）

相关函数：printf，vfprintf，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) ;
}
```


```
my_printf () : -100 3.141590 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>

定义函数：#include <stdarg.h>

函数说明：int vscanf（const char *format，va_list ap）；

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（）。

vsnprintf（格式化字符串复制）

相关函数：vprintf，printf，sprintf

表头文件：#include <stdio.h>

定义函数：int vsnprintf（char *str，size_t size，const char *format，va_list ap）；

函数说明：vsnprintf（）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'）或达到参数 size 指定的大小为止。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 snprintf（）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 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 sscanf（）及 vprintf（）。

文件及目录函数

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（writable）”，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作文件处理。例如，你会发现 DOS 的文件都具有“可执行（executable）”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（denied）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") ;
}
```


```
/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：请参考 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) ;
   }
 }
```


```
..
.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) ) ;
}
```


```
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 权限来更改更参数 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（owner）或有效用户识别码（effective UID）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_ISVTX 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（如 /tmp ）。

返 回 值：权限改变成功则返回 0，失败回 -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。
EROFS	欲写入权限的文件存在于只读文件系统内。
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，而将该文件的组变改为参数 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 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，或是参数 owner/group 不正确。
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。
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 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：调用成功则返回零，失败返回 -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。
ENOMEM	核心内存不足。
EACCESS	存取目录时被拒绝。
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") ;
   /* 因为根目录已改为 /tmp，故下一行切换到 / 事实上是切换到 /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

请参考 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 所指的文件描述词。

返 回 值：执行成功则返回 0，失败返回 -1，errno 为错误代码。

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) ;
}
```

执行结果

```
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，失败回 -1，错误原因存于 errno。

错误代码：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

```
/* 将 /etc/passwd 改为 S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH 的权限 */
#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，而将该文件的组变改为参数 group 组。如果参数 owner 或 group 为 -1，对映的所有者或组不会有所改变。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

```
/* 将 /etc/passwd 的所有者和组都设为 root */
#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 filedes，struct stat *buf）;

函数说明：fstat（）用来将参数 filedes 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

```
/* 显示 /etc/passwd 文件大小 */
#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) ;
}
```

执行结果

```
/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。

ftw（遍历目录树）

相关函数：opendir

表头文件：#include <ftw.h>

定义函数：int ftw（const char *dir，
int（*fn）（const char *file， const struct stat *sb，int flag），int depth）；

函数说明：ftw（）会从参数 dir 指定的目录开始，往下一层层地递归式遍历子目录。每进入一个目录，便会调用参数 *fn 定义的函数来处理。ftw（）会传三个参数给 fn（），第一个参数 *file 指向当时所在的目录路径，第二个参数是 *sb，为 stat 结构指针（结构定义请参考 stat（）），第三个参数为旗标，有下面几种可能值：

FTW_F	一般文件
FTW_D	目录
FTW_DNR	不可读取的目录。此目录以下将不被遍历。
FTW_SL	符号连接
FTW_NS	无法取得 stat 结构数据，有可能是权限问题。

最后一个参数 depth 代表 ftw（）在进行遍历目录时可同时打开的文件数。ftw（）在遍历时每一层目录至少需要一个文件描述词，如果遍历时用完了 depth 所给予的限制数目，整个遍历将因不断地关文件和开文件操作而显得缓慢。

如果要结束 ftw（）的遍历，fn（）只需返回一非零值即可，此值同时也会是 ftw（）的返回值。否则 ftw（）会试着走完所有的目录，然后返回 0。

返 回 值：遍历中断则返回 fn（）函数的返回值，全部遍历完则返回 0。若有错误发生则返回 -1。

附加说明：由于 ftw（）会动态配置内存使用，请使用正常方式（fn 函数返回非零值）来中断遍历，不要在 fn 函数中使用 longjmp（）。

范 例

```
/* 列出 /etc/X11 目录下的子目录 */
```

```
#include <sys/stat.h>
#include <unistd.h>
#include <ftw.h>
int fn (const char *file, const struct stat *sb, int flag)
{
  if (flag == FTW_D) printf ("%s -- directory\n", file) ;
  return 0;
}
main ()
{
  ftw ("/etc/X11", fn, 500) ;
}
```


```
/etc/X11 -- directory
/etc/X11/wmconfig -- directory
/etc/X11/twm -- directory
/etc/X11/xdm -- directory
/etc/X11/xdm/authdir -- directory
/etc/X11/xsm -- directory
/etc/X11/fs -- directory
/etc/X11/applnk -- directory
/etc/X11/applnk/System -- directory
/etc/X11/applnk/Graphics -- directory
/etc/X11/applnk/Internet -- directory
/etc/X11/gdm -- directory
/etc/X11/gdm/Sessions -- directory
/etc/X11/gdm/Init -- directory
/etc/X11/guichooser -- directory
/etc/X11/xinit -- directory
```

get_current_dir_name（取得当前的工作目录）

相关函数：getcwd，getwd，chdir

表头文件：#include <unistd.h>

定义函数：char *get_current_dir_name（void）;

函数说明：此函数会返回一字符串指针，指向目前的工作目录绝对路径字符串。

返 回 值：执行成功则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 *ptr;
 ptr = get_current_dir_name () ;
 printf ("current working directory : %s\n", ptr) ;
}
```

执行结果

```
current working directory : /tmp
```

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) ;
}
```

执行结果

```
current working directory : /tmp
```

getwd（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 *getwd（char *buf);

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，然后将此路径字符串指针返回。

返 回 值：执行成功则将结果复制到参数 buf 所指的内存空间，然后将指针返回。失败返回 NULL，错误代码存于 errno。

范 例

请参考 getcwd（）。

lchown（改变文件的所有者）

相关函数：chown，fchown，chmod

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

定义函数：int lchown（const char *path，uid_t owner，gid_t group）；

函数说明：lchown（）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。当 root 用 lchown（）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。lchown（）与 chown（）不同处在于，如果参数 path 指定的文件为一符号连接（symbolic link），lchown（）是改变连接本身的所有者或组，chown（）则是改变连接所指向的文件所有者或组。

返 回 值：成功则返回 0，失败回 -1，错误原因存于 errno。

附加说明：

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，或是参数 owner/group 不正确。
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。
ENAMETOOLONG	参数 path 太长。
ENOENT	参数 path 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

请参考 chown（）。

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的文件系统，如有需要请改用 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。
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（symbolic link）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

请参考 stat（）。

nftw（遍历目录树）

相关函数：ftw

表头文件：#include <ftw.h>

定义函数：int nftw（const char *dir，int（*fn）（const char *file，const struct stat *sb，int flag，struct FTW *s），depth，int flags）;

函数说明：nftw（）与 ftw（）很像，都是从参数 dir 指定的目录开始，往下一层层地递归式遍历子目录。每进入一个目录，便会调用参数 *fn 定义的函数来处理。nftw（）会传四个参数给 fn（），第一个参数 *file 指向当时所在的目

录路径，第二个参数是 *sb，为 stat 结构指针（结构定义请参考 stat（）），第三个参数为旗标，有底下几种可能值：

FTW_F	一般文件。
FTW_D	目录。
FTW_DNR	不可读取的目录。此目录以下将不被遍历。
FTW_SL	符号连接。
FTW_NS	无法取得 stat 结构数据，有可能是权限问题。
FTW_DP	目录，而且其子目录都已被遍历过了。
FTW_SLN	符号连接，但连接不存在的文件。

fn（）的第四个参数是 FTW 结构，定义如下：

```
struct FTW
{
int base;
int level;
};
```

level 代表遍历当时的深度，nftw（）第三个参数 depth 代表 nftw（）在进行遍历目录时可同时打开的文件数。ftw（）在遍历时每一层目录至少需要一个文件描述词，如果遍历时用完了 depth 所给予的限制数目，整个遍历将因不断地关文件和开文件操作而显得缓慢。

nftw（）最后一个参数 flags 用来指定遍历时的动作，可以指定下列的操作或用 OR 组合：

FTW_CHDIR	在读目录之前先用 chdir（）移到此目录。
FTW_DEPTH	执行深度优先搜索（depth-first search）。在遍历此目录前先将所有子目录遍历完。
FTW_MOUNT	遍历时不要跨越到其他文件系统。
FTW_PHYS	不要遍历符号连接的目录。预设会遍历符号连接目录。

如果要结束 nftw（）的遍历，fn（）只需返回一非 0 值即可，此值同时也会是 nftw（）的返回值。否则 nftw（）会试着遍历完所有的目录，然后返回 0。

返 回 值：遍历中断则返回 fn（）函数的返回值，全部遍历完则返回 0。若有错误发生则返回 -1。

附加说明：请参考 ftw（）。

范 例

请参考 ftw（）。

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。

范 例

请参考 readdir（）。

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;
        har 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	_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

```
/* 读取 /etc/rc.d 目录文件结构，然后显示该目录下的文件 */

#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) ;
}
```

执行结果

```
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	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。

realpath（将相对目录路径转换成绝对路径）

相关函数： readlink，getcwd

表头文件： #include <limits.h>

#include <stdlib.h>

定义函数： char *realpath（const char *path，char *resolved_path）；

函数说明： realpath（）用来将参数 path 所指的相对路径转换成绝对路径后存于参数 resolved_path 所指的字符串数组中。

返 回 值： 如果转换成功则返回指向 resolved_path 的指针。失败返回 NULL，错误代码存于 errno。

范 例

```
#include <unistd.h>
main ()
{
  char resolved_path[80];
  realpath ("/usr/X11R6/lib/modules/../../include/../", resolved_path) ;
  printf ("resolved_path : %s\n", resolved_path) ;
}
```

执行结果

```
resolved_path : /usr/X11R6
```

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	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 所指定的文件已经存在，则会被删除。

返 回 值：执行成功则返回 0，失败返回 -1，errno 为错误代码。

范 例

```
/* 设计一个 DOS 下的 rename (ren) 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 ", argv[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：无

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) ;        /* 倒带，从头读取 */
    printf ("Readdir again!\n") ;
    while ( (ptr = readdir (dir) ) != NULL)
   {
      printf ("d_name : %s\n", ptr->d_name) ;
   }
   closedir (dir) ;
}
```

执行结果

```
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
```

scandir（读取特定的目录数据）

相关函数：opendir，readdir，alphasort

表头文件：#include <dirent.h>

定义函数：int scandir（const char *dir，struct dirent ***namelist，
nt（*select）（const struct dirent *），
nt（*compar）（const struct dirent **，const struct dirent**））；

函数说明：scandir（）会扫描参数 dir 指定的目录文件，经由参数 select 指定的函数来挑选目录结构至参数 namelist 数组中，最后再调用参数 compar 指定的函数来排序 namelist 数组中的目录数据。每次从目录文件中读取一个目录结构后便将此结构传给参数 select 所指的函数，select 函数若不想要将此目录结构复制到 namelist 数组就返回 0，若 select 为空指针则代表选择所有的目录结构。scandir（）会调用 qsort（）来排序数据，参数 compar 则为 qsort（）的参数，若是要排列目录名称字母则可使用 alphasort（）。结构 dirent 定义请参考 readdir（）。

返 回 值：成功则返回复制到 namelist 数组中的数据结构数目，有错误发生则返回 -1。

错误代码：ENOMEM　核心内存不足。

范 例

```
/* 读取 / 目录下文件名长度大于 5 的目录结构 */

#include <dirent.h>
int select (const struct dirent *dir)
{
  if (strlen (dir->d_name) > 5)
    return 1;
  else
    return 0;
}
main ()
{
  struct dirent **namelist;
  int i, total;
  total = scandir ("/", &namelist, select, 0) ;
  if (total < 0)
    perror ("scandir") ;
    else {
    for (i = 0; i < total; i++)
      printf ("%s\n", namelist[i]->d_name) ;
      printf ("total = %d\n", total) ;
    }
}
```

执行结果

```
.gnome
.gnome_private
WebLog
ErrorLog
total = 5
```

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值。

返 回 值：无

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;     /* 记录第 5 个偏移值 */
  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) ;
}
```

执行结果

```
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!                  /* 从第 5 个位置开始重读 */
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（struct stat）。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（if inodedevice）*/
    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 last access */
    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（hard link）数目，刚建立的文件值为 1。

st_uid　文件所有者的用户识别码（user ID）。

st_gid　文件所有者的组识别码（group ID）。

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（symbolic link）
S_IFREG	0100000	一般文件
S_IFBLK	0060000	区块装置（block device）
S_IFDIR	0040000	目录
S_IFCHR	0020000	字符装置（character device）
S_IFIFO	0010000	先进先出（fifo）
S_ISUID	0004000	文件的（set user-id on execution）位
S_ISGID	0002000	文件的（set group-id on execution）位
S_ISVTX	0001000	文件的 sticky 位
S_IRWXU	00700	文件所有者的遮罩值（即所有权限值）
S_IRUSR	00400	文件所有者具可读取权限
S_IWUSR	00200	文件所有者具可写入权限
S_IXUSR	00100	文件所有者具可执行权限
S_IRWXG	00070	用户组的遮罩值（即所有权限值）
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_IRWXO	00007	其他用户的遮罩值（即所有权限值）
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（symbloic link）?
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）	否为先进先出（fifo）?
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（delete）或改名（rename）。

