# Linux C函数详解(总结)

# 参考资料

1.Linux程序设计(第4版)Neil Matthew

2.UNIX环境高级编程,[美]理查德·史蒂文斯

# 格式化I/O

## printf(格式化输出数据)

|  |  |
| --- | --- |
| **printf(格式化输出数据)** | |
| 相关函数 | scanf,snprintf |
| 表头文件 | #include <stdio.h> |
| 定义函数 | int printf(const char \*format,.............); |
| 函数说明 | printf()会根据参数format字符串来转换并格式化数据,然后将结果写出到标准输出设备,直到出现字符串结束(’\0’)为止。  参数format字符串可包含下列三种字符类型：  1.一般文本,伴随直接输出。  2.ASCII控制字符,如\t、\n等。  3.格式转换字符。  格式转换为一个百分比符号(％)及其后的格式字符所组成。一般而言,每个％符号在其后都必需有一printf()的参数与之相呼应(只有当％％转换字符出现时会直接输出％字符),而欲输出的数据类型必须与其相对应的转换字符类型相同。  printf()格式转换的一般形式如下：  ％〔flags〕〔width〕〔.prec〕type  以中括号括起来的参数为选择性参数,而％与type则是必要的。底下先介绍type的几种形式：  **整数：**  ％d 整数的参数会被转成一有符号的十进制数字  ％u 整数的参数会被转成一无符号的十进制数字  ％o 整数的参数会被转成一无符号的八进制数字  ％x 整数的参数会被转成一无符号的十六进制数字,并以小写abcdef表示  ％X 整数的参数会被转成一无符号的十六进制数字,并以大写ABCDEF表示  **浮点型数：**  ％f double型的参数会被转成十进制数字,并取到小数点以下六位,四舍五入。  ％e double型的参数以指数形式打印,有一个数字会在小数点前,六位数字在小数点后,而在指数部分会以小写的e来表示。  ％E 与％e作用相同,唯一区别是指数部分将以大写的E来表示。  ％g double型的参数会自动选择以％f或％e的格式来打印,其标准是根据欲打印的数值及所设置的有效位数来决定。  ％G 与％g作用相同,唯一区别在以指数形态打印时会选择％E格式。  **字符及字符串：**  ％c 整型数的参数会被转成unsigned char型打印出。  ％s 指向字符串的参数会被逐字输出,直到出现NULL字符为止  ％p 如果是参数是“void \*”型指针则使用十六进制格式显示。  **prec有几种情况：**  1.正整数的最小位数。  2.在浮点型数中代表小数位数  3.在％g格式代表有效位数的最大值。  4.在％s格式代表字符串的最大长度。  5.若为×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,整数则不加任何负号。此旗标会使得在打印  正数前多一个正号(+)。  # 此旗标会根据其后转换字符的不同而有不同含义。当在类型为o之前(如％#o),则会在打  印八进制数值前多印一个o。而在类型为x之前(％#x)则会在打印十六进制数前多印’0x’,  在型态为e、E、f、g或G之前则会强迫数值打印小数点。在类型为g或G之前时则同  时保留小数点及小数位数末尾的零。  0 当有指定参数时,无数字的参数将补上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同意义\*/  } | |
| 执行 |  |
| -100 3.14159 96  150 150 | |

## sacnf(格式化字符串输入)

|  |  |
| --- | --- |
| **sacnf(格式化字符串输入)** | |
| 相关函数 | fscanf,snprintf |
| 表头文件 | #include <stdio.h> |
| 定义函数 | int scanf(const char \*format,.......); |
| 函数说明 | scanf()会将输入的数据根据参数format字符串来转换并格式化数据。scanf()格式转换的一般形式如下：  ％[\*][size][l][h]type  以中括号括起来的参数为选择性参数,而％与type则是必要的。  \*代表该对应的参数数据忽略不保存。  size 为允许参数输入的数据长度。  l 输入的数据数值以long int 或double型保存。  h 输入的数据数值以short int 型保存。  底下介绍type的几种形式：  ％d 输入的数据会被转成一有符号的十进制数字(int)。  ％i 输入的数据会被转成一有符号的十进制数字,若输入数据以“0x”或“0X”开头代表转换十六进制数字,若以“0”开头则转换八进制数字,其他情况代表十进制。  ％0 输入的数据会被转换成一无符号的八进制数字。  ％u 输入的数据会被转换成一无符号的正整数。  ％x 输入的数据为无符号的十六进制数字,转换后存于unsigned int型变量。  ％X 同％x  ％f 输入的数据为有符号的浮点型数,转换后存于float型变量。  ％e 同％f  ％E 同％f  ％g 同％f  ％s 输入数据为以空格字符为终止的字符串。  ％c 输入数据为单一字符。  [] 读取数据但只允许括号内的字符。如[a-z]。  [^] 读取数据但不允许中括号的^符号后的字符出现,如[^0-9]. |
| 返回值 | 成功则返回参数数目,失败则返回-1,错误原因存于errno中。 |
| 范例 |  |
| #include <stdio.h>  main(){  int i;  unsigned int j;  char s[5];  scanf(“%d %x %5[a-z] %\*s %f”,&i,&j,s,s);  printf(“%d %d %s\n”,i,j,s);  } | |
| 执行 |  |
| 10 0x1b aaaaaaaaaa bbbbbbbbbb  10 27 aaaaa | |

## fprintf(格式化输出数据至文件)

|  |  |
| --- | --- |
| **fprintf(格式化输出数据至文件)** | |
| 相关函数 | printf,fscanf,vfprintf |
| 表头文件 | #include <stdio.h> |
| 定义函数 | int fprintf(FILE \*stream, const char \*format,.......); |
| 函数说明 | fprintf()会根据参数format字符串来转换并格式化数据,然后将结果输出到参数stream指定的文件中,直到出现字符串结束(‘\0‘)为止。 |
| 返回值 | 关于参数format字符串的格式请参考printf()。成功则返回实际输出的字符数,失败则返回-1,错误原因存于errno中。 |
| 范例 |  |
| #include <stdio.h>  main(){  int i = 150;  int j = -100;  double k = 3.14159;  fprintf(stdout,”%d %f %x \n”,j,k,i);  fprintf(stdout,”%2d %\*d\n”,i,2,i);  } | |
| 执行 |  |
| -100 3.141590 96  150 150 | |

## fscanf(格式化字符串输入)

|  |  |
| --- | --- |
| **fscanf(格式化字符串输入)** | |
| 相关函数 | scanf,sscanf |
| 表头文件 | #include <stdio.h> |
| 定义函数 | int fscanf(FILE \*stream ,const char \*format,....); |
| 函数说明 | fscanf()会自参数stream的文件流中读取字符串,再根据参数format字符串来转换并格式化数据。  格式转换形式请参考scanf()。转换后的结构存于对应的参数内。 |
| 返回值 | 成功则返回参数数目,失败则返回-1,错误原因存于errno中。 |
| 附加说明 |  |
| 范例 |  |
| #include <stdio.h>  main(){  int i;  unsigned int j;  char s[5];  fscanf(stdin,”%d %x %5[a-z] %\*s %f”,&i,&j,s,s);  printf(“%d %d %s \n”,i,j,s);  } | |
| 执行 |  |
| 10 0x1b aaaaaaaaa bbbbbbbbbb /\*从键盘输入\*/  10 27 aaaaa | |

## sprintf(格式化字符串复制)

|  |  |
| --- | --- |
| **sprintf(格式化字符串复制)** | |
| 相关函数 | printf,sprintf |
| 表头文件 | #include <stdio.h> |
| 定义函数 | int sprintf( char \*str,const char \*format,.........); |
| 函数说明 | sprintf()会根据参数format字符串来转换并格式化数据,然后将结果复制到参数str所指的字符串数组,直到出现字符串结束(’\0’)为止。关于参数format字符串的格式请参考printf()。 |
| 返回值 | 成功则返回参数str字符串长度,失败则返回-1,错误原因存于errno中。 |
| 附加说明 | 使用此函数得留意堆栈溢出,或改用snprintf()。 |
| 范例 |  |
| #include <stdio.h>  main(){  char \*a=”This is string A!”;  char buf[80];  sprintf(buf,”>>> %s<<<\n”,a);  printf(“%s”.buf);  } | |
| 执行 |  |
| >>>This is string A!<<< | |

## sscanf(格式化字符串输入)

|  |  |
| --- | --- |
| **sscanf(格式化字符串输入)** | |
| 相关函数 | scanf,fscanf |
| 表头文件 | #include <stdio.h> |
| 定义函数 | int sscanf(const char \*str,const char \*format,........); |
| 函数说明 | sscanf()会将参数str的字符串根据参数format字符串来转换并格式化数据。格式转换形式请参考scanf()。转换后的结果存于对应的参数内。 |
| 返回值 | 成功则返回参数数目,失败则返回-1,错误原因存于errno中。 |
| 范例 |  |
| #include <stdio.h>  main(){  int i;  unsigned int j;  char input[ ]=”10 0x1b aaaaaaaa bbbbbbbb”;  char s[5];  sscanf(input,”%d %x %5[a-z] %\*s %f”,&i,&j,s,s);  printf(“%d %d %s\n”,i,j,s);  } | |
| 执行 |  |
| 10 27 aaaaa | |

## vfprintf(格式化输出数据至文件)

|  |  |
| --- | --- |
| **vfprintf(格式化输出数据至文件)** | |
| 相关函数 | printf,fscanf,fprintf |
| 表头文件 | #include <stdio.h>  #include <stdarg.h> |
| 定义函数 | int vfprintf(FILE \*stream,const char \*format,va\_list ap); |
| 函数说明 | vfprintf()会根据参数format字符串来转换并格式化数据,然后将结果输出到参数stream指定的文件中,直到出现字符串结束(’\0’)为止。关于参数format字符串的格式请参考printf()。va\_list用法请参考附录C或vprintf()范例。 |
| 返回值 | 成功则返回实际输出的字符数,失败则返回­-1,错误原因存于errno中。 |
| 范例 |  |
| 参考fprintf()及vprintf()。 | |