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

```
/* 显示 /etc/passwd 文件大小 */
#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) ;
}
```

执行结果

```
/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连立连接。

返 回 值：成功则返回 0，失败回 -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

```
/* 将 /etc/passwd 作一符号连接 pass */
#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值。

返 回 值：返回下个读取位置，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) ;
}
```

执行结果

```
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（symbolic link），则此连接会被删除。

返 回 值：成功则返回 0，失败回 -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;  /* 存取时间 */
    time_t modtime; /* 更改时间 */
};
```

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。

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：

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：

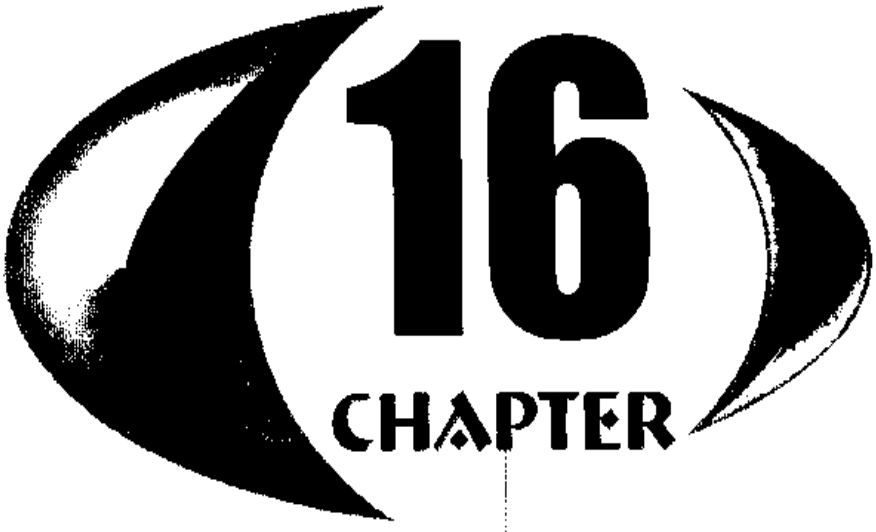
```
struct timeval {
    long    tv_sec;    /* 秒 */
    long    tv_usec;   /* 微秒 */
};
```

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。

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：EACCESS　　存取文件时被拒绝，权限不足。

ENOENT　　指定的文件不存在。



信号函数

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

```
/* 程序执行 5 秒后打印出 hello 字符串*/
#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) ;
   }
}
```

执行结果

```
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	将信号像广播般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 绝对值的所有行程。

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的进程。

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 ("child process receive signal %d\n", WTERMSIG (status) ) ;
  }
}
```

执行结果

```
send 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。

psignal（列出信号描述和指定字符串）

相关函数：strsignal

表头文件：#include <signal.h>

定义函数：void psignal（int sig，const char *s）;

函数说明：psignal（）会由全域变量 sys_siglist[] 列表中查找符合参数 sig 的信号编号，然后将参数 s 指定的字符串加上信号描述后输出至标准错误（stderr）。

返 回 值：无

范 例

```
/* 列出前 9 个信号描述 */
#include <signal.h>
main ()
{
  int sig;
  char s[40];
  for (sig = 1; sig < 10; sig++) {
    sprintf (s, "signal %d", sig) ;
    psignal (sig, s) ;
  }
}
```

执行结果

```
signal 1: Hangup
signal 2: Interrupt
signal 3: Quit
signal 4: Illegal instruction
signal 5: Trace/breakpoint trap
signal 6: Aborted
signal 7: Bus error
signal 8: Floating point exception
signal 9: Killed
```

raise（传送信号给目前的进程）

相关函数：kill，signal

表头文件：#include <signal.h>

定义函数：int raise（int sig）；

函数说明：raise（）用来将参数 sig 指定的信号传送给目前的进程。相当于 kill（getpid（），sig）；

返 回 值：执行成功则返回 0，否则返回非 0 值。

范 例

```
#include <unistd.h>
#include <signal.h>
void handler (}
{
  printf ("hello\n") ;
}
main (}
{
  signal (SIGUSR1, handler) ;
  raise (SIGUSR1) ;
}
```

执行结果

```
hello
```

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。如果参数 act 不是 NULL 指针，则用来设置新的信号处理方式。

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 signum 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新启动。

SA_NOMASK/SA_NODEFER：在处理此信号未结束前不理会此信号的再次到来。

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 signum 不合法，或是企图拦截 SIGKILL/SIGSTOP 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) ;
      show_handler (&oldact) ;
  }
}
```

执行结果

```
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：sigemptyset，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。

错误代码：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：sigismembert（）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）。

注意事项：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

请参考 alarm（）或 raise（）。

sigpause（暂停直到信号到来）

相关函数：sigsupend

表头文件：#include <signal.h>

定义函数：int sigpause（int sigmask）;

函数说明：sigpause（）会将目前的信号遮罩暂时换成参数 sigmask 所指定的信号遮罩，并将此进程暂停直到有信号到来才恢复原来的信号遮罩，继续程序执行。此函数已由 sigsuspend（）取代。

返 回 值：不论执行成功或失败都返回 -1，执行成功则 errno 会设成 EINTR。

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（）会将被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。

sigsuspend（暂停直到信号到来）

相关函数：signal，sigaction，sigprocmask，sigpending

表头文件：#include <signal.h>

定义函数：int sigsuspend（const sigset_t *mask）；

函数说明：sigsuspend（）会将目前的信号遮罩暂时换成参数 mask 所指定的信号遮罩，并将此进程暂停直到有信号到来才恢复原来的信号遮罩，继续程序执行。

返 回 值：不论执行成功或失败都返回 -1，执行成功则 errno 会设成 EINTR。

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（进入睡眠状态），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 secones 所指定的时间则返回 0，若有信号中断则返回剩余秒数。

isdigit（测试字符是否为阿拉伯数字）

相关函数：strsignal（由信号编号取得信号描述）

表头文件：psignal

#include <string.h>

定义函数：char *strsignal（int sig);

函数说明：strsignal（）会由全局变量 sys_siglist[] 列表中查找符合参数 sig 的信号编号，然后将此信号的描述由字符串指针返回。

返 回 值：返回描述信号的字符串指针。

范 例

```
/* 列出前 9 个信号描述 */
#include <string.h>
main ()
{
  int signum;
  for (signum = 1; signum < 10; signum++)
    printf ("%-2d : %s\n", signum, strsignal (signum) ) ;

}
```

执行结果

```
1  : Hangup
```

```
2 : Interrupt
3 : Quit
4 : Illegal instruction
5 : Trace/breakpoint trap
6 : Aborted
7 : Bus error
8 : Floating point exception
9 : Killed
```

错误处理函数

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，后面再加上错误的原因字符串。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：无

范 例

```
#include <stdio.h>
main ()
{
  FILE *fp;
  fp = fopen ("/tmp/noexist", "r+") ;
  if (fp == NULL) perror ("fopen") ;
}
```

执行结果

```
$ ./perror
fopen: No such file or directory
```

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。

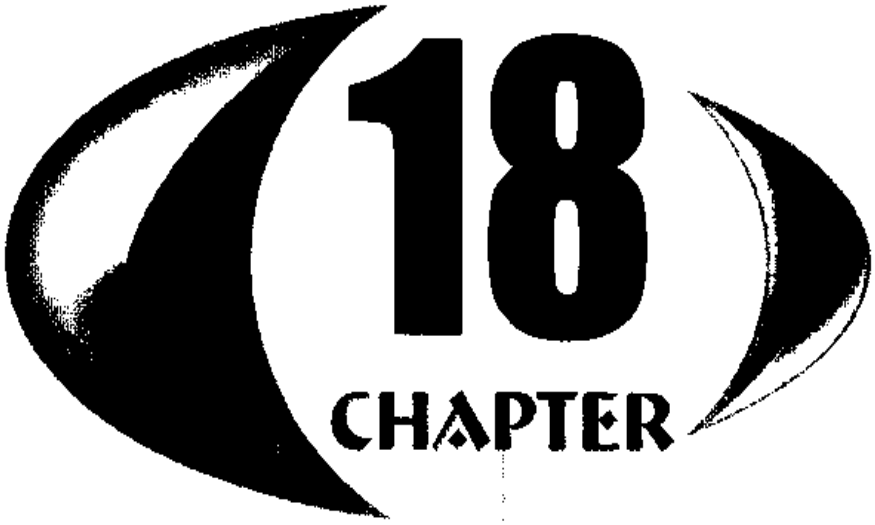
范 例

```
/* 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) ) ;
}
```

执行结果

```
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
```



管道相关函数

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该返回，但是若还没有其他进程打开 FIFO 文件案来读取，则写入的操作会返回 ENXIO 错误代码。
2. 没有使用 O_NONBLOCK 旗标时（一般情况下），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：

EACCES	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

```
/* 与 pipe () 范例作用相同 */
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO   "/tmp/fifo"
main ()
{
   char buffer[80];
   int fd;
   unlink (FIFO) ;          /* 删除 FIFO 文件 */
   mkfifo (FIFO, 0666) ;    /* 建立 FIFO 文件 */
   if (fork () > 0) {
      /* 父进程 */
      char s[] = "Hello!\n";
      fd = open (FIFO, O_WRONLY) ;   /* 打开 FIFO 文件   */
      write (fd, s, sizeof (s) ) ;   /* 将字符串写入该文件 */
      close (fd) ;
   }
  else {
      /* 子进程 */
      fd = open (FIFO, O_RDONLY) ;   /* 打开 FIFO 文件   */
      read (fd, buffer, 80) ;        /* 从 FIFO 文件读取字符串 */
      printf ("%s", buffer) ;
      close (fd) ;
   }
}
```

执行结果

```
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

请参考 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 filedes[2]）;

函数说明：pipe（）会建立管道，并将文件描述词由参数 filedes 数组返回。filedes[0] 为管道里的读取端，filedes[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 filedes 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 filedes[2];
```

```
  char buffer[80];
  pipe (filedes) ;
  if (fork () > 0) {
    /* 父进程 */
    char s[] = "Hello!\n";
    write (filedes[1], s, sizeof (s) ) ; /* 将字符串写至管道 */
  }
  else {
    /* 子进程 */
    read (filedes[0], buffer, 80) ; /* 从管道读取字符串 */
    printf ("%s", buffer) ;
  }
}
```


```
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用使用 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") ; /* 执行 cat /etc/passwd 命令 */
  fgets (buffer, sizeof (buffer) , fp) ; /* 读取一行 cat 的输出 */
  printf ("%s", buffer) ;
  pclose (fp) ;
}
```

执行结果

```
/* 显示 /etc/passwd 第一行 */
root: x: 0: 0: root, /root: /bin/bash
```

Socket 相关函数

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（firewall）拒绝此连线。
ENOBUFS	系统的缓冲内存不足。
ENOMEM	核心内存不足。

范 例

请参考 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 sockaddr 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

请参考 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    /* 使用的 port 号码 */
#define SERVER_IP     "127.0.0.1"/* server 的 IP     */
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) ) ;
    /* 将字符串传送给 server 端 */
    if (send (s, buffer, sizeof (buffer) , 0) < 0) {
      perror ("send") ;
      exit (1) ;
    }
  }
}
```

执行结果

```
$ ./connect
Welcome to server!      /* server 端传来的欢迎字符串 */
hi I am client!     /* 从键盘输入字符串 */
/* <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（）所打开的文件。

返 回 值：无

范 例

请参考 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：无

范 例

请参考 getservent（）。

gethostbyaddr（由 IP 地址取得网络数据）

相关函数：gethostbyname

表头文件：#include <netdb.h>

定义函数：struct hostent *gethostbyaddr（const char *addr，int len，int type）;

函数说明：gethostbyaddr（）会返回一个 hostent 结构，参数 addr 可以为 IPv4 或 IPv6 的 IP 地址，参数 len 为参数 addr 的长度，参数 type 为 AF_INET。结构 hostent 定义请参考 gethostbyname（）。

返 回 值：成功则返回 hostent 结构指针，若有错误则返回 NULL 指针。错误原因则存于 h_errno 变量。

错误代码：

HOST_NOT_FOUND	找不到指定的主机。
NO_ADDRESS	该主机有名称却无 IP 地址。
NO_RECOVERY	域名服务器有错误发生。
TRY_AGAIN	请再调用一次。

范 例

```
#include <netdb.h>
#include <sys/socket.h>
main (int argc, char *argv[])
{
```

```
  struct hostent *host;
  if (argc < 2) return;
  host = gethostbyaddr (argv[1], sizeof (argv[1]) , AF_INET) ;
  if (host == (struct hostent *) NULL) herror ("gethostbyaddr") ;
  else {
    printf ("name  : %s\n", host->h_name) ;
    printf ("type  : %d\n", host->h_addrtype) ;
    printf ("addr  : %s\n", host->h_addr_list[0]) ;
  }
}
```

gethostbyname（由主机名称取得网络数据）

相关函数：gethostbyaddr，sethostent

表头文件：#include <netdb.h>

定义函数：struct hostent *gethostbyname（const char *name）;

函数说明：gethostbyname（）会返回一个 hostent 结构，参数 name 可以为一个主机名称或 IPv4/IPv6 的 IP 地址。结构 hostent 定义如下：

```
struct hostent
{
  char *h_name;
  char **h_aliases;
  int h_addrtype;
  int h_length;
  char **h_addr_list;
};
```

各参数定义如下：

h_name	正式的主机名称。
h_aliases	指向主机名称的其他别名。
h_addrtype	地址的型态，通常是 AF_INET。
h_length	地址的长度。
h_addr_list	从域名服务器取得该主机的所有地址。

返 回 值：成功则返回 hostent 结构指针，若有错误则返回 NULL 指针。错误原因则存于 h_errno 变量。

错误代码：

HOST_NOT_FOUND	找不到指定的主机。
NO_ADDRESS	该主机有名称却无 IP 地址。
NO_RECOVERY	域名服务器有错误发生。
TRY_AGAIN	请再调用一次。

范 例

```
/* 利用 gethostbyname () 的简单反查 IP 程序 */
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int main (int argc, char *argv[])
{
  struct hostent *hp;
  struct in_addr in;
  struct sockaddr_in local_addr;
  if (argc < 2) return;
  if (! (hp=gethostbyname (argv[1]) ) )  {
    fprintf (stderr, "Can't resolve host.\n") ;
    exit (1) ;
  }
  memcpy (&local_addr.sin_addr.s_addr, hp->h_addr, 4) ;
  in.s_addr = local_addr.sin_addr.s_addr;
  printf ("Domain Name: %s\n", argv[1]) ;
  printf ("IP address : %s\n", inet_ntoa (in) ) ;
}
```

执行结果

```
$ ./nslookup linux.ee.tku.edu.tw
Domain Name: linux.ee.tku.edu.tw
IP address : 163.13.132.68
```

getprotobyname（由网络协议名称取得协议数据）

相关函数：getprotobynumber，getprotoent，setprotoent，endprotoent

表头文件：#include <netdb.h>

定义函数：struct protoent *getprotobyname（const char *name）；

函数说明：getprotobyname（）会返回一个 protoent 结构，参数 name 为欲查询的网络协议名称。此函数会从 /etc/protocols 中查找符合条件的数据并由结构 protoent 返回。结构 protoent 定义请参考 getprotoent（）。

返 回 值：成功则返回 protoent 结构指针，若有错误或找不到符合的数据则返回 NULL 指针。

范 例

```
/* 取得 icmp 协议数据 */
#include <netdb.h>
main ()
{
  struct protoent *protocol;
  protocol = getprotobyname ("icmp") ;
  printf ("protocol name   : %s\n", protocol->p_name) ;
  printf ("protocol number : %d\n", protocol->p_proto) ;
  printf ("protocol alias  : %s\n", protocol->p_aliases[0]) ;
}
```

getprotobynumber（由网络协议编号取得协议数据）

相关函数：getprotobyname，getprotoent，setprotoent，endprotoent

表头文件：#include <netdb.h>

定义函数：struct protoent *getprotobynumber（int proto）；

函数说明：getprotobynumber（）会返回一个 protoent 结构，参数 proto 为欲查询的网络协议编号。此函数会从 /etc/protocols 中查找符合条件的数据并由结构 protoent 返回。结构 protoent 定义请参考 getprotoent（）。

返 回 值：成功则返回 protoent 结构指针，若有错误或找不到符合的数据则返回 NULL 指针。

范 例

```
/* 取得协议编号 0 至 4 的协议数据 */
#include <netdb.h>
main ()
{
  int number;
  struct protoent *protocol;
  for (number = 0 ; number < 5; number++)
  {
    protocol = getprotobynumber (number) ;
    if (protocol == (struct protoent *) NULL) continue;
    printf ("%2d : %-10s : %-10s\n",
    protocol->p_proto, protocol->p_name, protocol->p_aliases[0]) ;
  }
}
```

执行结果

```
 0 : ip         : IP
 1 : icmp       : ICMP
 2 : igmp       : IGMP
 3 : ggp        : GGP
 4 : ipencap    : IP-ENCAP
```

getprotoent（取得网络协议数据）

相关函数：getprotobyname，getprotobynumber，setprotoent，endprotoent

表头文件：#include <netdb.h>

定义函数：struct protoent *getprotoent（void）;

函数说明：getprotoent（）会打开/etc/protocols，然后读取一行数据后由结构 protoent 返回。之后再调用此函数则会继续读取下一项数据，除非已到文件尾时返回 NULL 指针。

结构 protoent 定义如下：

```
struct protoent
{
    char    *p_name;        /* official protocol name */
    char    **p_aliases;    /* alias list */
    int     p_proto;        /* protocol number */
};
```

各参数定义如下：

p_name	正式的协议名称。
p_aliases	指向协议名称的其他别名。
p_proto	协议编号。

返 回 值：成功则返回 protoent 结构指针，若有错误或找不到符合的数据则返回 NULL 指针。

范 例

```
/* 取得所有协议数据 */
#include <netdb.h>
main ()
{
  struct protoent *p;
  while (p = getprotoent () )
    printf ("%s  %s  %d\n", p->p_name, p->p_aliases[0], p->p_proto) ;
  endprotoent () ;
}
```

执行结果

```
ip          IP          0
icmp        ICMP        1
igmp        IGMP        2
```

```
ggp         GGP         3
ipencap     IP-ENCAP    4
st          ST          5
tcp         TCP         6
egp         EGP         8
pup         PUP         12
udp         UDP         17
```

getservbyname（依名称取得网络服务的数据）

相关函数：getservent，getservbyport，setservent，endservent

表头文件：#include <netdb.h>

定义函数：struct servent *getservbyname（const char *name，const char *proto）；

函数说明：getservbyname（）会返回一个 servent 结构，参数 name 可以为一个网络服务的名称，参数 proto 则为该服务所使用的协议。此函数会从 /etc/services 中查找符合条件的数据并由结构 servent 返回。结构 servent 的定义请参考 getservent（）。

返 回 值：成功则返回 servent 结构指针，若有错误则返回 NULL 指针。

范 例

```
#include <netdb.h>
main ()
{
  struct servent *s;
  s = getservbyname ("telnet", "tcp") ;
 printf ("%s    %d/%s\n", s->s_name, ntohs (s->s_port) , s->s_proto) ;
}
```

执行结果

```
telnet  23/tcp
```

getservbyport（依 port 号码取得网络服务的数据）

相关函数：getservent，getservbyname，setservent，endservent

表头文件：#include <netdb.h>

定义函数：struct servent *getservbyport（int port，const char *proto）；

函数说明：getservbyport（）会返回一个 servent 结构，参数 port 可以为一个 port 号码，参数 proto 则为该服务所使用的协议。此函数会从 /etc/services 中查找符合条件的数据，并由结构 servent 返回。结构 servent 的定义请参考 getservent（）。

附加说明：参数 port 必须先由 htons（）转换。

返 回 值：成功则返回 servent 结构指针，若有错误则返回 NULL 指针。

范 例

```
#include <netdb.h>
main ()
{
  struct servent *s;
  s = getservbyport (htons (23) , "tcp") ;
 printf ("%s    %d/%s\n", s->s_name, ntohs (s->s_port) , s->s_proto) ;
}
```

执行结果

```
telnet  23/tcp
```

getservent（取得主机网络服务的数据）

相关函数：getservbyname，getservbyport，setservent，endservent

表头文件：#include <netdb.h>

定义函数：struct servent *getservent（void）;

函数说明：getservent（）会打开 /etc/services，然后读取一行数据后由结构 servent 返回。之后再调用此函数则会继续读取下一项数据，除非已到文件尾时返回 NULL 指针。

结构 servent 定义如下：

```
struct servent
{
char    *s_name;     /* 正式的服务名称 */
char    **s_aliases;    /* 别名列表 */
int     s_port; /* 所使用的 port 号码 */
char    *s_proto;    /* 使用的协议名称 */
};
```

返 回 值：成功则返回 servent 结构指针，若有错误或读到了文件尾则返回 NULL 指针。

范 例

```
/* 列出主机所有网络服务的数据 */
#include <netdb.h>
main ()
{
  struct servent *s;
  while ( (s = getservent () ) )
    printf ("%s  %d/%s\n", s->s_name, ntohs (s->s_port) , s->s_proto) ;
  endservent () ;
}
```

执行结果

```
tcpmux  1/tcp
echo    7/tcp
echo    7/udp
discard 9/tcp
```

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) ;
}
```


```
optval = 1 /* SOCK_STREAM 的定义正是此值 */
```

herror（打印出网络错误原因信息字符串）

相关函数：hstrerror

表头文件：#include <netdb.h>

定义函数：void herror（const char *s）;

函数说明：herror（）用来将上一个网络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，后面再加上错误的原因字符串。此错误原因系依照全局变量 h_errno 的值来决定要输出的字符串。

返 回 值：无

范 例

请参考 perror（）。

hstrerror（返回网络错误原因的描述字符串）

相关函数：herror

表头文件：#include <netdb.h>

定义函数：const char *hstrerror（int err）;

函数说明：hstrerror（）用来依参数 err 的错误代码来查询 socket 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 5 的错误原因描述 */
#include <netdb.h>
main ()
{
  int i;
  for (i=0;i<6;i++)
    printf ("%d : %s\n", i, hstrerror (i) ) ;
}
```

执行结果

```
0: Resolver Error 0 (no error)
1: Unknown host
2: Host name lookup failure
3: Unknown server error
4: No address associated with name
5: Unknown resolver error
```

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

请参考 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

请参考 getservbyport（）或 connect（）。

inet_addr（将网络地址转成网络二进制的数字）

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 所指的网络地址字符串转换成网络所使用的二进制的数字。网络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络二进制的数字，失败返回 -1。

inet_aton（将网络地址转成网络二进制的数字）

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_aton（const char *cp，struct in_addr *inp）；

函数说明：inet_aton（）用来将参数 cp 所指的网络地址字符串转换成网络所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;
};
```

网络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：成功则返回非 0 值，失败则返回 0。

inet_ntoa（将网络二进制的数字转换成网络地址）

相关函数：inet_addr，inet_aton

表头文件：#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

定义函数：char *inet_ntoa（struct in_addr in）；

函数说明：inet_ntoa（）用来将参数 in 所指的网络二进制的数字转换成网络地址，然后将指向此网络地址字符串的指针返回。

结构 in_addr 定义如下：

```
struct in_addr
{
   unsigned long int s_addr;
};
```

网络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：成功则返回字符串指针，失败则返回 NULL。

范 例

请参考 connect ()。

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 client 端将收到 ECONNREFUSED 的错误（请参考 connect ()）。listen () 并未开始接受连线，只是设置 socket 为 listen 模式，真正接受 client 端连线的是 accept()。通常 listen()会在 socket ()、bind () 之后调用，接着才调用 accept ()。

附加说明：listen () 只适用 SOCK_STREAM 或 SOCK_SEQPACKET 的 socket 类型。如果 scoket 为 AF_INET 则参数 backlog 最大值可设至 128。

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。
EOPNOTSUPP　指定的 socket 并未支援 listen 模式。

范 例

```
/* 利用 socket 的 TCP server
   此程序会接收由 TCP client 传来的字符串并显示.
   TCP client 范例请参考 connect () .
*/
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#define PORT          1234 /* 使用的 port 号码 */
#define MAXSOCKFD 10   /* 可同时服务的最大连线数目 */
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) ;
  }
  /* 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 = htonl (INADDR_ANY) ;
  if (bind (sockfd, &addr, sizeof (addr) ) < 0) {
    perror ("connect") ;
    exit (1) ;
  }
  if (listen (sockfd, 3) <0) {
    perror ("listen") ;
    exit (1) ;
  }
  /* 清除连线状态旗标 */
  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;
    /* 判断是否有新连线或新信息进来 */
   for (fd = 0; fd < MAXSOCKFD; fd++)
    if (FD_ISSET (fd, &readfds) ) {
      if (sockfd == fd) {
        /* 接收新连线 */
        if ( (newsockfd = accept (sockfd, &addr, &addr_len) ) < 0)
                        perror ("accept") ;
  /* 将欢迎字符串传送给 client 端 */
        write (newsockfd, msg, sizeof (msg) ) ;
        is_connected[newsockfd] = 1;
        printf ("Connect from %s\n", inet_ntoa (addr.sin_addr) ) ;
      } else {
        /* 接收新信息 */
        bzero (buffer, sizeof (buffer) ) ;
        if (read (fd, buffer, sizeof (buffer) ) <= 0) {
          /* 连线已中断，清除连线状态旗标 */
          printf ("Connection closed.\n") ;
          is_connected[fd] = 0;
          close (fd) ;
        } else
          printf ("%s", buffer) ;
      }
    }
  }
}
```

执行结果

```
(请执行此 tcp server 范例后再执行 tcp client)
$ ./listen
Connect from 127.0.0.1      /* 接收到来自 127.0.0.1 的连线 */
hi I am client!             /* 收到 client 端传来的字符串 */
Connection closed.          /* 连线中断 */
```

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

请参考 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：返回对应的主机字符顺序。

范 例

请参考 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	强迫接受到 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

请参考 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