## vfscanf(格式化字符串输入)

|  |  |
| --- | --- |
| **vfscanf(格式化字符串输入)** | |
| 相关函数 | scanf,sscanf,fscanf |
| 表头文件 | #include <stdio.h> |
| 定义函数 | int vfscanf(FILE \*stream,const char \*format ,va\_list ap); |
| 函数说明 | vfscanf()会自参数stream的文件流中读取字符串,再根据参数format字符串来转换并格式化数据。格式转换形式请参考scanf()。转换后的结果存于对应的参数内。va\_list用法请参考附录C或vprintf()。 |
| 返回值 | 成功则返回参数数目,失败则返回-1,错误原因存于errno中。 |
| 范例 |  |
| 参考fscanf()及vprintf()。 | |

## vprintf(格式化输出数据)

|  |  |
| --- | --- |
| **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);  } | |
| 执行 |  |
| my\_printf() : -100 3.14159 96  my\_printf() : 150 150 | |

## vscanf(格式化字符串输入)

|  |  |
| --- | --- |
| **vscanf(格式化字符串输入)** | |
| 相关函数 | vsscanf,vfscanf |
| 表头文件 | #include <stdio.h>  #include <stdarg.h> |
| 定义函数 | int vscanf( const char \*format,va\_list ap); |
| 函数说明 | vscanf()会将输入的数据根据参数format字符串来转换并格式化数据。格式转换形式请参考scanf()。转换后的结果存于对应的参数内。va\_list用法请参考附录C或vprintf()范例。 |
| 返回值 | 成功则返回参数数目,失败则返回-1,错误原因存于errno中。 |
| 范例 |  |
| 请参考scanf()及vprintf()。 | |

## vsprintf(格式化字符串复制)

|  |  |
| --- | --- |
| **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)

|  |  |
| --- | --- |
| **vsscanf(格式化字符串输入)** | |
| 相关函数 | vscanf,vfscanf |
| 表头文件 | #include <stdio.h> |
| 定义函数 | int vsscanf(const char \*str,const char \*format,va\_list ap); |
| 函数说明 | vsscanf()会将参数str的字符串根据参数format字符串来转换并格式化数据。格式转换形式请参考附录C或vprintf()范例。 |
| 返回值 | 成功则返回参数数目,失败则返回-1,错误原因存于errno中。 |
| 范例 |  |
| 请参考sscanf()及vprintf()。 | |

# 错误处理

## ferror(检查文件流是否有错误发生)

|  |  |
| --- | --- |
| **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)

|  |  |
| --- | --- |
| **perror(打印出错误原因信息字符串)** | |
| 相关函数 | strerror |
| 表头文件 | #include <stdio.h> |
| 定义函数 | void perror(const char \*s); |
| 函数说明 | perror()用来将上一个函数发生错误的原因输出到标准错误(stderr)。参数s所指的字符串会先打印出,后面再加上错误原因字符串。此错误原因依照全局变量errno的值来决定要输出的字符串。 |
| 返回值 | 无 |
| 范例 |  |
| #include <stdio.h>  main(){  FILE \*fp;  fp = fopen(“/tmp/noexist”,”r+”);  if(fp = =NULL) perror(“fopen:”);  } | |
| 执行 |  |
| $ ./perror  fopen : No such file or diretory | |

## strerror(返回错误原因的描述字符串)

|  |  |
| --- | --- |
| **strerror(返回错误原因的描述字符串)** | |
| 相关函数 | perror |
| 表头文件 | #include <string.h> |
| 定义函数 | char \*strerror(int errnum); |
| 函数说明 | strerror()用来依参数errnum的错误代码来查询其错误原因的描述字符串,然后将该字符串指针返回。 |
| 返回值 | 返回描述错误原因的字符串指针。 |
| 范例 |  |
| /\*显示错误代码0至9的错误原因描述 \*/  #include <string.h>  main(){  int i;  for(i=0;i<10;i++)  printf(“%d : %s\n”,i,strerror(i));  } | |
| 执行 |  |
| 0 : Success  1 : Operation not permitted  2 : No such file or directory  3 : No such process  4 : Interrupted system call  5 : Input/output error  6 : Device not configured  7 : Argument list too long  8 : Exec format error  9 : Bad file descriptor | |

# 字符测试

## isalnum(测试字符是否为英文或数字)

|  |  |
| --- | --- |
| **isalnum(测试字符是否为英文或数字)** | |
| 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 |  |
| 1 is an apphabetic character  2 is an apphabetic character  3 is an apphabetic character  c is an apphabetic character  F is an apphabetic character  D is an apphabetic character  s is an apphabetic character  P is an apphabetic character  e is an apphabetic character | |

## isalpha(测试字符是否为英文字母)

|  |  |
| --- | --- |
| **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 alphanumeric character\n”,str[i]);  } | |
| 执行 |  |
| c is an apphabetic character  F is an apphabetic character  D is an apphabetic character  s is an apphabetic character  P is an apphabetic character  e is an apphabetic character | |

## isascii(测试字符是否为ASCII码字符)

|  |  |
| --- | --- |
| **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 |  |
| 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 | |

## iscntrl(测试字符是否为ASCII码的控制字符)

|  |  |
| --- | --- |
| **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30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)

|  |  |
| --- | --- |
| **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[]="123@#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 |  |
| 1 is an digit character  2 is an digit character  3 is an digit character | |

## isxdigit(测试字符是否为16进制数字)

|  |  |
| --- | --- |
| **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：0123456789ABCDEF |
| 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 |  |
| /\*找出字符串str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]);  } | |
| 执行 |  |

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

## isgraph(测试字符是否为可打印字符)

|  |  |
| --- | --- |
| **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字符串中哪些为可打印字符 \*/  #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 |  |
| str[0] is printable character:a  str[1] is printable character:5  str[3] is printable character:@  str[4] is printable character:; | |

## isprint(测试字符是否为可打印字符)

|  |  |
| --- | --- |
| **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字符串中哪些为可打印字符 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]);  } | |
| 执行 |  |
| 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)

|  |  |
| --- | --- |
| **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 | int isspace(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(’’)、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 |  |
| 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 /\*\t \*/ | |

## ispunct(测试字符是否为标点符号或特殊符号)

|  |  |
| --- | --- |
| **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 | #inlude<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 |  |
| @#[? | |

## islower(测试字符是否为小写字母)

|  |  |
| --- | --- |
| **islower(测试字符是否为小写字母)** | |
| 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 |  |
| #include <ctype.h>  main(){  char str[]="123@#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 |  |
| c is a lower-case character  s is a lower-case character  e is a lower-case character | |

## isupper(测试字符是否为大写英文字母)

|  |  |
| --- | --- |
| **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 |  |
| /\*找出字符串str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 an uppercase character\n",str[i]);  } | |
| 执行 |  |
| F is an uppercase character  D is an uppercase character  P is an uppercase character | |

# 字符(串)转换

## atoi(将字符串转换成整型数)

|  |  |
| --- | --- |
| **atoi(将字符串转换成整型数)** | |
| 相关函数 | atof,atoll,atrtod,strtol,strtoul |
| 表头文件 | #include <stdlib.h> |
| 定义函数 | int atoi(const char \*nptr); |
| 函数说明 | atoi()会扫描参数nptr字符串,跳过前面的空格字符,直到遇上数字或正负符号才开始做转换,而再遇到非数字或字符串结束时(’\0’)才结束转换,并将结果返回。 |
| 返回值 | 返回转换后的整型数。 |
| 附加说明 | atoi()与使用strtol(nptr,(char\*\*)NULL,10)结果相同。也可以使用sscanf()。 |
| 范例 |  |
| /\*将字符串a与字符串b转换成数字后相加 \*/  #include <stdlib.h>  mian(){  char a[]=”-100”;  char b[]=”456”;  int c;  c=atoi(a)+atoi(b);  printf(c=%d\n”,c);  } | |
| 执行 |  |
| c=356 | |

## atol(将字符串转换成长整型数)

|  |  |
| --- | --- |
| **atol(将字符串转换成长整型数)** | |
| 相关函数 | atof,atoi,strtod,strtol,strtoul |
| 表头文件 | #include <stdlib.h> |
| 定义函数 | long atol(const char \*nptr); |
| 函数说明 | atol()会扫描参数nptr字符串,跳过前面的空格字符,直到遇上数字或正负符号才开始做转换,而再遇到非数字或字符串结束时(’\0’)才结束转换,并将结果返回。 |
| 返回值 | 返回转换后的长整型数。 |
| 附加说明 | atol()与使用strtol(nptr,(char\*\*)NULL,10)；结果相同。 |
| 范例 |  |
| /\*将字符串a与字符串b转换成数字后相加\*/  #include <stdlib.h>  main(){  char a[]=”1000000000”;  char b[]=” 234567890”;  long c;  c=atol(a)+atol(b);  printf(“c=%d\n”,c);  } | |
| 执行 |  |
| c=1234567890 | |

## atof(将字符串转换成浮点型数)

|  |  |
| --- | --- |
| **atof(将字符串转换成浮点型数)** | |
| 相关函数 | atoi,atol,atof,strtod,strtol,strtoul |
| 表头文件 | #include <stdlib.h> |
| 定义函数 | double atof(const char \*nptr); |
| 函数说明 | atof()会扫描参数nptr字符串,跳过前面的空格字符,直到遇上数字或正负符号才开始做转换,而再遇到非数字或字符串结束时(’\0’)才结束转换,并将结果返回。参数nptr字符串可包含正负号、小数点或E(e)来表示指数部分,如123.456或123e-2。 |
| 返回值 | 返回转换后的浮点型数。 |
| 附加说明 | atof()与使用strtod(nptr,(char\*\*)NULL)；结果相同。 |
| 范例 |  |
| /\*将字符串a与字符串b转换成数字后相加\*/  #include <stdlib.h>  main(){  char \*a=”-100.23”;  char \*b=”200e-2”;  float c;  c=atof(a)+atof(b);  printf(“c=%.2f\n”,c);  } | |
| 执行 |  |
| c=-98.23 | |