```
/* 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送给 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 /* 使用的 port 号码 */
#define SERVER_IP     "127.0.0.1" /* server 的 IP */
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 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) ;
  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) ;
  }
```

```
}
```

执行结果

```
(先执行 udp server 再执行此 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。
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

请参考 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 为已建立好连线的 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（routing-table）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 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

请参考 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 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。
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

请参考 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 指向欲连线的数据内容，参数 len 则为数据长度。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 的结构长度。

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。
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

```
/*
  利用 socket 的 UDP server.
  此程序会接收由 UDP client 传来的字符串然后再传送回 client。
  TCP client 范例请参考 recvfrom ().
*/
#include <sys/types.h>
#include <sys/socket.h>
```

```
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#define PORT    2345    /* 使用的 port 号码 */
main ()
{
  int sockfd, len;
  struct sockaddr_in addr;
  int addr_len = sizeof (struct sockaddr_in) ;
  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.s_addr = htonl (INADDR_ANY) ;
  if (bind (sockfd, &addr, sizeof (addr) ) < 0) {
    perror ("connect") ;
    exit (1) ;
  }
  while (1) {
    /* 接收 client 端传来的字符串 */
    bzero (buffer, sizeof (buffer) ) ;
    len = recvfrom (sockfd, buffer, sizeof (buffer) , 0, &addr, &addr_len) ;
    /* 显示 client 端的网络地址 (IP) */
    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) ;
  }
}
```

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（）将不会自动关闭此文件。

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_t optlen);

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 型态（如：SOCK_STREAM）。
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（out-of-band）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

请参考 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 进程通讯协议
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（out-of-band）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。

参数 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。

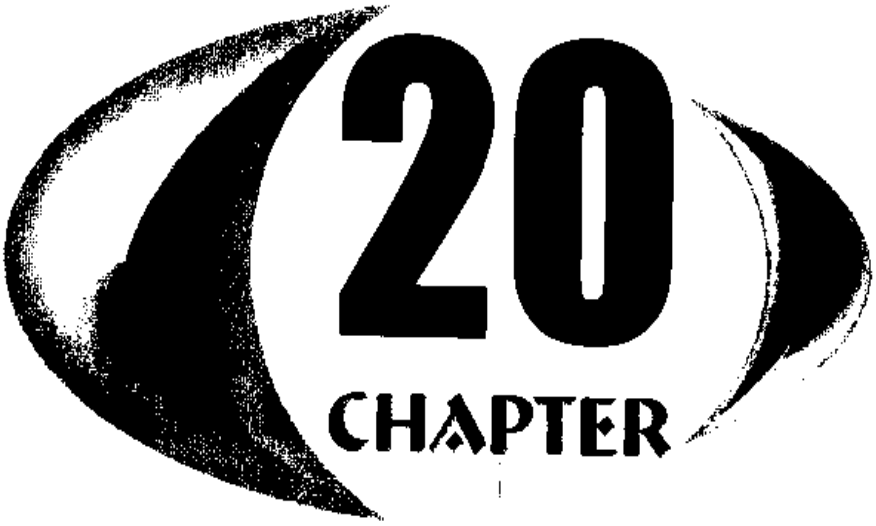
返 回 值：成功则返回 socket 处理代码，失败返回 -1。

错误代码：

EPROTONOSUPPORT	参数 domain 指定的类型不支援参数 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，无法建立参数 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

请参考 connect（）。



进程通信（IPC）函数

ftok（将文件路径和计划代号转为 System V IPC key）

相关函数：msgget，semget，shmget

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

定义函数：key_t ftok（char *pathname，char proj）；

函数说明：ftok（）用来将参数 pathname 指定的文件和计划代号（project ID），转换为 System V IPC 函数所需使用的 key。参数 pathname 指定的文件必须是存在且可以存取。

返 回 值：若成功则返回 key_t 值，否则返回 -1，错误原因存于 errno 中。

错误代码：

ENOENT	参数 pathname 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 个符号连接。
EACCESS	存取文件时被拒绝。
ENOMEM	核心内存不足。
ENAMETOOLONG	参数 pathname 的路径名称太长。

范 例

```
key_t key = ftok ("/tmp/sysvipc", 1) ;
```

msgctl（控制信息队列的运作）

相关函数：msgget，msgsnd，msgrcv

表头文件：#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

定义函数：int msgctl（int msqid，int cmd，struct msqid_ds *buf）；

函数说明：msgctl（）提供了几种方式来控制信息队列的运作。参数 msgid 为欲处理的信息队列识别代码，参数 cmd 为欲控制的操作，有下列几种数值：

IPC_STAT　把信息队列的 msqid_ds 结构数据复制到参数 buf。

IPC_SET　将参数 buf 所指的 msqid_ds 结构中的 msg_perm.uid、msg_perm.gid、msg_perm.mode 和 msg_qbytes 参数复制到信息队列的 msqid_ds 结构内。

IPC_RMID　删除信息队列和其数据结构。

msqid_ds 的结构定义如下：

```
struct msqid_ds
{
    struct ipc_perm msg_perm;
    struct msg *msg_first;
    struct msg *msg_last;
    time_t msg_stime;
    time_t msg_rtime;
    time_t msg_ctime;
    struct wait_queue *wwait;
    struct wait_queue *rwait;
    unsigned short int msg_cbytes;
    unsigned short int msg_qnum;
    unsigned short int msg_qbytes;
    ipc_pid_t msg_lspid;
    ipc_pid_t msg_lrpid;
};
```

msg_first　指向第一个存于队列的信息。

msg_last　指向最后一个存于队列的信息。

msg_stime　最后一次用 msgsnd（）送入信息的时间。

msg_rtime　最后一次用 msgrcv（）读取信息的时间。

msg_ctime　最后一次更动此信息队列结构的时间。

msg_cbytes　目前信息队列中存放的字符数。

msg_qnum　信息队列中的信息个数。

msg_qbytes　信息队列所能存放的最大字符数。

msg_lspid　　最后一个用 msgsnd（）送入信息的进程识别码。

msg_lrpid　　最后一个用 msgrcv（）读取信息的进程识别码。

msg_perm　　所使用的结构 ipc_perm 定义如下：

```
struct ipc_perm
{
    key_t key;
    unsigned short int uid;
    unsigned short int gid;
    unsigned short int cuid;
    unsigned short int cgid;
    unsigned short int mode;
    unsigned short int seq;
};
```

- **key**：此信息的 IPC key。
- **uid**：此信息队列所属的用户识别码。
- **gid**：此信息队列所属的组识别码。
- **cuid**：建立信息队列的用户识别码。
- **cgid**：建立信息队列的组识别码。
- **mode**：此信息队列的读写权限。
- **seq**：序号。

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 cmd 为 IPC_STAT，却无权限读取该信息队列。

EFAULT　参数 buf 指向无效的内存地址。

EIDRM　参数 key 所指的信息队列已经删除。

EINVAL　无效的参数 cmd 或 msqid。

EPERM　参数 cmd 为 IPC_SET 或 IPC_RMID，却无足够的权限执行。

范 例

请参考 msgget（）。

msgget（建立信息队列）

相关函数：ftok，msgctl，msgsnd，msgrcv

表头文件：#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

定义函数：int msgget（key_t key，int msgflg）；

函数说明：msgget（）用来取得参数 key 所关联的信息队列识别代号。如果参数 key 为 IPC_PRIVATE 则会建立新的信息队列，如果 key 不为 IPC_PRIVATE，也不是已建立的 IPC key，那么系统则会视参数 msgflg 是否有 IPC_CREAT 位（msgflg & IPC_CREAT 为真）来决定建立 IPC key 为 key 的信息队列。如果参数 msgflg 包含了 IPC_CREAT 和 IPC_EXCL 位，而无法依参数 key 来建立信息队列，则表示信息队列已存在。此外，参数 msgflg 也用来决定信息队列的存取权限，其值相当于 open（）的参数 mode 用法（执行位不使用）。

返 回 值：若成功则返回信息队列识别代号，否则返回 -1，错误原因存于 errno 中。

错误代码：
EACCESS　参数 key 所指的信息队列存在，但无存取的权限。
EEXIST　欲建立 key 所指新的信息队列，但该队列已经存在。
EIDRM　参数 key 所指的信息队列已经删除。
ENOENT　参数 key 所指的信息队列不存在，参数 msgflg 也未设 IPC_CREAT 位。
ENOMEM　核心内存不足。
ENOSPC　超过可建立的信息队列最大数目。

范 例

```
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define KEY          1234
```

```
#define TEXT_SIZE      48
struct msgbuffer
{
  long mtype;
  char mtext[TEXT_SIZE];
}msgp;
main ()
{
  int msqid;
  msqid = msgget (KEY, IPC_CREAT | 0600) ;          /* 建立信息队列 */
  if (fork () == 0) {
  msgp.mtype = 1;
    strcpy (msgp.mtext, "Hi! I am child process!\n") ;
    msgsnd (msqid, &msgp, TEXT_SIZE, 0) ;    /* 传送信息至队列 */
    return;
  } else {
    sleep (3) ;   /* 等子进程执行完毕 */
    msgrcv (msqid, &msgp, TEXT_SIZE, 0, 0) ;     /* 接收队列的信息 */
    printf ("parent receive mtext: %s", msgp.mtext) ;
    msgctl (msqid, IPC_RMID, NULL) ;         /* 删除信息队列 */
  }
}
```

执行结果

```
parent receive mtext. Hi! I am child process!
```

msgrcv（从信息队列读取信息）

相关函数：msgget，msgctl，msgsnd

表头文件：#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

定义函数：int msgrcv(int msqid, struct msgbuf *msgp, int msgsz, long msgtyp, int msgflg);

函数说明：msgrcv（）用来从参数 msqid 指定的信息队列读取信息出来，然后存于参数 msgp 所指定的结构内。msgbuf 结构，其定义如下：

```
struct msgbuf
{
   long mtype;  /* 信息的种类 */
   char mtext[1];   /* 信息数据 */
};
```

参数 msgsz 为信息数据的长度，即 mtext 参数的长度。

参数 msgtyp 是用来指定所要读取的信息种类：msgtyp = 0 返回队列内第一项信息。msgtyp > 0 返回队列内第一项 msgtyp 与 mtype 相同的信息。msgtyp < 0 返回队列内第一项 mtype 小于或等于 msgtyp 绝对值的信息。

参数 msgflg 可以设成 IPC_NOWAIT，意思是如果队列内没有信息可读，则不要等待，立即返回 ENOMSG。如果 msgflg 设成 MSG_NOERROR，则信息大小超过参数 msgsz 时会被截断。

返 回 值：若成功则返回实际读取到的信息数据长度，否则返回 -1，错误原因存于 errno 中。

错误代码：

E2BIG	信息数据长度大于参数 msgsz 却没设置 MSG_NOERROR。
EACCESS	无权限读取该信息队列。
EFAULT	参数 msgp 指向无效的内存地址。
EIDRM	参数 msqid 所指的信息队列已经删除。
EINTR	等待读取队列内的信息情况下被信号中断。
ENOMSG	参数 msgflg 设成 IPC_NOWAIT，而队列内没有信息可读。

范 例

请参考 msgget（）。

msgsnd（将信息送入信息队列）

相关函数：msgget，msgctl，msgrcv

表头文件：#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

定义函数：int msgsnd（int msqid，struct msgbuf *msgp，int msgsz，int msgflg）；

函数说明：msgsnd（）用来将参数 msgp 指定的信息送至参数 msqid 的信息队列内。

参数 msgp 为 msgbuf 结构，其定义如下：

```
struct msgbuf
{
  long mtype;        /* 信息的种类，必须大于 0 */
  char mtext[1];         /* 信息数据 */
};
```

参数 msgsz 为信息数据的长度，即 mtext 参数的长度。

参数 msgflg 可以设成 IPC_NOWAIT，意思是如果队列已满或有其他情况无法马上送入信息，则立即返回 EAGAIN。

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：EAGAIN　参数 msgflg 设 IPC_NOWAIT，而队列已满。

EACCESS　无权限写入该信息队列。

EFAULT　参数 msgp 指向无效的内存地址。

EIDRM　参数 msqid 所指的信息队列已经删除。

EINTR　队列已满而处于等待情况下被信号中断。

EINVAL　无效的参数 msqid 、msgsz 或参数 mtype 小于 0。

范 例

请参考 msgget（）。

semctl（控制信号队列的操作）

相关函数：semget，semop

表头文件：#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

定义函数： int semctl（int semid，int semnum，int cmd，union semun arg）；

函数说明： semctl（）提供了几种方式来控制信号队列的操作。参数 semid 为欲处理的信号队列识别代码，参数 cmd 为欲控制的操作，有下列几种数值：

IPC_STAT　把信号队列的 semid_ds 结构数据复制到参数 arg.buf。

IPC_SET　将参数 arg.buf 所指的 semid_ds 结构中的 sem_perm.uid、sem_perm.gid 和 sem_perm.mode 复制到共享内存的 semid_ds 结构内。

IPC_RMID　删除信号队列和其数据结构。

GETALL　将信号队列所有集合的 semval 值复制到参数 arg.array。

GETPID　返回信号队列的 sempid 值。

GETVAL　把信号队列中参数 semnum 指定集合的 semval 值返回。

GETZCNT　返回信号队列的 semzcnt 值。

SETALL　将信号队列所有集合的 semval 值设置成参数 arg.array。

SETVAL　把信号队列中参数 semnum 指定集合的 semval 值设 arg.val。

union semun 的定义如下：

```
union semun
{
    int val;
    struct semid_ds *buf;
    unsigned short int *array;
    struct seminfo *__buf;
};
```

val　SETVAL 用的 semval 值。

buf　指向 IPC_STAT 或 IPC_SET 用的 semid_ds 结构。

Array　GETALL 或 SETAL 用的数组。

semid_ds 的结构定义如下：

```
struct semid_ds
{
    struct ipc_perm sem_perm;
```

```
    time_t sem_otime;
    time_t sem_ctime;
    struct sem *sembase;
    struct sem_queue *sem_pending;
    struct sem_queue *sem_pending_last;
    struct sem_undo *undo;
    unsigned short int sem_nsems;
};
```

sem_nsems　　信号集合的数目。
shm_ctime　　上一次调用 semctl（）的时间。
sem_otime　　上一次调用 semop（）的时间。
sem_perm　　ipc_perm 结构定义请参考 msgctl（）。
sembase 参数的结构 sem 定义如下：

```
struct sem
{
    int    semval;
    int    sempid;
};
```

semval　　信号 semval 值。
sempid　　上一次更动此数据结构的进程识别代码。

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 cmd 为 IPC_STAT，却无权限读取该信号队列。
EFAULT　参数 arg.buf 或 arg.array 指向无效的内存地址。
EIDRM　参数 semid 所指的信号队列已经删除。
EINVAL　无效的参数 cmd 或 semid。
EPERM　参数 cmd 为 IPC_SET 或 IPC_RMID，却无足够的权限执行。
ERANGE　参数 cmd 为 SETALL 或 SETVAL，但欲设置的 semval 值却小于 0 或大于 SEMVMX（32767）。

semget（配置信号队列）

相关函数：semctl，semop

表头文件：#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

定义函数：int semget（key_t key，int nsems，int semflg）；

函数说明：semget（）用来取得参数 key 所关联的信号识别代号。如果参数 key 为 IPC_PRIVATE 则会建立新的信号队列，参数 nsems 为信号集合的数目。如果 key 不为 IPC_PRIVATE，也不是已建立的信号队列 IPC key，那么系统会视参数 semflg 是否有 IPC_CREAT 位（semflg & IPC_CREAT 为真）来决定 IPC key 为参数 key 的信号队列。
如果参数 semflg 包含了 IPC_CREAT 和 IPC_EXCL 位，而无法依参数 key 来建立信号队列，则表示该信号队列已存在。此外，参数 semflg 也用来决定信号队列的存取权限，其值相当于 open（）的参数 mode 用法（执行位不使用）。

返 回 值：若成功则返回信号队列识别代号，否则返回 -1，错误原因存于 errno 中。

错误代码：

EACCES	key 指定的信号队列虽存在却无权限可存取。
EEXIST	欲建立 key 所指新的信号队列，但已经存在。
EIDRM	参数 key 所指的信号队列已经删除。
ENOENT	参数 key 所指的信号队列不存在，参数 semflg 也未设 IPC_CREAT 位。
ENOMEM	核心内存不足。
ENOSPC	已超过系统允许可建立的信号队列最大值。

semop（信号处理）

相关函数：semctl，semget

表头文件：#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

定义函数：int semop（int semid，struct sembuf *sops，unsigned nsops）；

函数说明：参数 semid 为欲处理的信号队列识别代码，参数 sops 指向结构 sembuf，其结构定义如下：

```
struct sembuf
{
   short int sem_num;
   short int sem_op;
   short int sem_flg;
};
```

sem_num　欲处理的信号编码，0 代表第一个信号。

sem_op：

1. 如果此值大于 0：此值会加至 semval。
2. 如果此值等于 0：semop（）会等到 semval 降为 0，除非 sem_flg 参数有设为 IPC_NOWAIT。
3. 如果此值小于 0：若 semval 大于或等于 sem_op 的绝对值，则 semval 的值会减去 sem_op 的绝对值。若 semval 比 sem_op 的绝对值小且 sem_flg 有设 IPC_NOWAIT，则 semop（）会立刻返回错误。

参数 nsops 代表参数 sops 的结构数目。

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：

E2BIG	参数 nsops 大于系统允许的最大值（SEMOPM）。
EACCESS	无权限存取指定的信号集合。
EAGAIN	该调用无法马上处理，而 sem_flg 参数设了 IPC_NOWAIT。
EFAULT	参数 sops 指向无效的内存地址。
EFBIG	sem_num 参数小于 0 或大于等于信号集合的数目。
EIDRM	参数 semid 所指的信号队列已经删除。
EINVAL	无效的参数 cmd 或 semid。
EINTR	此调用等待系统处理被信号所中断。
EINVAL	参数 semid 指定的信号不存在或小于 0，也可能是 nsops 为负值。

shmat（attach 共享内存）

相关函数：shmget，shmctl，shmdt

表头文件：#include <sys/types.h>

#include <sys/shm.h>

定义函数：void *shmat（int shmid，const void *shmaddr，int shmflg）；

函数说明：shmat（）用来将参数 shmid 所指的共享内存和目前进程连接（attach）。参数 shmid 为欲连接的共享内存识别代码，而参数 shmaddr 有下列几种情况：

- shmaddr 为 0，核心自动选择一个地址。
- shmaddr 不为 0，参数 shmflg 也无指定 SHM_RND 旗标，则以参数 shmaddr 为连接地址。
- shmaddr 不为 0，但参数 shmflg 设置了 SHM_RND 旗标，则参数 shmaddr 会自动调整为 SHMLBA 的整数倍。

参数 shmflg 还可以有 SHM_RDONLY 旗标，代表此连接操作只是用来读取该共享内存。

返 回 值：若成功则返回已连接好的地址，否则返回 -1，错误原因存于 errno 中。

附加说明：1. 在经过 fork（）后，子进程将继承已连接的共享内存地址。

2. 在经过 exec（）后，已连接的共享内存地址会自动脱离（detach）。

3. 在结束进程后，已连接的共享内存地址会自动脱离（detach）。

错误代码：EACCESS　无权限以指定方式连接共享内存。

EINVAL　无效的参数 shmid 或 shmaddr；或者 shmaddr 地址并无分页边界对齐也无设置 SHM_RND 旗标。

ENOMEM　核心内存不足。

范 例

```
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
#define KEY    1234
#define SIZE   1024    /* 欲建立的共享内存大小 */
main ()
{
  int shmid;
  char *shmaddr;
  struct shmid_ds buf;
  shmid = shmget (KEY, SIZE, IPC_CREAT | 0600) ;  /* 建立共享内存 */
  if (fork () == 0) {
    shmaddr = (char *) shmat (shmid, NULL, 0) ;
    strcpy (shmaddr, "Hi! I am child process!\n") ;
    shmdt (shmaddr) ;
    return;
  } else {
    sleep (3) ;   /* 等子进程执行完毕 */
  shmctl (shmid, IPC_STAT, &buf) ;      /* 取得共享内存状态 */
  printf ("shm_segsz = %d bytes\n", buf.shm_segsz) ;
  printf ("shm_cpid  = %d\n", buf.shm_cpid) ;
  printf ("shm_lpid  = %d\n", buf.shm_lpid) ;
  shmaddr = (char *) shmat (shmid, NULL, 0) ;
  printf ("%s", shmaddr) ;       /* 显示共享内存内容 */
  shmdt (shmaddr) ;
  shmctl (shmid, IPC_RMID, NULL) ;      /* 删除共享内存 */
   }
}
```

执行结果

```
shm_segsz = 1024 bytes
shm_cpid  = 1176
shm_lpid  = 1177
Hi! I am child process!
```

shmctl（控制共享内存的操作）

相关函数：shmget，shmat，shmdt

表头文件： #include <sys/ipc.h>

#include <sys/shm.h>

定义函数： int shmctl（int shmid，int cmd，struct shmid_ds *buf）；

函数说明： shmctl（）提供了几种方式来控制共享内存的操作。参数 shmid 为欲处理的共享内存识别代码，参数 cmd 为欲控制的操作，有下列几种数值：

IPC_STAT	把共享内存的 shmid_ds 结构数据复制到引数 buf。
IPC_SET	将参数 buf 所指的 shmid_ds 结构中的 shm_perm.uid、shm_perm.gid 和 shm_perm.mode 复制到共享内存的 shmid_ds 结构内。
IPC_RMID	删除共享内存和其数据结构。
SHM_LOCK	不让此共享内存置换到 swap。
SHM_UNLOCK	允许此共享内存置换到 swap。

SHM_LOCK 和 SHM_UNLOCK 为 Linux 特有，且唯有超级用户（root）允许使用。

shmid_ds 的结构定义如下：

```
struct shmid_ds
{
    struct   ipc_perm shm_perm;
    int      shm_segsz;
    time_t   shm_atime;
    time_t   shm_dtime;
    time_t   shm_ctime;
    unsigned   short   shm_cpid;
    unsigned   short   shm_lpid;
    short    shm_nattch;
    unsigned   short   shm_npages;
    unsigned   long    *shm_pages;
    struct  shm_desc *attaches;
};
```

shm_segsz	共享内存的大小（bytes）。
shm_atime	最后一次 attach 此共享内存的时间。
shm_dtime	最后一次 detach 此共享内存的时间。
shm_ctime	最后一次更动此共享内存结构的时间。

shm_cpid　　建立此共享内存的进程识别码。
shm_lpid　　最后一个操作此共享内存的进程识别码。
shm_perm　　所使用的结构 ipc_perm 定义请参考 msgctl（）。

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 cmd 为 IPC_STAT，却无权限读取该共享内存。
EFAULT　参数 buf 指向无效的内存地址。
EIDRM　参数 shmid 所指的共享内存已经删除。
EINVAL　无效的参数 cmd 或 shmid。
EPERM　参数 cmd 为 IPC_SET 或 IPC_RMID，却无足够的权限执行。

范 例

请参考 shmat（）。

shmdt（detach 共享内存）

相关函数：shmget，shmctl，shmat

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

定义函数：int shmdt（const void *shmaddr）;

函数说明：shmdt（）用来将先前用 shmat（）连接（attach）好的共享内存脱离（detach）目前的进程。参数 shmaddr 为先前 shmat（）返回的共享内存地址。

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：EINVAL　无效的参数 shmaddr；或者 shmaddr 地址并非共享内存地址。

范 例

请参考 shmat（）。

shmget（配置共享内存）

相关函数：ftok，shmctl，shmat，shmdt

表头文件：#include <sys/ipc.h>

#include <sys/shm.h>

定义函数：int shmget（key_t key，int size，int shmflg）；

函数说明：shmget（）用来取得参数 key 所关联的共享内存识别代号。如果参数 key 为 IPC_PRIVATE 则会建立新的共享内存，其大小由参数 size 决定。如果 key 不为 IPC_PRIVATE，也不是已建立的共享内存 IPC key，那么系统则会视参数 shmflg 是否有 IPC_CREAT 位（shmflg & IPC_CREAT 为真）来决定 IPC key 为参数 key 的共享内存。如果参数 shmflg 包含了 IPC_CREAT 和 IPC_EXCL 位，而无法依参数 key 来建立共享内存，则表示该共享内存已存在。此外，参数 shmflg 也用来决定共享内存的存取权限，其值相当于 open（）的参数 mode 用法（执行位不使用）。

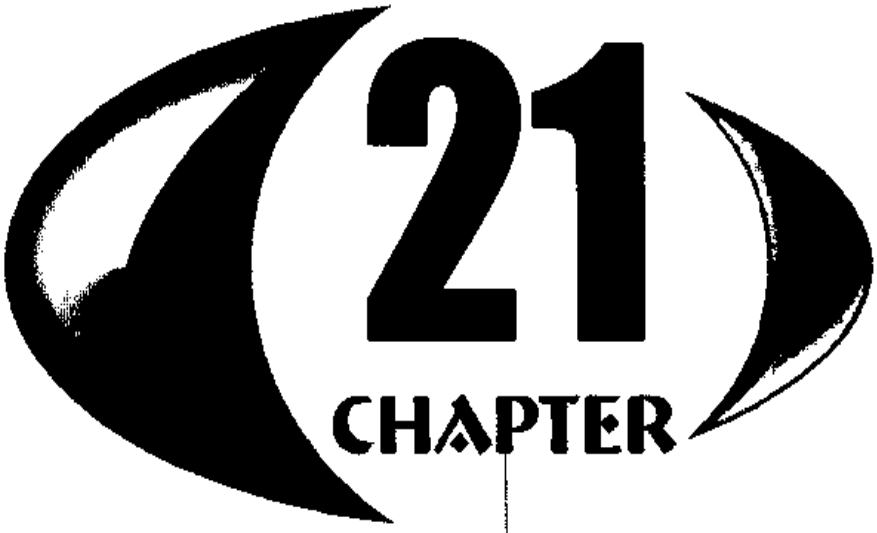
返 回 值：若成功则返回共享内存识别代号，否则返回 -1，错误原因存于 errno 中。