## toascii(将整型数转换成合法的ASCII码字符)

|  |  |
| --- | --- |
| **toascii(将整型数转换成合法的ASCII码字符)** | |
| 相关函数 | isascii,toupper,tolower |
| 表头文件 | #include <ctype.h> |
| 定义函数 | int toascii(int c) |
| 函数说明 | toascii()会将参数c转换成7位的unsigned char值,第八位则会被清除,此字符即会被转成ASCII码字符。 |
| 返回值 | 将转换成功的ASCII码字符值返回。 |
| 范例 |  |
| #include <stdlib.h>  main(){  int a=217;  char b;  printf(“before toascii() : a value =%d(%c)\n”,a,a);  b=toascii(a);  printf(“after toascii() : a value =%d(%c)\n”,b,b);  } | |
| 执行 |  |
| before toascii() : a value =217()  after toascii() : a value =89(Y)  注释：如果a的值在0-127之间,则会转换成该值对应的ascii码字符,如果a的值不在这个范围内,则会转换失真,如上述a=217转换成了Y -- 截短,只取其低7位的值。 | |

## gcvt(将浮点型数转换为字符串,取四舍五入)

|  |  |
| --- | --- |
| **gcvt(将浮点型数转换为字符串,取四舍五入)** | |
| 相关函数 | ecvt,fcvt,sprintf |
| 表头文件 | #include <stdlib.h> |
| 定义函数 | char \*gcvt(double number,size\_t ndigits,char \*buf); |
| 函数说明 | gcvt()用来将参数number转换成ASCII码字符串,参数ndigits表示显示的位数。gcvt()与ecvt()和fcvt()不同的地方在于,gcvt()所转换后的字符串包含小数点或正负符号。若转换成功,转换后的字符串会放在参数buf指针所指的空间。 |
| 返回值 | 返回一字符串指针,此地址即为buf指针。 |
| 附加说明 |  |
| 范例 |  |
| #include <stdlib.h>  main(){  double a=123.45;  double b=-1234.56;  char \*ptr;  int decpt,sign;  gcvt(a,5,ptr);  printf(“a value=%s\n”,ptr);  ptr=gcvt(b,6,ptr);  printf(“b value=%s\n”,ptr);  } | |
| 执行 |  |
| a value=123.45  b value=-1234.56 | |

## strtod(将字符串转换成浮点数)

|  |  |
| --- | --- |
| **strtod(将字符串转换成浮点数)** | |
| 相关函数 | atoi,atol,strtod,strtol,strtoul |
| 表头文件 | #include <stdlib.h> |
| 定义函数 | double strtod(const char \*nptr,char \*\*endptr,int base); |
| 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 | 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>  mian(){  char a[]=”1000000000”;  char b[]=”1000000000”;  char c[]=”ffff”;  printf(“a=%d\n”,strtod(a,NULL,10));  printf(“b=%d\n”,strtod(b,NULL,2));  printf(“c=%d\n”,strtod(c,NULL,16));  } | |
| 执行 |  |
| a=1000000000  b=512  c=65535 | |

## strtol(将字符串转换成长整型数)

|  |  |
| --- | --- |
| **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,10));  printf(“b=%d\n”,strtol(b,NULL,2));  printf(“c=%d\n”,strtol(c,NULL,16));  } | |
| 执行 |  |
| a=1000000000  b=512  c=65535 | |

## strtoul(将字符串转换成无符号长整型数)

|  |  |
| --- | --- |
| **strtoul(将字符串转换成无符号长整型数)** | |
| 相关函数 | atof,atoi,atol,strtod,strtol |
| 表头文件 | #include <stdlib.h> |
| 定义函数 | unsigned long int strtoul(const char \*nptr,char \*\*endptr,int base); |
| 函数说明 | strtoul()会将参数nptr字符串根据参数base来转换成无符号的长整型数。参数base范围从2至36,或0。参数base代表采用的进制方式,如base值为10则采用10进制,若base值为16则采用16进制数等。当base值为0时则是采用10进制做转换,但遇到如’0x’前置字符则会使用16进制做转换。一开始strtoul()会扫描参数nptr字符串,跳过前面的空格字符串,直到遇上数字或正负符号才开始做转换,再遇到非数字或字符串结束时(’\0’)结束转换,并将结果返回。若参数endptr不为NULL,则会将遇到不合条件而终止的nptr中的字符指针由endptr返回。 |
| 返回值 | 返回转换后的长整型数,否则返回ERANGE并将错误代码存入errno中。 |
| 附加说明 | ERANGE指定的转换字符串超出合法范围。 |
| 范例 |  |
| 参考strtol() | |

## tolower(将大写字母转换成小写字母)

|  |  |
| --- | --- |
| **tolower(将大写字母转换成小写字母)** | |
| 相关函数 | isalpha,toupper |
| 表头文件 | #include <stdlib.h> |
| 定义函数 | int tolower(int c); |
| 函数说明 | 若参数c为大写字母则将该对应的小写字母返回。 |
| 返回值 | 返回转换后的小写字母,若不须转换则将参数c值返回。 |
| 附加说明 |  |
| 范例 |  |
| /\*将s字符串内的大写字母转换成小写字母\*/  #include <ctype.h>  main(){  char s[]=”aBcDeFgH12345;!#$”;  int i;  printf(“before tolower() : %s\n”,s);  for(i=0;i<sizeof(s);i++)  s[i]=tolower(s[i]);  printf(“after tolower() : %s\n”,s);  } | |
| 执行 |  |
| before tolower() : aBcDeFgH12345;!#$  after tolower() : abcdefgh12345;!#$ | |

## toupper(将小写字母转换成大写字母)

|  |  |
| --- | --- |
| **toupper(将小写字母转换成大写字母)** | |
| 相关函数 | isalpha,tolower |
| 表头文件 | #include <ctype.h> |
| 定义函数 | int toupper(int c); |
| 函数说明 | 若参数c为小写字母则将该对映的大写字母返回。 |
| 返回值 | 返回转换后的大写字母,若不须转换则将参数c值返回。 |
| 附加说明 |  |
| 范例 |  |
| /\*将s字符串内的小写字母转换成大写字母 \*/  #include <ctype.h>  main(){  char s[]=”aBcDeFgH12345;!#$”;  int i;  printf(“before toupper() : %s\n”,s);  for(i=0;I<sizeof(s);i++)  s[i]=toupper(s[i]);  printf(“after toupper() : %s\n”,s);  } | |
| 执行 |  |
| before toupper() : aBcDeFgH12345;!#$  after toupper() : ABCDEFGH12345;!#$ | |

# 内存控制

## malloc(配置内存空间)

|  |  |
| --- | --- |
| **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\*/ | |

## calloc(配置内存空间)

|  |  |
| --- | --- |
| **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(ptr);  } | |

## free(释放原先配置的内存)

|  |  |
| --- | --- |
| **free(释放原先配置的内存)** | |
| 相关函数 | malloc,calloc,realloc,brk |
| 表头文件 | #include <stdlib.h> |
| 定义函数 | void free(void \*ptr); |
| 函数说明 | 参数ptr为指向先前由malloc()、calloc()或realloc()所返回的内存指针。调用free()后ptr所指的内存空间便会被收回。假若参数ptr所指的内存空间已被收回或是未知的内存地址,则调用free()可能会有无法预期的情况发生。若参数ptr为NULL,则free()不会有任何作用。 |

## getpagesize(取得内存分页大小)

|  |  |
| --- | --- |
| **getpagesize(取得内存分页大小)** | |
| 相关函数 | sbrk |
| 表头文件 | #include <unistd.h> |
| 定义函数 | size\_t getpagesize(void); |
| 函数说明 | 返回一分页的大小,单位为字节(byte)。此为系统的分页大小,不一定会和硬件分页大小相同。 |
| 返回值 | 内存分页大小。 |
| 附加说明 | 在Intel x86上其返回值应为4096bytes。 |
| 范例 |  |
| #include <unistd.h>  main(){  printf(“page size = %d\n”,getpagesize( ) );  } | |

## mmap(建立内存映射)

|  |  |
| --- | --- |
| **mmap(建立内存映射)** | |
| 相关函数 | munmap,open |
| 表头文件 | #include <unistd.h>  #include <sys/mman.h> |
| 定义函数 | void \*mmap(void \*start,size\_t length,int prot,int flags,int fd,off\_t offsize); |
| 函数说明 | mmap()用来将某个文件内容映射到内存中,对该内存区域的存取即是直接对该文件内容的读写。  参数start 指向欲对应的内存起始地址,通常设为NULL,代表让系统自动选定地址,对应成功后该地址会返回。参数length代表将文件中多大的部分对应到内存。参数prot代表映射区域的保护方式,有下列组合：  PROT\_EXEC 映射区域可被执行  PROT\_READ 映射区域可被读取  PROT\_WRITE 映射区域可被写入  PROT\_NONE 映射区域不能存取  参数flags会影响映射区域的各种特性：  MAP\_FIXED 如果参数start所指的地址无法成功建立映射时,则放弃映射,不对地址做修  正。通常不鼓励用此旗标。  MAP\_SHARED对映射区域的写入数据会复制回文件内,而且允许其他映射该文件的进程  共享。  MAP\_PRIVATE 对映射区域的写入操作会产生一个映射文件的复制,即私人的“写入时复  制”(copy on write)对此区域作的任何修改都不会写回原来的文件内容。  MAP\_ANONYMOUS建立匿名映射。此时会忽略参数fd,不涉及文件,而且映射区域无法和  其他进程共享。  MAP\_DENYWRITE只允许对映射区域的写入操作,其他对文件直接写入的操作将会被拒绝。  MAP\_LOCKED 将映射区域锁定住,这表示该区域不会被置换(swap)。  在调用mmap()时必须要指定MAP\_SHARED或MAP\_PRIVATE。参数fd为open()返回的文件描述符,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符  EACCES 存取权限有误。如果是MAP\_PRIVATE情况下文件必须  可读,使用MAP\_SHARED则要有PROT\_WRITE以及该文件要能写入。  EINVAL 参数 start、length或offset有一个不合法。  EAGAIN 文件被锁住,或是有太多内存被锁住。  ENOMEM 内存不足。 |
| 范例 |  |
| /\*利用mmap()来读取/etc/passwd文件内容\*/  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <unistd.h>  #include <sys/mman.h>  main(){  int fd;  void \*start;  struct stat sb;  fd=open(“/etc/passwd”,O\_RDONLY); /\*打开/etc/passwd\*/  fstat(fd,&sb); /\*取得文件大小\*/  start=mmap(NULL,sb.st\_size,PROT\_READ,MAP\_PRIVATE,fd,0);  if(start= = MAP\_FAILED) /\*判断是否映射成功\*/  return;  printf(“%s”,start);  munma(start,sb.st\_size); /\*解除映射\*/  closed(fd);  } | |
| 执行 |  |
| root : x : 0 : root : /root : /bin/bash  bin : x : 1: 1 : bin : /bin :  daemon : x : 2 : 2 :daemon : /sbin  adm : x : 3 : 4 : adm : /var/adm :  lp : x :4 :7 : lp : /var/spool/lpd :  sync : x : 5 : 0 : sync : /sbin : bin/sync :  shutdown : x : 6 : 0 : shutdown : /sbin : /sbin/shutdown  halt : x : 7 : 0 : halt : /sbin : /sbin/halt  mail : x : 8 : 12 : mail : /var/spool/mail :  news : x :9 :13 : news : /var/spool/news :  uucp : x :10 :14 : uucp : /var/spool/uucp :  operator : x : 11 : 0 :operator : /root:  games : x : 12 :100 : games :/usr/games:  gopher : x : 13 : 30 : gopher : /usr/lib/gopher-data:  ftp: x :14 :50 : FTP User: /home/ftp:  nobody : x :99: 99: Nobody : /:  xfs :x :100 :101 : X Font Server : /etc/xll/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)