错误代码：

EINVAL	参数 size 小于 SHMMIN 或大于 SHMMAX。
EEXIST	欲建立 key 所指新的共享内存，但已经存在。
EIDRM	参数 key 所指的共享内存已经删除。
ENOENT	参数 key 所指的共享内存不存在，参数 shmflg 也未设 IPC_CREAT 位。
ENOMEM	核心内存不足。
ENOSPC	已超过系统允许可建立的共享内存最大值（SHMALL）

范 例

请参考 shmat（）。



记录函数

closelog（关闭信息记录）

相关函数：openlog，syslog

表头文件：#include <syslog.h>

定义函数：void closelog（void）；

函数说明：closelog（）用来结束由 openlog（）打开的信息记录。

返 回 值：无

范 例

请参考 syslog（）。

openlog（准备做信息记录）

相关函数：syslog，closelog

表头文件：#include <syslog.h>

定义函数：void openlog（char *ident，int option，int facility）；

函数说明：openlog（）用来做 syslog（）调用前的准备。参数 ident 字符串用来指明做记录的程序名称，一般都用 argv[0]。参数 option 可以有下列几种选项组合：

LOG_CONS	如果无法将信息送至 syslogd 则直接输出到控制台（console）。
LOG_NDELAY	直接打开和 syslogd 的连线，一般都会等到第一次有信息要记录时才会连线。
LOG_PERROR	输出至标准输出设备（stderr）中。
LOG_PID	将信息字符串加上产生信息的进程识别码。

参数 facility 的信息种类有下列几种：

LOG_AUTH	认证相关的信息。
LOG_CRON	由 cron 或 at 程序产生的信息。

LOG_DAEMON 系统 daemon 产生的信息。
LOG_KERN 核心产生的信息。
LOG_LPR 列表机队列产生的信息。
LOG_MAIL 由 mail 相关程序产生的信息。
LOG_NEWS 由 news 相关程序产生的信息。
LOG_USER 用户进程产生的信息（预设值）。
LOG_UUCP 由 uucp 相关程序产生的信息。
LOG_LOCAL0 至 LOG_LOCAL7 为保留使用。

信息种类会依照 /etc/syslog.conf 的设置来做输出，例如：

```
mail.*                  /var/log/maillog
```

则表示信息种类为 LOG_MAIL 的信息字符串都会输出至/var/log/maillog 文件。

返 回 值：无

范 例

请参考 syslog（）。

syslog（将信息记录至系统日志文件）

相关函数：openlog，closelog

表头文件：#include <syslog.h>

定义函数：void syslog（int priority，char *format，...）；

函数说明：syslog（）类似 printf（）般，只是将字符串格式和信息字符串输出至系统日志文件中。

参数 priority 可以用来指定信息的种类或等级。参数 format 的字符串格式请参考 printf（），字符串格式多了 %m 可用来依全局变量 errno 来显示错误描述字符串。

参数 priority 的信息种类可参考 openlog（）。

参数 priority 的信息等级有下列几种：

LOG_EMERG 核心不稳或其他紧急的信息。

LOG_ALERT	须立即处理的错误信息。
LOG_CRIT	严重的错误信息。
LOG_ERR	一般的错误信息。
LOG_WARNING	警告信息。
LOG_NOTICE	提醒留意的信息。
LOG_INFO	资讯相关的信息。
LOG_DEBUG	除错相关的信息。

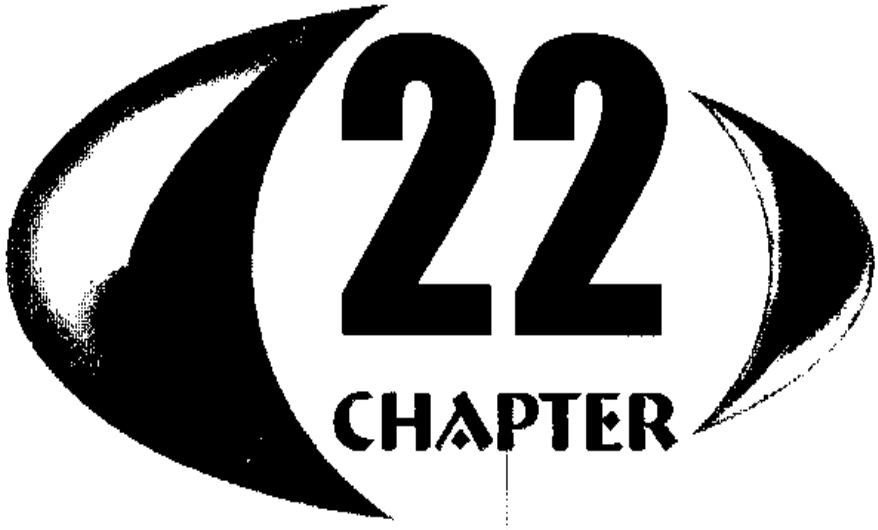
返 回 值：无

范 例

```
#include <syslog.h>
main (int argc, char *argv[])
{
  char *str = "test";
  openlog (argv[0], LOG_PID, LOG_USER) ;
  syslog (LOG_INFO, "%s\n", str) ;
  closelog () ;
}
```

（输出至 /var/log/messages 中）

Jan 14 16：44：22 localhost ./openlog[898]：test



环境变量函数

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

```
/* 取得环境变量 USER 内容 */
#include <stdlib.h>
main ()
{
  char *p;
  if ( (p = getenv ("USER") ) )
    printf ("USER=%s\n", p) ;

}
```

执行结果

```
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

```
/* 设置环境变量 USER 内容为 test */
#include <stdlib.h>
main ()
{
  char *p;
  if ( (p = getenv ("USER") ) )
    printf ("USER=%s\n", p) ;
  putenv ("USER=test") ;
  printf ("USER=%s\n", getenv ("USER") ) ;
}
```

执行结果

```
USER=root
USER=test
```

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，且该环境变量原已有内容，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

```
/* 可与 putenv () 范例对照参考 */
#include <stdlib.h>
main ()
{
  char *p;
  if ( (p = getenv ("USER") ) )
    printf ("USER=%s\n", p) ;
  setenv ("USER", "test", 1) ;  /* 设置 USER 环境变量内容为 test */
  printf ("USER=%s\n", getenv ("USER") ) ;
  unsetenv ("USER") ;   /* 清除 USER 环境变量内容 */
  printf ("USER=%s\n", getenv ("USER") ) ;
}
```

执行结果

```
USER=root
USER=test
USER= (null)     /* USER 环境变量内容已清除，所以为空字符串 (null) */
```

unsetenv（清除环境变量内容）

相关函数：getenv，putenv，setenv

表头文件：#include <stdlib.h>

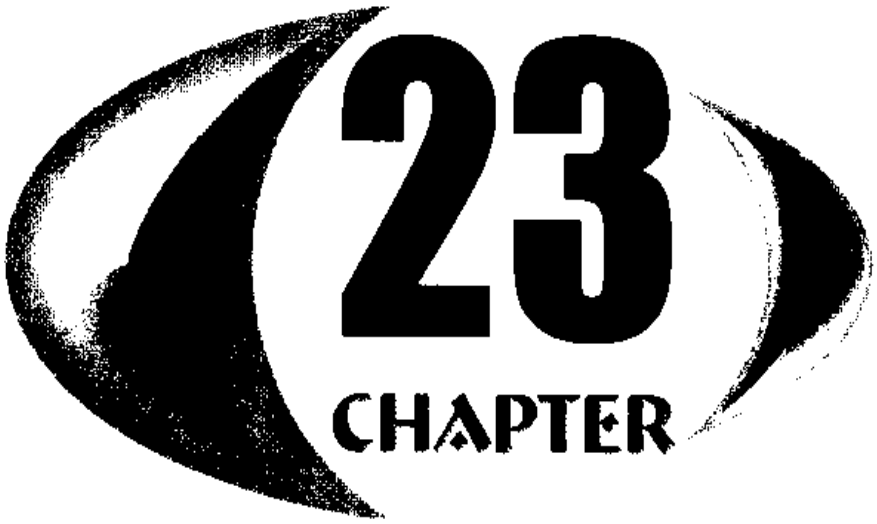
定义函数：void unsetenv（const char *name）;

函数说明：unsetenv（）用来清除特定的环境变量的内容。参数 name 为欲清除的环境变量名称字符串。

返 回 值：无

范 例

请参考 setenv（）范例。



正则表达式

regcomp（编译正则表达式字符串）

相关函数：regexec，regerror，regfree

表头文件：#include <stdio.h>

#include <regex.h>

定义函数：int regcomp（regex_t *preg，const char *regex，int cflags）；

函数说明：参数 preg 为一指针，指向 pattern buffer，用来存放编译后的结果。参数 regex 指向正则表达式的字符串，而参数 cflags 则是旗标。

参数 cflags 有下列几种情况，可以用 OR（|）组合：

REG_EXTENDED	使用 POSIX 延伸正规表示语法，否则使用 POSIX 基本语法。
REG_ICASE	忽略大小写的差异。
REG_NOSUB	忽略参数 nmatch 和 pmatch。
REG_NEWLINE	特殊字符.将不会以换行字符作比较，特殊字符 ^ 则代表换行字符后的第一个位置，特殊字符 $ 则代表换行字符的前一个位置。

返 回 值：如果成功返回 0，有错误发生时则返回错误原因。请参考 regerror（）。

附加说明：正则表达式（regular expression：简称 RE）是以字符串来表达某种规则的字符串集合，以便作搜索之用。分作一般字符和特殊字符两类：

一般字符：

非 .*?+^$|\[]{}（）<> 等符号的字符。

特殊字符：

. 代表一个任何字符，如 "abc." 代表 "abc" 加上任何一个字符。

? 代表出现一次或无出现的字符，如 "abc?" 代表 "abc" 或 "abc 加上任何一个字符。

* 代表任何重复出现的字符，如 "abc*" 代表重复出现 "abc" 的字符串，包含 "abc"。

+ 代表重复出现至少一次的字符，如 "abc+" 代表重复出现 "abc" 的字符串，但不包含 "abc"。

| 代表或者（OR）的意思，如 "abc|ABC" 代表 "abc 或 "ABC"。
\ 代表下一字符非特殊字符，如 "\." 代表 "." 为一般字符。
^ 代表欲搜索的字符串必须在行首。
$ 代表欲搜索的字符串必须在行尾。
[代表字符集合开始。
] 代表字符集合结束。

字符集合以 [为代表开始，] 则代表结束，在此字符集合内的符号有下列几种情况：

^ 用来把字符集合反向。
- 用来指定范围，如 [a-z] 或 [0-9]。

底下是几个标准的字符集合名称和意义：

[: alnum :]	英文字母或数字，即代表 [a-z]、[A-Z] 和 [0-9]。
[: alpha:]	英文字母，即代表 [a-z]、[A-Z]。
[: blank:]	空格键（space）或定位字符（tab）。
[: cntrl:]	ASCII 码的控制字符。
[: digit:]	阿拉伯数字，即 [0-9]。
[: graph:]	可打印字符，但不包括空格字符（' '）。
[: lower:]	小写英文字母，即 [a-z]。
[: print:]	可打印的字符，包括空格字符（' '）。
[: punct:]	标点符号或特殊符号，即非空格、非数字和非英文字母。
[: space:]	代表空格（' '）、定格（'\t'）、CR（'\r'）、换行（'\n'）、垂直跳格（'\v'）或翻页（'\f'）的情况。
[: upper:]	大写英文字母，即 [A-Z]。
[: xdigit:]	十六进制的数字，即 [0-9]、[a-f] 和 [A-F]。

范 例

请参考 regexec（）。

regerror（取得正则搜索的错误原因）

相关函数：regcomp，regexec，regfree

表头文件：#include <stdio.h>
#include <regex.h>

定义函数：size_t regerror（int errcode，const regex_t *preg，char *errbuf，size_t errbuf_size）；

函数说明：regerror（）用来取得经 regcomp（）或 regexec（）的错误原因。参数 errcode 为之前 regcomp（）或 regexec（）返回的错误代码，而参数 preg 为一指针，指向 pattern buffer，参数 errbuf 指向欲存放错误字符串的缓冲区，参数 errbuf_size 则为该缓冲区大小。

返 回 值：返回错误字符串的长度。

范 例

```
#include <stdio.h>
#include <regex.h>
main ()
{
  regex_t preg;
  char *regex = "[ab[d";
  char errbuf[256];
  int errcode;
  errcode = regcomp (&preg, regex, 0) ;
  if (errcode != 0) {
    regerror (errcode, &preg, errbuf, sizeof (errbuf) ) ;
    printf ("regerror : %s\n", errbuf) ;
  }
}
```

执行结果

```
regerror : Unmatched [ or [^
```

regexec（进行正则表达式的搜索）

相关函数：regcomp，regerror，regfree

表头文件：#include <stdio.h>

#include <regex.h>

定义函数：int regexec（const regex_t *preg，const char *string，size_t nmatch，regmatch_t pmatch[]，int eflags）；

函数说明：参数 preg 为一指针，指向 pattern buffer，此为先前经 regcomp（）编译后的结果。参数 string 指向欲寻找的字符串地址，参数 nmatch 代表 pmatch 数组的大小，参数 pmatch 则为一结构数组，regexec（）会将搜索后的结果以此结构数组返回。

结构 regmatch_t 的定义如下：