|  |  |
| --- | --- |
| **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符时不会解除映射。 |
| 返回值 | 如果解除映射成功则返回0,否则返回－1,错误原因存于errno中 |
| 错误代码 | EINVAL 参数start或length不合法。 |
| 范例 |  |
| 参考mmap() | |

# 内存及字符串操作

## bcmp(比较内存内容)

|  |  |
| --- | --- |
| **bcmp(比较内存内容)** | |
| 相关函数 | bcmp,strcasecmp,strcmp,strcoll,strncmp,strncasecmp |
| 表头文件 | #include <string.h> |
| 定义函数 | int bcmp( const void \*s1,const void \*s2,int n); |
| 函数说明 | bcmp()用来比较s1和s2所指的内存区间前n个字节,若参数n为0,则返回0。 |
| 返回值 | 若参数s1和s2所指的内存内容都完全相同则返回0值,否则返回非零值。 |
| 附加说明 | 建议使用memcmp()取代。 |
| 范例 |  |
| 参考memcmp()。 | |

## bcopy(拷贝内存内容)

|  |  |
| --- | --- |
| **bcopy(拷贝内存内容)** | |
| 相关函数 | memccpy,memcpy,memmove,strcpy,ctrncpy |
| 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指针放在钱\*/  printf(‘\nmemcpy() : “);  for(i=0;i<30;i++)  printf(“%c”,dest[i]);  } | |
| 执行 |  |
| bcopy() : string string  memcpy() :string sring | |

## bzero(将一段内存内容全清为零)

|  |  |
| --- | --- |
| **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 |
| 范例 |  |
| 参考memset()。 | |

## index(查找字符串中第一个出现的指定字符)

|  |  |
| --- | --- |
| **index(查找字符串中第一个出现的指定字符)** | |
| 相关函数 | rindex,srechr,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 |  |
| 56789012345678901234567890 | |

## memccpy(拷贝内存内容)

|  |  |
| --- | --- |
| **memccpy(拷贝内存内容)** | |
| 相关函数 | bcopy,memcpy,memmove,strcpy,strncpy |
| 表头文件 | #include <string.h> |
| 定义函数 | void \*memccpy(void \*dest, const void \*src, int c,size\_t n); |
| 函数说明 | memccpy()用来拷贝src所指的内存内容前n个字节到dest所指的地址上。与memcpy()不同的是,memccpy()会在复制时检查参数c是否出现,若是则返回dest中值为c的下一个字节地址。 |
| 返回值 | 返回指向dest中值为c的下一个字节指针。返回值为0表示在src所指内存前n个字节中没有值为c的字节。 |
| 范例 |  |
| #include <string.h>  main(){  char a[]=”string[a]”;  char b[]=”string(b)”;  memccpy(a,b,’B’,sizeof(b));  printf(memccpy():%s\n”,a);  } | |
| 执行 |  |
| memccpy():string(b) | |

## memchr(在某一内存范围中查找一特定字符)

|  |  |
| --- | --- |
| **memchr(在某一内存范围中查找一特定字符)** | |
| 相关函数 | index,rindex,strchr,strpbrk,strrchr,strsep,strspn,strstr |
| 表头文件 | #include <string.h> |
| 定义函数 | void \*memchr(const void \*s,int c,size\_t n); |
| 函数说明 | memchr()从头开始搜寻s所指的内存内容前n个字节,直到发现第一个值为c的字节,则返回指向该字节的指针。 |
| 返回值 | 如果找到指定的字节则返回该字节的指针,否则返回0。 |
| 范例 |  |
| #include <string.h>  main(){  char \*s=”0123456789012345678901234567890”;  char \*p;  p=memchr(s,’5’,10);  printf(“%s\n”,p);  } | |
| 执行 |  |
| 56789012345678901234567890 | |

## memcmp(比较内存内容)

|  |  |
| --- | --- |
| **memcmp(比较内存内容)** | |
| 相关函数 | bcmp,strcasecmp,strcmp,strcoll,strncmp,strncasecmp |
| 表头文件 | #include <string.h> |
| 定义函数 | int memcmp(const void \*s1, const void \*s2, size\_t n); |
| 函数说明 | memcmp()用来比较s1和s2所指的内存区间前n个字符。字符串大小的比较是以ASCII码表上的顺序来决定,次顺序亦为字符的值。memcmp()首先将s1第一个字符值减去s2第一个字符的值,若差为0则再继续比较下个字符,若差值不为0则将差值返回。例如,字符串“Ac”和“ba”比较则会返回字符‘A’(65)和‘b’(98)的差值(-33)。 |
| 返回值 | 若参数s1和s2所指的内存内容都完全相同则返回0值。s1若大于s2则返回大于0的值。s1若小于s2则返回小于0的值。 |
| 范例 |  |
| #include <string.h>  main(){  char \*a =”aBcDeF”;  char \*b=”AbCdEf”;  char \*c=”aacdef”;  char \*d=”aBcDeF”;  printf(“memcmp(a,b):%d\n”,memcmp((void\*)a,(void\*) b,6));  printf(“memcmp(a,c):%d\n”,memcmp((void\*)a,(void\*) c,6));  printf(“memcmp(a,d):%d\n”,memcmp((void\*)a,(void\*) d,6));  } | |
| 执行 |  |
| memcmp(a,b):1 /\*字符串a>字符串b,返回1\*/  memcmp(a,c):-1 /\*字符串a<字符串c,返回－1\*/  memcmp(a,d):0 /\*字符串a＝字符串d,返回0\*/ | |

## memcpy(拷贝内存内容)

|  |  |
| --- | --- |
| **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所指的内存地址上。与strcpy()不同的是,memcpy()会完整的复制n个字节,不会因为遇到字符串结束‘\0‘而结束。 |
| 返回值 | 返回指向dest的指针。 |
| 附加说明 | 指针src和dest所指的内存区域不可重叠。 |
| 范例 |  |
| #include <string.h>  main(){  char a[30]=”string(a)”;  char b[30]=”string\0string”;  int i;  strcpy(a,b);  printf(strcpy():”);  for(i=0;i<30;i++)  printf(“%c”,a[i]);  memcpy(a,b,30);  printf(“\nmemcpy() :”);  for(i=0;i<30;i++)  printf(“%c”,a[i]);  } | |
| 执行 |  |
| strcpy() : string a )  memcpy() : string string | |

## memmove(拷贝内存内容)

|  |  |
| --- | --- |
| **memmove(拷贝内存内容)** | |
| 相关函数 | bcopy,memccpy,memcpy,strcpy,strncpy |
| 表头文件 | #include <string.h> |
| 定义函数 | void \*memmove(void \*dest,const void \*src,size\_t n); |
| 函数说明 | memmove()与memcpy()一样都是用来拷贝src所指的内存内容前n个字节到dest所指的地址上。不同的是,当src和dest所指的内存区域重叠时,memmove()仍然可以正确的处理,不过执行效率上会比使用memcpy()略慢些。 |
| 返回值 | 返回指向dest的指针。 |
| 附加说明 | 指针src和dest所指的内存区域可以重叠。 |
| 范例 |  |
| 参考memcpy()。 | |

## memset(将一段内存空间填入某值)

|  |  |
| --- | --- |
| **memset(将一段内存空间填入某值)** | |
| 相关函数 | bzero,swab |
| 表头文件 | #include <string.h> |
| 定义函数 | void \*memset(void \*s ,int c, size\_t n); |
| 函数说明 | memset()会将参数s所指的内存区域前n个字节以参数c填入,然后返回指向s的指针。在编写程序时,若需要将某一数组作初始化,memset()会相当方便。 |
| 返回值 | 返回指向s的指针。 |
| 附加说明 | 参数c虽声明为int, 但必须是unsigned char ,所以范围在0到255之间。 |
| 范例 |  |
| #include <string.h>  main(){  char s[30];  memset(s,’A’,sizeof(s));  s[30]=’\0’;  printf(“%s\n”,s);  } | |
| 执行 |  |
| AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA | |

## rindex(查找字符串中最后一个出现的指定字符)

|  |  |
| --- | --- |
| **rindex(查找字符串中最后一个出现的指定字符)** | |
| 相关函数 | index,memchr,strchr,strrchr |
| 表头文件 | #include <string.h> |
| 定义函数 | char \*rindex( const char \*s,int c); |
| 函数说明 | rindex()用来找出参数s字符串中最后一个出现的参数c 地址,然后将该字符出现的地址返回。字符串结束字符(NULL)也视为字符串一部分。 |
| 返回值 | 如果找到指定的字符则返回该字符所在的地址,否则返回0。 |
| 范例 |  |
| #include <string.h>  mian(){  char \*s =”0123456789012345678901234567890”;  char \*p;  p= rindex(s,’5’);  printf(“%s\n”,p);  } | |
| 执行 |  |
| 567890 | |

## strchr(查找字符串中第一个出现的指定字符)

|  |  |
| --- | --- |
| **strchr(查找字符串中第一个出现的指定字符)** | |
| 相关函数 | index,memchr,rinex,strbrk,strsep,strspn,strstr,strtok |
| 表头文件 | #include <string.h> |
| 定义函数 | char \*strchr(const char \*s,int c); |
| 函数说明 | strchr()用来找出参数s字符串中第一个出现的参数c地址,然后将该字符出现的地址返回。 |
| 返回值 | 如果找到指定的字符则返回该字符所在地址,否则返回0。 |
| 范例 |  |
| #include <string.h>  main(){  char \*s=0123456789012345678901234567890”;  char \*p;  p=strchr(s,’5’);  printf(“%s\n”,p);  } | |
| 执行 |  |
| 56789012345678901234567890 | |

## strcoll(采用目前区域的字符排列次序来比较字符串)

|  |  |
| --- | --- |
| **strcoll(采用目前区域的字符排列次序来比较字符串)** | |
| 相关函数 | strcmp,bcmp,memcmp,strcasecmp,strncasecmp |
| 表头文件 | #include <string.h> |
| 定义函数 | int strcoll( const char \*s1, const char \*s2); |
| 函数说明 | strcoll()会依环境变量LC\_COLLATE所指定的文字排列次序来比较s1和s2字符串。 |
| 返回值 | 若参数s1和s2字符串相同则返回0。s1若大于s2则返回大于0的值。s1若小于s2则返回小于0的值。 |
| 附加说明 | 若LC\_COLLATE为“POSIX”或“C”,则strcoll()与strcmp()作用完全相同。 |
| 范例 |  |
| 参考strcmp()。 | |

## strcspn(返回字符串中连续不含指定字符串内容的字符数)

|  |  |
| --- | --- |
| **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 | #inclued<string.h> |
| 定义函数 | size\_t strcspn( const char \*s,const char \*reject); |
| 函数说明 | strcspn()从参数s字符串的开头计算连续的字符,而这些字符都完全不在参数reject所指的字符串中。简单地说,若strcspn()返回的数值为n,则代表字符串s开头连续有n个字符都不含字符串reject内的字符。 |
| 返回值 | 返回字符串s开头连续不含字符串reject内的字符数目。 |
| 范例 |  |
| #include <string.h>  main(){  char \*str=”Linux was first developed for 386/486-based pcs.”;  printf(“%d\n”,strcspn(str,” “));  printf(“%d\n”,strcspn(str,”/-“));  printf(“%d\n”,strcspn(str,”1234567890”));  } | |
| 执行 |  |
| 1. /\*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)

|  |  |
| --- | --- |
| **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()在内部**先用malloc()分配与参数s字符串相同大小的空间**,然后将参数s字符串的内容复制到该内存地址,然后返回该地址。该地址**最后需要利用free()来释放**,否则会造成内存泄露。 |
| 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。 |
| 附加说明 | strdup()函数的参数不能是NULL,否则报错segmentation fault(core dumped)。与strlen()一样。 |
| 范例 |  |
| #include <string.h>  main(){  char a[]=”strdup”;  char \*b;  b=strdup(a);  printf(“b[ ]=\”%s\”\n”,b);  **free(b); //容易忘记**  } | |
| 执行 |  |
| b[ ]=”strdup”  strdup()定义,@strdup.c  char \*strdup(const char \*str){  size\_t siz;  char \*copy;  siz = strlen(str) + 1;  if((copy = malloc(siz)) == NULL)  return(NULL);  (void)memcpy(copy, str, siz);  return(copy);  } | |

**strdup函数的用法**

http://blog.csdn.net/leichelle/article/details/7465769

函数名: strdup

功能: 将串拷贝到新建的位置处

原型: char \*strdup(const char \*str)；

这个函数在linux的man手册里解释为：

The strdup() function returns a pointer to a new string which is a duplicate of the string s. Memory for the new string is obtained with malloc(3), and can be freed with free(3). The strndup() function is similar, but only copies at most n charac-ters. If s is longer than n, only n characters are copied, and a termi-nating NUL is added.

strdup函数原型：

strdup()主要是拷贝字符串s的一个副本,由函数返回值返回,这个副本有自己的内存空间,和s不相干。strdup函数复制一个字符串,使用完后要记得删除在函数中动态申请的内存,strdup函数的参数不能为NULL,一旦为NULL,就会报段错误,因为该函数包括了strlen函数,而该函数参数不能是NULL。

(报错segmentation fault(core dumped))

实例1：

C/C++ code

#include <stdio.h>

#include <string.h>

#include <alloc.h>

int main(void){

char \*dup\_str,\*string = "abcde";

dup\_str =strdup(string);

printf("%s\n", dup\_str); free(dup\_str); return 0;

free(dup\_str);

}

实例2：

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

unsigned int Test(){

char buf[]="Hello,World!";

char\*pb =strndup(buf,strlen(buf));

return(unsigned int)(pb);

}

int main(){

unsigned int pch= Test();

printf("Testing:%s\n",(char\*)pch);

free((void\*)pch);

return 0;

}

在Test函数里使用strndup而出了Test函数仍可以操作这段内存,并且可以释放。

由这个问题而延伸出来的问题就是,如何让函数得到的内存数据传出函数但仍可用。

解决方法目前本人只想到两个： 一个外部变量,如传递一个内存块指针给函数,但这种做法就是你得传递足够的内存,也就是你不能事先知道这个函数到底要多大的BUFFER。

 另一种方法就是在函数内部申请static变量,当然这也是全局区的变量,但这种做法的缺点就是,当函数多次运行时,static变量里面的数据会被覆盖。这种类型的另一个方法就是使用全局变量,但这和使用static变量很相同,不同的是全局变量可以操作控制,而static变量如果不把它传出函数,就不可对它操作控制了。另一类方法就是上面所述的,利用堆里的内存来实现,但存在危险。strdup是从堆中分配空间的！strdup调用了malloc,所以它需要释放！对于堆栈：堆是由程序员来管理的,比如说new,malloc等等都是在堆上分配的！

栈是由编译器来管理的。

## strlen(返回字符串长度)

|  |  |
| --- | --- |
| **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()用来计算指定的字符串s的长度,不包括结束字符“\0”。 |
| 返回值 | 返回字符串s的字符数。 |
| 附加说明 | strlen()函数的参数不能是NULL,否则报错segmentation fault(core dumped)。 |
| 范例 |  |
| /\*取得字符串str的长度\*/  #include <string.h>  main(){  char \*str =”12345678”;  printf(“str length = %d\n”,strlen(str));  } | |
| 执行 |  |
| str length = 8  strlen所作的仅仅是一个计数器的工作,它从内存的某个位置(可以是字符串开头,中间某个位置,甚至是某个不确定的内存区域)开始扫描,直到碰到第一个字符串结束符'\0'为止(不包括'\0'),然后返回计数器值。 | |

## strcmp(比较字符串)

|  |  |
| --- | --- |
| **strcmp(比较字符串)** | |
| 相关函数 | bcmp,memcmp,strcasecmp,strncasecmp,strcoll |
| 表头文件 | #include <string.h> |
| 定义函数 | int strcmp(const char \*s1,const char \*s2); |
| 函数说明 | strcmp()用来比较参数s1和s2字符串。字符串大小的比较是以ASCII码表上的顺序来决定,此顺序亦为字符的值。strcmp()首先将s1第一个字符值减去s2第一个字符值,若差值为0则再继续比较下个字符,若差值不为0则将差值返回。例如字符串“Ac”和“ba”比较则会返回字符“A”(65)和‘b’(98)的差值(－33)。 |
| 返回值 | 若参数s1和s2字符串相同则返回0，s1若大于s2则返回大于0的值，s1若小于s2则返回小于0的值。 |
| 范例 |  |
| #include <string.h>  main(){  char \*a=”aBcDeF”;  char \*b=”AbCdEf”;  char \*c=”aacdef”;  char \*d=”aBcDeF”;  printf(“strcmp(a,b) : %d\n”,strcmp(a,b));  printf(“strcmp(a,c) : %d\n”,strcmp(a,c));  printf(“strcmp(a,d) : %d\n”,strcmp(a,d));  } | |
| 执行 |  |
| strcmp(a,b) : 32  strcmp(a,c) :-31  strcmp(a,d) : 0 | |

## strcasecmp(忽略大小写比较字符串)

|  |  |
| --- | --- |
| **strcasecmp(忽略大小写比较字符串)** | |
| 相关函数 | bcmp,memcmp,strcmp,strcoll,strncmp |
| 表头文件 | #include <string.h> |
| 定义函数 | int strcasecmp( const char \*s1, const char \*s2); |
| 函数说明 | strcasecmp()用来比较参数s1和s2字符串,比较时会自动忽略大小写的差异。 |
| 返回值 | 若参数s1和s2字符串相同则返回0。s1长度大于s2长度则返回大于0的值,s1长度若小于s2长度则返回小于0的值。 |
| 范例 |  |
| #include <string.h>  main(){  char \*a=”aBcDeF”;  char \*b= “AbCdEf”;  if(!strcasecmp(a,b))  printf(“%s=%s\n”,a,b);  } | |
| 执行 |  |
| aBcDeF=AbCdEf | |