```
typedef struct
{
   regoff_t rm_so;
   regoff_t rm_eo;
} regmatch_t;
```

rm_so代表符合条件的起始位置。

rm_eo　　代表符合条件的结束位置。

如果 rm_so 的值为 -1，则代表此结构并未让 regexec（）使用。

参数 eflags 有两种可能值，可使用 OR（|）组合：

REG_NOTBOL　　让特殊字符 ^ 无作用。

REG_NOTEOL　　让特殊字符 $ 无作用。

返 回 值：如果成功返回 0，有错误发生时则返回错误代码。请参考 regerror（）。

范 例

```
#include <stdio.h>
#include <regex.h>
#define nmatch 5
char *string =
```

```
"Tabs are 8 characters, and thus indentations are also 8 characters.\n";
main ()
{
  regex_t preg;
  char *regex = "char[a-z]+";
  regmatch_t pmatch[nmatch];
  unsigned int i, len;
  /* 初始化 pattern buffer */
  bzero (&preg, sizeof (regex_t) ) ;
  if (regcomp (&preg, regex, REG_EXTENDED) == 0)
  {
    if (regexec (&preg, string, nmatch, pmatch, 0) == 0)
    {
      for (i = 0; i < nmatch; i++)
      {
        if (pmatch[i].rm_so == -1) continue;
        len = (pmatch[i].rm_eo - pmatch[i].rm_so) ;
        printf ("%.*s\n", len, (string + pmatch[i].rm_so) ) ;
      }
    }
    regfree (&preg) ;
  }
}
```


```
characters
```

regfree（释放正则表达式使用的内存）

相关函数：regcomp，regexec，regerror，regfree

表头文件：#include <stdio.h>#include <regex.h>

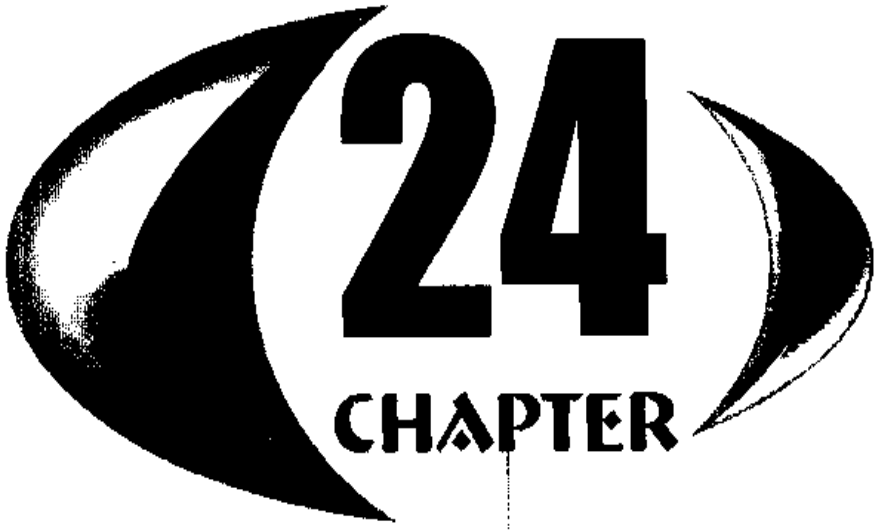
定义函数：void regfree（regex_t *preg）；

函数说明：此函数用来释放先前 regcomp（）和 regexec（）使用的 pattern buffer。参数 preg 为指向此 pattern buffer 的指针。

返 回 值：无

范 例

请参考 regexec（）。



动态函数

dlclose（关闭动态函数库文件）

相关函数：dlopen，dlsym，dlerror

表头文件：#include <dlfcn.h>

定义函数：int dlclose（void *handle）;

函数说明：dlclose（）用来减少参数 handle 指定的动态对象的参考计数，如果该计数为 0 则自动关闭该动态函数库文件。动态对象的参考计数会因重复调用 dlopen（）函数而增加。

附加说明：编译时请加入 -ldl 参数。

范 例

请参考 dlopen（）。

dlerror（动态函数错误处理）

相关函数：dlopen，dlclose，dlsym

表头文件：#include <dlfcn.h>

定义函数：const char *dlerror（void）;

函数说明：dlerror（）用来处理调用 dlopen（）、dlclose（）与 dlsym（）时所发生的错误情况。dlerror（）内存有一个错误变量，当有错误发生时错误原因会存在此变量，调用 dlerror（）即把此变量返回然后清除。因此，dlerror（）只会返回一次错误，如果没有再次的错误，dlerror（）则会持续返回 NULL。利用这个特性，dlerror（）通常会在调用完 dlsym（）后马上调用，以便判断 dlsym（）是否真的有错误发生。

附加说明：编译时请加入 -ldl 参数。

返 回 值：如果处理动态函数库时有错误发生则返回一个描述该错误原因的字符串指

针。否则返回 NULL。

范 例

```
/* 下面是常见的程序片段 */
char *error;   /* 由于 dlerror () 只返回一次错误原因，因此最好能保留 */
dlerror () ;   /* 先清除错误变量 */
function = dlsym (handle, symbol) ;
if ( (error = dlerror () ) != NULL) {
   /* 错误处理 */
}
```

dlopen（打开动态函数库文件）

相关函数：dlsym，dlclose，dlerror

表头文件：#include <dlfcn.h>

定义函数：void *dlopen（const char *filename，int flag）;

函数说明：dlopen（）用来将参数 filename 指定的动态函数库文件载入到内存，若参数 filname 为 NULL，则会打开进程本身的文件。如果参数 filename 并非以 / 字符开头的绝对路径，dlopen（）便会依下列的次序搜索文件路径：

1. 依环境变量 LD_LIBRARY_PATH 决定。
2. 依 /etc/ld.so.cache 文件内容中注明的路径。
3. 从 /usr/lib 下查找。
4. 从 /lib 下查找。

参数 flag 有几种可能：

RTLD_LAZY	暂缓决定，等有需要时再解出符号。
RTLD_NOW	立即决定，返回前解出所有未决定的符号。
RTLD_GLOBAL	允许导出符号。

dlopen（）执行成功后会返回所打开的共享对象代码，此代码只能当作 dlsym（）及 dlclose（）函数的参数。

附加说明：编译时请加入 -ldl 参数。

返 回 值：执行成功则返回所打开的共享对象代码，如果失败或有错误则返回 NULL。

范 例

```
#include <dlfcn.h>
main ()
{
  int (*my_printf) (const char *, ...) ;
  void *handle = dlopen ("/lib/libc-2.1.2.so", RTLD_LAZY) ;
  my_printf = dlsym (handle, "printf") ;    /* 查找 printf 符号地址 */
  /* 调用 my_printf 即调用 libc-2.1.2.so 里的 printf () */
  my_printf ("printf () 's address : 0x%x\n", my_printf) ;
  dlclose (handle) ;

}
```

执行结果

```
printf () 's address : 0x40067f4c
```

dlsym（从共享对象中搜索动态函数）

相关函数：dlopen，dlclose，dlerror

表头文件：#include <dlfcn.h>

定义函数：void *dlsym（void *handle，char *symbol);

函数说明：dlsym（）用来从共享对象中搜索动态函数。参数 handle 为调用 dlopen（）所返回的共享对象代码，参数 symbol 则表示欲搜寻符号的名称字符串。dlsym（）会返回该字符串的地址，如果返回 NULL（0）不一定是因为有错误发生，有可能该地址本身即为 0 的情况。因此通常会接着调用 dlerror（）来判断是否真的发生错误。

附加说明：编译时请加入 -ldl 参数。

返 回 值：执行成功则返回所打开的共享对象代码。如果返回 NULL 请参考 dlerror()。

范 例

请参考 dlopen () 及 dlerror ()。

其他函数

getopt（分析命令行参数）

相关函数：无

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误信息，并将全域变量 optopt 设为 '?' 字符，如果不希望 getopt()印出错误信息，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;  /* 不打印出错误信息 */
  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) ;
}
```

执行结果

```
$ ./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为终端机）

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

请参考 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（int 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;          /* 秒数  */
    time_t tv_usec;         /* 毫秒数 */
};
```

返 回 值：如果参数 timeout 设为 NULL（0），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

下面是常见的程序范例片段：

```
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) ;
}
```


```
/dev/tty is a tty.
ttyname = /dev/tty
```

编译程序 - gcc

gcc 是 Linux 下使用的 C 语言编译器，主要有预处理、编译、翻译和连接四个过程，下面是 gcc 可辨识的文件类型：

.c	C 源代码；预处理，编译，翻译
.C	C++ 源代码；预处理，编译，翻译
.cc	C++ 源代码；预处理，编译，翻译
.cxx	C++ 源代码；预处理，编译，翻译
.m	Objective-C 源代码；预处理，编译，翻译
.i	已经预处理后的 C 源代码；编译，翻译
.ii	已经预处理后的 C++ 源代码 ；编译，翻译
.s	汇编语言源代码 ；翻译
.S	汇编语言源代码 ；预处理，翻译
.h	前置处理文件

gcc 常用且好用的参数或选项：

-c	对源代码只作编译或翻译操作，而不作连接（link）。处理过的文件即为目的文件，其文件名以 .o 结尾，不过可以使用 -o 参数来指定目的文件文件名。
-S	将源代码编译后不作翻译操作，然后以汇编语言作输出，预设产生的文件将以.s为延伸文件名。
-o file	此选项用来告知 GCC 在处理过源代码后的输出以 file 为文件名。如果没有指定此选项，可执行文件预设的输出文件名为 a.out，而目的文件默认的输出文件名则为 .o 结尾，翻译文件预设的输出文件名则为 .s 结尾，预处理过的 C 语言原始文件则以标准输出设备作输出。
-v	详细地列出 GCC 编译的过程，这些信息将会输出到标准错误输出上。
-pipe	使用管道来取代使用临时文件作沟通。
-E	只将源代码作预处理，然后把结果传给标准输出设备，而非传给编译器。

连接器选项：

-lLIBRARY	在作连接（link）时会到 /lib 及 /usr/lib 目录中查找 libLIBRARY.a 函数库文件。
-Ldir	在作连接（link）时会到 dir 目录中查找函数库文件，再到 /lib 及 /usr/lib 目录中查找。
-Idir	程序中所使用的表头文件目录位置。

-nostartfiles 连接时不使用系统的启始文件。

-s 从执行文件中删除所有符号表和重定位信息，可以使执行文件变小。和程序 strip 作用一样。

与 C 语言相关的选项：

-Dmacro 定义 macro，如同在程序中 #define marco。

-Dmacro=defn 定义 macro 内容为 defn。如同在程序中 #define marco defn。

-Umacro 解除 marco 的宏定义。

-ansi 支援 ANSI C 码标准。一些 GCC 的关键字，如 'asm'，'inline' 和'typeof' 等，与原来 ANSI C 码并不相容，而且也不能在程序中使用 C++ 的 '//' 作为注解。如果源代码不符合 ANSI 标准，程序仍然可以编译成功。

-fno-asm 不使用 'asm'，'inline' 和 'typeof' 关键字，这些关键字可改用'__asm__'，'__inline__' 和 '__typeof__' 取代。 -ansi 也会打开此选项。如果是 C++，由于 'asm' 和 'inline' 为 C++ 关键字，所以此选项只会影响'typeof'。

-traditional 支援传统 C 编译。

-pedantic 以所有 ANSI C 码标准来产生警告信息。

有关警告的选项：

-fsyntax-only 只检查程序的语法错误。

-Wformat 在调用格式化字符串的相关函数，如 printf（），scanf（）等，必须要指定字符串格式的参数，否则会给予警告。近来有许多程序因为未指定字符串格式的参数，造成系统安全的问题。

-Wimplicit-function-declaration

-Werror-implicit-function-declaration

在调用一函数前要先对此定义函数，不然的话此选项将给予警告信息。

-Wparentheses 如果括号会影响到程序流程或造成阅读的困惑时则给予警告信息，例如：

```
{
if（a）
    if（b）
        foo（）;
else
    bar（）;
}
```

这里的 else 应该是对应到 if（b），但这样的写法有时会不小心造成难以检查的错误。

-Wunused　如果声明的参数在程序中并未使用，则给予警告信息。如果想检查传给函数的参数有无使用到，则要同时使用 -W 和 -Wunused 选项。

-Wsign-compare　当程序中试图将声明为 signed 和 unsigned 的值作比较时，signed 值转换成 unsigned 时可能会造成一些错误，此选项会给予警告。-W 也会打开此选项，如果在使用 -W 时想忽略此警告，可使用 -W -Wno-sign-compare。

-w　忽略所有警告信息。

-W　列出更详细的警告信息。例如底下的函数，当此函数不能明确地返回值时则产生警告：

```
foo（a）
{
    if（a>0）
            return a;
}
```

-Wall　对源代码中可能发生问题的部分产生更多的警告信息。

除错及统计选项：

-g　此参数用来产生除错用的扩充符号表，以提供 GDB 使用。

-ggdb　此选项用来产生除错用的扩充符号表，以提供 GDB 使用。

有关优化的选项：

-ffast-math　此选项会令 GCC 采用非 ANSI 或 IEEE 标准来加快浮点运算。例如，GCC 会假设传给 sqrt（）的参数都为正数。优化的 '-O'不会将此选项打开。

-fomit-frame-pointer　尽可能不保留堆栈框指针（frame-pointer）。此选项会令一些平台无法对程序排错。以 Linux 来说，函数一开头会加入底下两行：

```
push %ebp                （保留堆栈框指针）
mov %esp，%ebp           （设置堆栈框指针）
```

离开函数前（ret）会有 leave 指令（还原堆栈框指针）

-fno-inline　忽略 inline 关键字，此选项将抑制所有的 inline 函数展开。通常 inline 函数只有在打开优化选项时才会有效。

-finline-functions　将声明成 inline 的函数展开到调用它的函数里。GCC 会自行决定将那些函数展开于调用它的程序中。inline 函数可以省去调用函数的额外负担。

-O0　此为默认选项，GCC 不会做任何优化的操作，所建立的程序码较能方便排错。除非程序有声明 register 关键字，否则 GCC 也不会主动让寄存器提供变量使用。

-O -O1　进行优化操作，让 GCC 尝试减少程序大小及执行时间。此选项会同时打开 -fthread-jumps 与 -fdefer-pop 选项，其他优化细节选项则视平台决定。

-O2　进行更多的优化操作，GCC 会产生执行速度更快的程序码。此选项会打开所有的优化选项，除了循环展开和 inline 函数展开的相关选项。

-O3　与 -O2 作用相同，只是多打开了 inline 函数展开的选项。

-Os　针对程序大小作优化。此选项会打开所有 -O2 中不会增加程序码的优化选项，尽可能产生最小的程序码。

-fforce-mem　强迫内存运算使用寄存器。

-frerun-loop-opt　使用两次循环优化。

-funroll-loops　对循环执行进行展开的优化操作。此选项会由编译器决定该循环是否值得进行展开的操作。

-funroll-all-loops　对所有循环进行展开的优化操作，当编译器编译一个循环时，如下情况：

```
for (i=0; i<3; i++) {
  a+=b[i];
}
```

将会展开成：

```
a+=b[0];
a+=b[1];
a+=b[2];
```

这将比原来的循环执行更少的指令，但不是所有的循环适合如此，大部分的情况会产生更庞大的执行文件。

宏与函数

宏定义（#define）除了定义常数外，也常作指令的宏定义。如：

```
#define DEBUG   /* 定义 DEBUG，此 DEBUG 不代表任何数值 */
#define MAX     200     /* 定义 MAX 值为 200 */
```

在 C 中还可以使用 const 来定义常数。如：

```
const int max = 50;
```

如果用 gdb 来观察，const 所作的常数声明和一般变量相同，都是使用堆栈，如上一行 const int max = 50；实际上可能为 movl $0x32，0xfffffffc（%ebp），而用 #define 所定义的常数即为真正的常数，如

```
push   $0xc8     /* 0xc8 = 200 = MAX */
```

理论上来说，用 #define 声明常数会较快些。const 所声明的常数其实仍然可以改变，不过在编译时会给予警告（gcc）。

如果用 #define 来定义宏指令可能有几种情况：

```
#define IS_RANGE (x)    (x<127) && (x>0)    /* (1) */
#define each (i)        for (i=0;i<100;i++)  /* (2) */
#define DEBUG (msg)     printf ("%s\n", msg)  /* (3) */
```

然后便可以利用 if（IS_RANGE（i））来使用，这样看来比未使用 #define 好看多了，而且较易理解程序码。第二个例子，我把具有某种意义的 for-loop 用 each 来定义，也让程序较为清楚：

```
each (i)
{
  printf ("%d\n", i) ;
}
```

此外，当发展一些程序时可能会希望打印除错用的信息或字符串，第（3）个例子就蛮好用的，如：

```
void show ()
{
   DEBUG ("Entry void show () ") ;
   .....
}
```

当程序除错完成后可借由重新 #define DEBUG 宏即可取消打印字符串，如：

```
#define DEBUG (msg)      {}
```

这么一来，原来的程序几乎都不用更改，而且如在 void show（）函数内所保留的 DEBUG（"Entry void show（）"）；也算是程序的注解了。

虽然用 #define 来定义程序码宏相当好用，但有一些小地方要留意。像是第一个例子，若是使用 if（IS_RANGE（i++）），此 i++ 经宏展开后会变成两个 i++，而不会是预期的 i 值加 1。还有个例子，如

```
#define printd (d)   if (d > 0) printf ("%d", d)    /* (4) */
```

看来使用起来没什么问题，但遇到下列的程序使用时：

```
if (a > 5)
  printd (d) ;
else
  printf ("a < 5!\n") ;
```

程序码经编译器展开后是下列的模样：

```
if (a > 5)
  if (d > 0)
    printf ("%d", d) ;
else
  printf ("a < 5!\n") ;
```

原来 else 打算和 if（a > 5）对应，现在却是和 if（d > 0）呼应了，这种错误会令人不易发现，却可能是要命的 bug（s）。为避免这种错误的发生，我们可以用下列的方式重新定义：

```
#define printd (d)   if (d > 0) { printf ("%d", d) ; }   /* (5) */
#define printd (d) \
do { if (d > 0) printf ("%d", d) ; } while (0)        /* (6) */
```

利用第（5）个例子将先前的程序展开后如下：

```
if (a > 5)
  if (d > 0)
  {
```

```
        printf ("%d", d) ;
    );
  else
    printf ("a < 5!\n") ;
```

注意到，由于 else 前多了个 "; " ，会令编译器找不到 else 对应的 if ，而给予警告，至少这让编译器告诉我们有这个问题，若要完全解决则要使用第（6）个例子所使用的 do { ..... } while（0）技巧，当程序展开后即为：

```
if (a > 5)
  do {
    if (d > 0)
       printf ("%d", d) ;
  } while (0) ;
else
  printf ("a < 5!\n") ;
```

程序经编译时不再会有错误信息，执行也不会有问题了。在使用这个技巧时刻意将 do {.....} while（0）后面原该接的 ；省略，而且由于 while 内为 0，gcc 在编译时会自动省略此 do-while。

用 #define 定义的程序宏由于减少了参数的传递和避免分支流程，会让程序比使用小而多量的函数调用较为有效率，这让我想到对于 for-loop 什么情况下要 unroll-loop 以空间换取时间的考量。

使用函数除了可避免掉上述所说的宏问题，也具有可返回指针等特性，如果想要同时具有函数及宏的好处则可使用 gcc 扩充的 inline 关键字来定义函数，如果有学过 C++ 对于 inline 应不陌生，如：

```
inline int printd (int d)
{
 if (d > 0) printf ("%d", d) ;
 return d;

}
```

当未使用优化选项编译时，此 inline function 为真正的函数调用，只有在指定优化选项时才会展开。

不定参数

C 语言中有些函数会利用不定参数的方式来取得参数内容，常见的如 printf()、sprintf()、execl() 等等，其最后参数是以 ..... 来定义，即代表后面的参数为不定个数的参数内容。不定参数相关的表头文件在 /usr/lib/bcc/include/stdarg.h 其定义如下：

```
typedef char *va_list;
#define va_start (ap, p)        (ap = (char *) (& (p) +1) )
#define va_arg (ap, type)       ( (type *) (ap += sizeof (type) ) ) [-1]
#define va_end (ap)
```

范 例

```
#include <stdio.h>
#include <stdarg.h>
void fun (char *s , ......) /* ..... 为声明不定个数参数 */
{
  va_list ap;
  int t;    /* 与 main () 中的 a, b, c 相同的类型声明 */
  va_start (ap, s) ;
  printf ("%s", s) ;
  while ( (t = va_arg (ap, int) ) )
    printf ("%d ", t) ;
  va_end (ap) ;
}
main ()
{
  int a = 1, b = 2, c = 3;
  fun ("test: ", a, b, c, NULL) ; /* 以 NULL 作结束 */
  printf ("\n") ;
}
```

执行结果

```
test: 1 2 3
```

Linux信号列表

编　号	名　称	操　作	简单说明
1	SIGHUP	A	当终端机察觉到终止连线操作时便会传送这个信号
2	SIGINT	A	当由键盘要求某个中断的方式时，如 CTRL+C，则会产生此信号
3	SIGQUIT	A	当由键盘要求停止执行时，如 CTRL+\，则会产生此信号
4	SIGILL	A	进程执行了一个不合法的 CPU 指令
5	SIGTRAP	CG	当子进程因被追踪而暂停时便产生此信号给父进程
6	SIGABRT	C	调用 abort（）时会产生此信号
7	SIGBUS	AG	Bus 发生错误时会产生此信号
8	SIGFPE	C	进行数值运算时发生了例外的状况，如除以 0
9	SIGKILL	AEF	终止进程的信号，此信号不能被拦截或忽略
10	SIGUSR1	A	供用户自定的信号
11	SIGSEGV	C	试图存取不被允许的内存地址
12	SIGUSR2	A	供用户自定的信号
13	SIGPIPE	A	错误的管道；欲写入无读取端的管道时便产生此信号
14	SIGALRM	A	利用 alarm（）所设置的时间到时就产生此信号
15	SIGTERM	A	终止进程
16	SIGSTKFLT	AG	堆栈错误
17	SIGCHLD	B	子进程暂停或结束时便会传送此信号给父进程
17	SIGCLD		和 SIGCHLD 相同
18	SIGCONT		此信号会让暂停的进程继续执行
19	SIGSTOP	DEF	此信号用来让进程暂停执行，此信号不能被拦截或忽略
20	SIGTSTP	D	当由键盘（CTRL+Z）表示暂停时就产生此信号
21	SIGTTIN	D	背景进程欲从终端机读取数据时便产生此信号
22	SIGTTOU	D	背景进程欲写入数据至终端机时便产生此信号

（续）

编　号	名　称	操　作	简单说明
23	SIGURG	BG	socket 的紧急状况发生时便产生此信号
24	SIGXCPU	AG	当进程超过可用的 CPU 时间时便产生此信号
25	SIGXFSZ	AG	当进程的文件大小超过系统限制时便产生此信号
26	SIGVTALRM	AG	利用 setitimer（）所设置的虚拟计时到达
27	SIGPROF	AG	利用 setitimer（）设置的间隔计时器到达
28	SIGWINCH	BG	当视窗大小改变时便产生此信号
29	SIGIO	AG	发生了一个非同步事件
29	SIGPOLL		和 SIGIO 相同
30	SIGPWR	AG	主机电源不稳时则产生此信号
31	SIGUNUSED		未使用

操作符号所代表的意义：

A　默认为终止进程。

B　默认为忽略此信号。

C　默认为内存倾卸（core dump）。

D　默认为暂停进程执行。

E　此信号不可拦截。

F　此信号不可忽略。

G　此信号非 POSIX 标准。

常见错误代码及原因

错误代码	原 因
EACCESS	存取操作被拒绝，权限不足
EAFNOSUPPORT	网络地址类型不支持
EAGAIN	欲使用不可阻断 I/O 而无数据可读取，或系统希望稍后再调用
EALREADY	socket 为不可阻断且先前的连线操作还未完成
EBADF	文件描述词为无效的或该文件已关闭
EBUSY	试图卸下正在使用的文件系统
ECHILD	指定的进程代码不存在
ECONNREFUSED	连线要求遭 server 端拒绝
EDOM	参数所代表的定义域不正确（数学函数）
EEXIST	欲建立新文件，该文件却已存在
EFAULT	欲传给系统调用的参数为一无效指针，指向无法存取的内存空间
EFBIG	欲写入的文件大小超过该文件系统所能处理的范围
EINTR	此调用被信号所中断
EINVAL	传给系统调用的参数不合法
EIO	I/O 存取错误
EISCONN	socket 已经是连线状态
EISDIR	参数指向一目录，欲调用的函数不适用于处理目录
ELOOP	欲打开的文件有过多符号连接问题，上限为 16 个符号连接
EMFILE	已达到进程可同时打开的文件数上限
EMLINK	文件连接数已超过文件系统的最大值
ENAMETOOLONG	路径名称太长
ENETUNREACH	无法传送封包至指定的主机
ENFILE	已达到系统可同时打开的文件数上限
ENOBUFS	系统的缓冲内存不足
ENOENT	指定的文件不存在
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
ENOMEM	核心内存不足

（续）

错误代码	原　因
ENOSPC	文件系统的剩余空间不足
ENOTDIR	路径中的目录存在但却非真正的目录
ENOTSOCK	错将文件描述词视作 socket 处理
ENXIO	指定的设备或地址不存在
EOPNOTSUPP	指定的 socket 并未支持相关函数的操作
EPERM	进程的权限不够，无法完成所调用的函数操作
EROFS	欲打开写入权限的文件存在于只读文件系统内
ERANGE	计算结果超出合法范围
ESPIPE	文件描述词所指的文件为一管道（pipe）
ESRCH	参数指定的进程或进程组不存在
ETIMEDOUT	企图连线的操作超过限定时间仍未有回应
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
E2BIG	参数数组过大