## strncmp(比较字符串)

|  |  |
| --- | --- |
| **strncmp(忽略大小写比较字符串)** | |
| 相关函数 | bcmp,memcmp,strcmp,strcoll,strncasecmp |
| 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()用来比较参数s1和s2字符串前n个字符。 |
| 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(!strncmp(a,b,6))  printf(“%s =%s\n”,a,b);  } | |
| 执行 |  |
| aBcDef=AbCdEf | |

## strncasecmp(忽略大小写比较字符串)

|  |  |
| --- | --- |
| **strncasecmp(忽略大小写比较字符串)** | |
| 相关函数 | bcmp,memcmp,strcmp,strcoll,strncmp |
| 表头文件 | #include <string.h> |
| 定义函数 | int strncasecmp(const char \*s1,const char \*s2,size\_t n); |
| 函数说明 | strncasecmp()用来比较参数s1和s2字符串前n个字符,比较时会自动忽略大小写的差异。 |
| 返回值 | 若参数s1和s2字符串相同则返回0。s1若大于s2则返回大于0的值,s1若小于s2则返回小于0的值。 |
| 范例 |  |
| #include <string.h>  main(){  char \*a=”aBcDeF”;  char \*b=”AbCdEf”;  if(!strncasecmp(a,b,6))  printf(“%s =%s\n”,a,b);  }else{  printf(“%s !=%s\n”,a,b);  }  } | |
| 执行 |  |
| aBcDef !=AbCdEf | |

## strcat(连接两字符串)

|  |  |
| --- | --- |
| **strcat(连接两字符串)** | |
| 相关函数 | bcopy,memccpy,memcpy,strcpy,strncpy |
| 表头文件 | #include <string.h> |
| 定义函数 | char \*strcat(char \*dest,const char \*src); |
| 函数说明 | strcat()会将参数src字符串拷贝到参数dest所指的字符串尾。第一个参数dest要有足够的空间来容纳要拷贝的字符串。 |
| 返回值 | 返回参数dest的字符串起始地址 |
| 范例 |  |
| #include <string.h.>  main(){  char a[30]=”string(1)”;  char b[]=”string(2)”;  printf(“before strcat() : %s\n”,a);  printf(“after strcat() : %s\n”,strcat(a,b));  } | |
| 执行 |  |
| before strcat() : string(1)  after strcat() : string(1)string(2) | |

## strncat(连接两字符串)

|  |  |
| --- | --- |
| **strncat(连接两字符串)** | |
| 相关函数 | bcopy,memccpy,memecpy,strcpy,strncpy |
| 表头文件 | #inclue <string.h> |
| 定义函数 | char \*strncat(char \*dest,const char \*src,size\_t n); |
| 函数说明 | strncat()会将参数src字符串拷贝n个字符到参数dest所指的字符串尾。第一个参数dest要有足够的空间来容纳要拷贝的字符串。 |
| 返回值 | 返回参数dest的字符串起始地址。 |
| 范例 |  |
| #include <string.h>  main(){  char a[30]=”string(1)”;  char b[]=”string(2)”;  printf(“before strnact() :%s\n”,a);  printf(“after strncat() :%s\n”,strncat(a,b,6));  } | |
| 执行 |  |
| before strnact() : string(1)  after strncat() : string(1) string | |

## strcpy(拷贝字符串)

|  |  |
| --- | --- |
| **strcpy(拷贝字符串)** | |
| 相关函数 | bcopy,memcpy,memccpy,memmove |
| 表头文件 | #include <string.h> |
| 定义函数 | char \*strcpy(char \*dest,const char \*src); |
| 函数说明 | strcpy()会将参数src字符串拷贝至参数dest所指的地址。 |
| 返回值 | 返回参数dest的字符串起始地址。 |
| 附加说明 | 如果参数dest所指的内存空间不够大,可能会造成缓冲溢出(buffer Overflow)的错误情况,在编写程序时请特别留意,或者用strncpy()来取代。 |
| 范例 |  |
| #include <string.h>  main(){  char a[30]=”string(1)”;  char b[]=”string(2)”;  printf(“before strcpy() :%s\n”,a);  printf(“after strcpy() :%s\n”,strcpy(a,b));  } | |
| 执行 |  |
| before strcpy() :string(1)  after strcpy() :string(2) | |

## strncpy(拷贝字符串)

|  |  |
| --- | --- |
| **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 |  |
| #inclue <string.h>  main(){  char a[30]=”string(1)”;  char b[]=”string(2)”;  printf(“before strncpy() : %s\n”,a);  printf(after strncpy() : %s\n”,strncpy(a,b,6));  } | |
| 执行 |  |
| before strncpy() : string(1)  after strncpy() : string(1) | |

## strpbrk(查找字符串中第一个出现的指定字符)

|  |  |
| --- | --- |
| **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 <include.h> |
| 定义函数 | char \*strpbrk(const char \*s,const char \*accept); |
| 函数说明 | strpbrk()用来找出参数s字符串中最先出现存在参数accept字符串中的任意字符。 |
| 返回值 | 如果找到指定的字符则返回该字符所在地址,否则返回0。 |
| 范例 |  |
| #include <string.h>  main(){  char \*s=”0123456789012345678901234567890”;  char \*p;  p=strpbrk(s,”a1 839”); /\*1会最先在s字符串中找到\*/  printf(“%s\n”,p);  p=strprk(s,”4398”);/\*3会最先在s字符串中找到\*/  printf(“%s\n”,p);  } | |
| 执行 |  |
| 123456789012345678901234567890  3456789012345678901234567890 | |

## strrchr(查找字符串中最后出现的指定字符)

|  |  |
| --- | --- |
| **strrchr(查找字符串中最后出现的指定字符)** | |
| 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=”0123456789012345678901234567890”;  char \*p;  p=strrchr(s,’5’);  printf(“%s\n”,p);  } | |
| 执行 |  |
| 567890 | |

## strspn(返回字符串中连续不含指定字符串内容的字符数)

|  |  |
| --- | --- |
| **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。简单的说,若strspn()返回的数值为n,则代表字符串s开头连续有n个字符都是属于字符串accept内的字符。 |
| 返回值 | 返回字符串s开头连续包含字符串accept内的字符数目。 |
| 范例 |  |
| #include <string.h>  main(){  char \*str=”Linux was first developed for 386/486-based PCs.”;  char \*t1=”abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ”;  printf(“%d\n”,strspn(str,t1));  } | |
| 执行 |  |
| 5 /\*计算大小写字母。不包含“ ”,所以返回Linux的长度。\*/ | |

## strstr(在一字符串中查找指定的字符串)

|  |  |
| --- | --- |
| **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=”012345678901234567890123456789”;  char \*p;  p= strstr(s,”901”);  printf(“%s\n”,p);  } | |
| 执行 |  |
| 9012345678901234567890 | |

## strtok(分割字符串)

|  |  |
| --- | --- |
| **strtok(分割字符串)** | |
| 相关函数 | index,memchr,rindex,strpbrk,strsep,strspn,strstr |
| 表头文件 | #include <string.h> |
| 定义函数 | char \*strtok(char \*s,const char \*delim); |
| 函数说明 | strtok()用来将字符串**分割成一个个片段**。参数s指向欲分割的字符串,参数delim则为分割字符串,当strtok()在参数s的字符串中发现到参数delim的分割字符时则会**将该字符改为\0字符**。在第一次调用时,strtok()必需给予参数s字符串,往后的调用则将参数s设置成NULL。每次调用成功则返回下一个分割后的字符串指针。 |
| 返回值 | 返回下一个分割后的字符串指针,如果已无从分割则返回NULL。 |
| 范例 |  |
| #include <string.h>  main(){  char s[]=”ab-cd : ef;gh :i-jkl;mnop;qrs-tu: vwx-y;z”;  char \*delim=”-: “;  char \*p;  printf(“%s”,strtok(s,delim));  while((p=strtok(NULL,delim)))printf(“%s ”,p);  printf(“\n”);  } | |
| 执行 |  |
| ab cd ef;gh i jkl;mnop;qrs tu vwx y;z /\*－与：字符已经被\0字符取代\*/ | |

### 范例

#include <string.h>

#include <unistd.h>

main(){

char s[]="ab-cd:ef;gh:i-jkl;mnop;qrs-tu:vwx-y;z";

char \*delim="-:";

char \*p;

printf("%s\n",strtok(s,delim));

printf("pause1\n");

while((p=strtok(NULL,delim))){

printf("%s\n",p);

printf("pause2\n");

}

printf("\n");

}

执行

ab

pause1

cd

pause2

ef;gh

pause2

i

pause2

jkl;mnop;qrs

pause2

tu

pause2

vwx

pause2

y;z

pause2

分析

每一次执行strtok()返回被分割后的字符,第一次返回ab,第二次返回cd,第三次返回ef;gh,…

**s可以用字符指针，或字符数组**。

参考Android5.0源码，external/chromium\_org/third\_party/yasm/source/patched-yasm/tools/re2c/parser.c

**void** **line\_source**(FILE \*o, **unsigned** **int** **line**){

206 **char** \***fnamebuf**;

217 **token** = strtok( **fnamebuf**, "\\" );

218 … }

### 小结:

第一次调用使用strtok(s,delim)，将第一个分隔符**改为\0**，返回第一个分隔符**前面的**字符串指针；

第二次调用使用strtok(NULL,delim)，将第二个分隔符**改为\0**，返回第二个分隔符**前面的**字符串指针；

第三次调用使用strtok(NULL,delim)，将第三个分隔符**改为\0**，返回第三个分隔符**前面的**字符串指针；

第二次及以后的调用使用strtok(NULL,delim)，……

### 注意1:char \*strtok(char \*s,const char \*delim);

由于strtok(s,delim)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**,也就是说,**在调用strtok(s,delim)的过程中,s参数指向的内容是会被修改的,而且是被改为字符串结束符\0。**如果s指向的内容不允许修改,那么,需要**将s中的内容复制到可修改的内存处**,然后再调用strtok(s,delim)函数。

可使用memcpy/strcpy/strdup等拷贝。

如下,使用strdup()复制s参数中的内容到temp指针处。

void some\_function(const char \*s) {//**s指向的内容不允许修改**

char \*temp = strdup(s); //**复制到一个允许修改的内存处**

char \*delim=”-: “;

char \*p;

printf(“%s”,strtok(temp,delim)); //**temp指向的内容允许修改**

while((p=strtok(NULL,delim)))printf(“%s ”,p);

printf(“\n”);

free(temp);

}

### 注意2:字符串常量不可以修改。

main(){

char \*str="I # am # a #c#program";

char \*temp;

int i=0;

printf("before,str=%s,i=%d\n",str,i);

temp = strtok(str,"#");

while(temp) {

i++;

printf("temp=%s\n",temp);

temp = strtok(NULL, "#");

}

printf("after,str=%s,i=%d\n",str,i);

}

**$ ./strtok**

before,str=I # am # a #c#program,i=0

Segmentation fault(core dumped)

**报错。str指向的是文字常量区,不允许修改。**

**解决办法如下：**

将字符串**复制到可修改的内存处**,如下：

main(){

char \*str="I # am # a #c#program";

char \*temp, \*str2;

int i=0;

printf("before,str=%s,i=%d\n",str,i);

**str2 = strdup(str);**

printf("before,str2=%s,i=%d\n",str2,i);

temp = strtok(str2,"#");

while(temp) {

i++;

printf("temp=%s\n",temp);

temp = strtok(NULL, "#");

}

printf("after,str2=%s,i=%d\n",str2,i);

**free(str2);**

}

**$ ./strtok (运行环境，Ubuntu13.0命令行。)**

before,str=I # am # a #c#program,i=0

before,str2=I # am # a #c#program,i=0

temp=I

temp= am

temp= a

temp=c

temp=program

after,str2=I ,i=5

结论1：字符串常量不可以被修改。如果想要修改,可以将其**复制到可修改的内存处**,如memcpy/strcpy/strdup。

结论2：str2已经**被分割**了,上面最后打印的时候只打印了第一个单词I。

char \*strtok(char \*s,const char \*delim),字符串s被分割,s中的**delim被替换为字符串结束符\0,这是分割的含义**,哈哈哈。**strtok()用\0来将字符串分割成一个个片段**。

结论3：**每一次执行strtok()返回被分割后的字符**,第一次返回I,第二次返回am,第三次返回 a,…

### 特别强调

**特别强调，编译器不同会导致不同的结果，这里的代码在Ubuntu13.0/MS VC6.0编译执行正常，但是在cygwin下编译执行却出错。**

注意2中的代码在Cygwin编译执行报错如下：

1>>before,str=I # am # a #c#program,i=0

2>>before,str2=I # am # a #c#program,i=0

Segmentation fault (核心已转储)

终于找到原因了，**是cygwin导致的问题**。**无中生有的问题**，害死人。

**不推荐使用cygwin编译执行代码，有时候导致莫名的出错。**

**要么安装虚拟机用Ubuntu，推荐VMware Player，比VMware Machine占内存小。或者VC++6.0。**

**20160120**

### 注意3:查找字符串中不包含分隔字符,返回什么？

main(){

char s[]=”I love China”;

char \*delim=”#“;

char \*p = strtok(s,delim)); //字符数组s中不包含分隔字符#

while(p) {

printf(“%s \n”,p);

p=strtok(NULL,delim);

}

}

结果：I love China

结论4：**字符串中不包含分隔字符时,返回字符串本身(why?)**。

上述,第一次strtok返回s本身I love China,第二次strtok返回null,while不会执行。

### 注意4:delim是字符指针

char \*strtok(char \*s,const **char \***delim); char \*strchr(const char \*s,**int** c);

和strchr等不同，strtok第2个参数必须是字符指针，必须传入字符串实参，匹配字符串里的每一个字符。即使只想匹配单个字符，也**要写在双引号里**，不能写在单引号里，如char a[]=“I love china”; strtok(a,"e")，写成strtok(a,'e')是错的。而strchr(a, 'e')，写成strchr(a, "e")是错的。

### 1.strtok和strtok\_r的使用要点和实现原理

http://blog.csdn.net/liuintermilan/article/details/6280816

关键:**多次调用**strtok必须是**同一个源字符串。**

**关于函数strtok和strtok\_r的使用要点和实现原理（一）**

strtok函数的使用是一个老生常谈的问题了。该函数的作用很大，争议也很大。以下的表述可能与一些资料有区别或者说与你原来的认识有差异，因此，我尽量以实验为证。交代一下实验环境是必要的，winxp+vc6.0，一个极端平民化的实验环境。本文中使用的源代码大部分来自于网络，我稍加修改作为例证。当然，本人水平有限，有不妥之处在所难免，各位见谅的同时不妨多做实验，以实验为证。

strtok的函数原型为char \*strtok(char \*s, char \*delim)，功能为“Parse S into tokens separated by characters in DELIM.If S is NULL, the saved pointer in SAVE\_PTR is used as the next starting point. ” 翻译成汉语就是：作用于字符串s，以包含在delim中的字符为分界符，将s切分成一个个子串；如果，s为空值NULL，则函数保存的指针SAVE\_PTR在下一次调用中将作为起始位置。

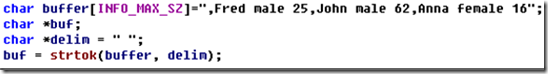
函数的返回值为从指向被分割的子串的指针。

这个定义和国内一些网站上的说法有一些差别，正是这些差别导致很多人对strtok没有一个正确的认识。希望读者在调用一些函数前，最好能够读一读官方的文档（多半都是英文的），而非看一些以讹传讹的资料。

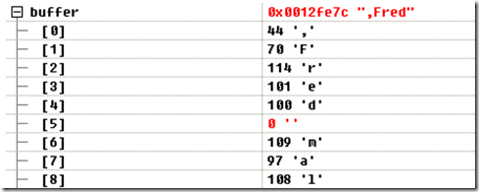
**使用strtok需要注意的有以下几点：**

1.**函数的作用是分解字符串，所谓分解，即没有生成新串，只是在s所指向的内容上做了些手脚而已。因此，源字符串s发生了变化！**

设源字符串s为 char buffer[INFO\_MAX\_SZ]=",Fred male 25,John male 62,Anna female 16";  过滤字符串delim为 char \*delim = " "，即空格为分界符。



上图的代码会产生这样的结果：



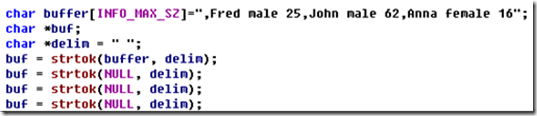
首先，**buffer发生了变化**。如果此时打印buffer的值，会显示“,Fred”，而后面" male 25…16”不翼而飞了。实际上，strtok函数根据delim中的分界符，找到其首次出现的位置，即Fred后面那个空格（buffer[5]），将其修改成了'/0’。其余位置不变。这就很好解释为什么打印buffer的值只能出现“,Fred”，而非buffer中的全部内容了。**因此，使用strtok时一定要慎重，以防止源字符串被修改。**

理解了buffer的变化，就很好解释函数的返回值了。返回值buf为分界符之前的子串（其实这个说法并不确切，详见"3”中对于返回值的详细说明）。**注意，由变量的地址可知，buf依然指向源字符串。**



分界符delim没有发生变化，就不再截图了。

**2.若要在第一次提取子串完毕之后，继续对源字符串s进行提取，应在其后（第二次，第三次。。。第n次）的调用中将strtok的第一个参数赋为空值NULL。**



第一次调用的结果如前文所述，提取出了",Fred”。我们还想继续以空格为分界，提取出后面的"male”等。由上图可以看到，第一次之后的调用我们都给strtok的第一个参数传递了空值NULL（**表示函数继续从上一次调用隐式保存的位置，继续分解字符串；对于上述的第二次调用来说，第一次调用结束前用一个this指针指向了分界符的下一位，即'm’所在的位置**），这样可依次提取出

 ， 。。。。以此类推。。。。。



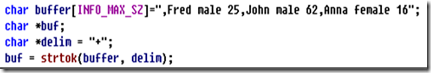
至于为什么要赋空值，要么你就记住结论，要么去查strtok的源代码。本文的最后会有一些介绍。

当然也有部分爱钻牛角尖的人，非不按套路出牌，要看看不赋空值继续赋值为buffer会有什么结果。其实，答案想也能想的到。再一次传递buffer，相当于还从字符串的开头查找分界符delim，而且此时buffer已经被修改（可见的部分只剩下",Fred”），因此，其结果必然是找不到分界符delim。

**3.关于函数返回值的探讨**

由"1”中所述，**在提取到子串的情况下，strtok的返回值（假设返回值赋给了指针buf）是提取出的子串的指针。这个指针指向的是子串在源字符串中的起始位置。子串末尾的下一个字符在提取前为分隔符，提取后被修改成了'/0’。**因此，若打印buf的值，可以成功的输出子串的内容。

**在没有提取到子串的情况下，函数会返回什么值呢？**



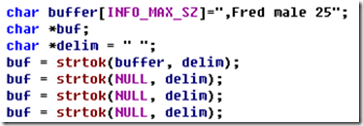
由上图可以看到buffer中并不包含分界符delim。调用strtok后buf的值为



因为没有找到，源字符串buffer没有发生改变，buf指向源字符串的首地址，打印输出的值为整个字符串的完整值。

**什么时候函数的返回值为空值NULL呢？**

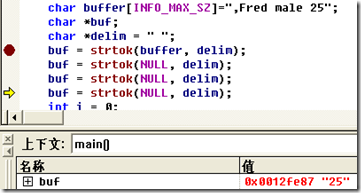
百度百科上说，“**当没有被分割的串时则返回NULL。**”这是一个很模棱两可的说法。如果想要确切的了解清楚这个问题，可能需要看一下strtok的实现原理。这里先以实验说明。



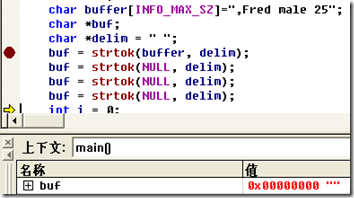
第一次调用strtok，毫无疑问，buf指向",Fred”。

第二次调用strtok，由于第一个参数为NULL，表示函数继续以上次调用所保存的this指针的位置开始分解，即对"male 25”分解。分解完毕后，buf指向"male”。

第三次调用strtok，参数继续设定为NULL，此时即对第二次保存的this指针的位置开始分解，即对"25”分解。因为无法找到包含分隔符delim的子串，所以buf指向"25”。

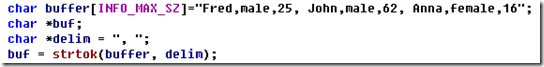


第四次调用，参数仍为NULL，此时第三次调用保存的this指针已指向字符串的末尾'/0’，已无法再进行分解。因此函数返回NULL，这也就是百度百科中所提到的“当没有被分割的串时函数返回NULL。”

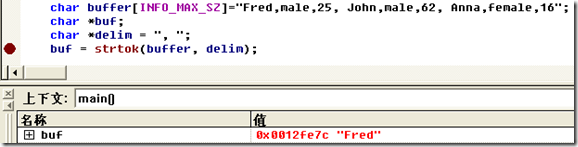


**4.参数 分隔符delim的探讨（delim是分隔符的集合）**

很多人在使用strtok的时候，都想当然的以为函数在分割字符串时完整匹配分隔符delim，比如delim=”ab”，则对于"acdab”这个字符串，函数提取出的是"acd”。至少我在第一次使用的时候也是这么认为的。其实我们都错了，我是在看函数的源代码时才发现这个问题的，且看下面的例子。



源字符串为buffer，分隔符delim为 逗号和空格，按照一般的想法我们会以为调用函数后，buf的值为"Fred,male,25”，结果是这样么？



**第一次调用之后的结果竟然是"Fred”，而非我们所想的结果。**这是为什么呢？

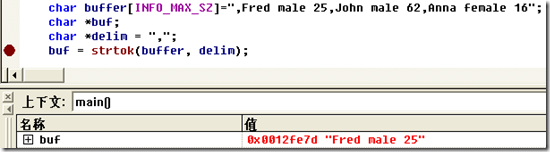
我们回到GNU C Library中对strtok的功能定义：“Parse S into tokens separated by characters in DELIM”。**也就是说包含在delim中的字符均可以作为分隔符，而非严格匹配。可以把delim理解为分隔符的集合。这一点是非常重要的~**

**当然，我们在分解字符串的时候，很少使用多个分隔符。这也导致，很多人在写例子的时候只讨论了一个分隔符的情况。有更多的人在看例子的时候也就错误的认识了delim的作用。**

**5.待分解的字符串，首字符就为分隔符**

首字符为分隔符不能算作一个很特殊的情况。按照常规的分解思路也能正确分解字符串。

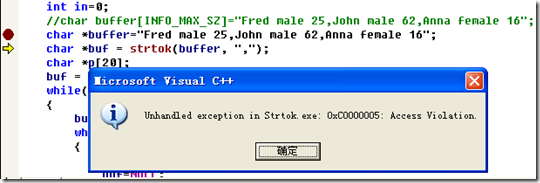
我想说明的是，strtok对于这种情况采用了比常规处理更快的方式。



如上图例子所示。仅用一次调用就可以得到以逗号分隔的字符串"Fred male 25”，而F前面的','被忽略了。由此可见，**strtok在调用的时候忽略了起始位置开始的分隔符。**这一点，可以从strtok的源代码得到证实。

**6.不能向第一个参数传递字符串常量！**

本文中所举的例子都将源字符串保存为字符串数组变量。若你将源字符串定义成字符串常量，可想而知，程序会因为strtok函数试图修改源字符串的值，而抛出异常。



好了，本文详细介绍了使用strtok的注意事项，（二）中我将详细介绍strtok不能实现的一些功能并引出strtok\_r函数，最后介绍一下两个函数的实现。

**关于函数strtok和strtok\_r的使用要点和实现原理（二）**

（一）中已经介绍了使用strtok函数的一些注意事项，本篇将介绍strtok的一个应用并引出strtok\_r函数。

**1.一个应用实例**

网络上一个比较经典的例子是将字符串切分，存入结构体中。如，现有结构体

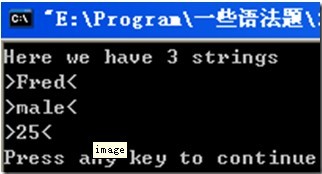
typedef struct person{   
    char name[25];   
    char sex[10];   
    char age[4];   
}Person;

需从字符串 char buffer[INFO\_MAX\_SZ]="Fred male 25,John male 62,Anna female 16"; 中提取出人名、性别以及年龄。

一种可行的思路是设置两层循环。外循环，先以 ',’ （逗号） 为分界符，将三个人的信息分开，然后对于每一个子串，再以 ' ’（空格） 为分界符分别得到人名、性别和年龄。

按照这个思路，**理应**能够实现所要的功能。为了简化步骤，我们调用strtok，先将子串先一一保存到字符串指针数组中，程序末尾打印指针数组中保存的所有子串，验证程序的正确性。得到的程序应该如下：

1. **int** in=0;
2. **char** buffer[INFO\_MAX\_SZ]="Fred male 25,John male 62,Anna female 16";
3. **char** \*p[20];
4. **char** \*buf = buffer;
5. **while**((p[in]=strtok(buf,","))!=NULL) {
6. buf=p[in];
7. **while**((p[in]=strtok(buf," "))!=NULL) {
8. in++;
9. buf=NULL;
10. }
11. buf=NULL;
12. }
13. printf("Here we have %d strings/n", in);
14. **for** (**int** j=0; j<in; j++) {
15. printf(">%s</n",p[j]);
16. }



**执行的结果是，仅仅提取出了第一个人的信息。看来程序的执行并没有按照我们的预想。**原因是什么？

原因是：在**第一次外循环**中，strtok将"Fred male 25,"后的这个逗号，改为了'\0’，**这时strtok内部的this指针指向的是逗号的后一个字符'J’**。**经过第一次的内循环**，分别提取出了“Fred” “male” “25”。提取完"25”之后，**函数内部的this指针被修改指向了"25”后面的'\0’**。**内循环结束后（内循环实际执行了4次），开始第二次的外循环**，由于函数第一个参数被设定为NULL，strtok将以this指针指向的位置作为分解起始位置。**很遗憾，此时this指针指向的是'\0’，strtok对一个空串无法切分，返回NULL。外循环结束。**所以，我们只得到了如图所示的第一个人的信息。

看来使用strtok并不能通过两层循环的办法，解决提取多人信息的问题。**有没有其他办法呢？ 显然，是有其他途径的。**

我给出了一种解决办法。同时以 ',’ （逗号） 和 ' ’（空格） 为分界符，一层循环解决问题。

1. in = 0;
2. **while** ((p[in] = strtok(buf, " ,")) != NULL)  {
3. **switch** (in % 3)  {
4. **case** 0:
5. printf("第%d个人：Name!/n", in/3+1);
6. **break**;
7. **case** 1:
8. printf("第%d个人：Sex!/n", in/3+1);
9. **break**;
10. **case** 2:
11. printf("第%d个人：Age!/n", in/3+1);
12. **break**;
13. }
14. in++;
15. buf = NULL;
16. }
17. printf("Here we have %d strings/n", in);
18. **for** (**int** j=0; j<in; j++)  {
19. printf(">%s</n",p[j]);
20. }



**程序虽然可以达到理想的结果，但不是一个太好解决方案。**程序要求你在提取之前**必须要知道一个结构体中究竟包含了几个数据成员。**明显不如双重循环那样直观。

倘若一定要采用二重循环那种结构提取，有没有合适的函数能够代替strtok呢？ 有的，它就是strtok\_r。

**2.strtok\_r及其使用**

strtok\_r是linux平台下的strtok函数的**线程安全**版。windows的string.h中并不包含它。要想使用这个函数，上网搜其linux下的实现源码，复制到你的程序中即可。别的方式应该也有，比如使用GNU C Library。我下载了GNU C Library，在其源代码中找到了strtok\_r的实现代码，复制过来。可以看作是第一种方法和第二种方法的结合。

strtok的函数原型为**char \*strtok\_r(char \***str**, const char \***delim**, char \*\***saveptr**);**

下面对strtok的英文说明摘自http://www.linuxhowtos.org/manpages/3/strtok\_r.htm，译文是由我给出的。

The **strtok\_r**() function is a reentrant version **strtok**(). The saveptr argument is a pointer to a char \* variable that is used internally by **strtok\_r**() in order to maintain context between successive calls that parse the same string.

strtok\_r函数是strtok函数的**可重入**版本。**char \*\***saveptr参数是一个指向char \*的指针变量，用来在strtok\_r内部保存切分时的上下文，以应对连续调用分解**相同源字符串**。

On the first call to **strtok\_r**(), str should point to the string to be parsed, and the value of saveptr is ignored. In subsequent calls, str should be NULL, and saveptr should be unchanged since the previous call.

第一次调用strtok\_r时，str参数必须指向待提取的字符串，saveptr参数的值可以忽略。连续调用时，str赋值为NULL，saveptr为上次调用后返回的值，不要修改。

Different strings may be parsed concurrently using sequences of calls to **strtok\_r**() that specify different saveptrarguments.

**一系列不同的字符串可能会同时连续调用strtok\_r进行提取，要为不同的调用传递不同的saveptr参数。**

The **strtok**() function uses a static buffer while parsing, so it's not thread safe. Use **strtok\_r**() if this matters to you.

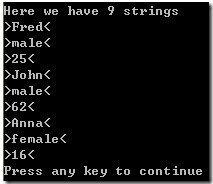
strtok函数在提取字符串时使用了静态缓冲区，因此，它是线程不安全的。如果要顾及到线程的安全性，应该使用strtok\_r。

**strtok\_r实际上就是将strtok内部隐式保存的this指针，以参数的形式与函数外部进行交互。由调用者进行传递、保存甚至是修改。需要调用者在连续切分相同源字符串时，除了将str参数赋值为NULL，还要传递上次切分时保存下的saveptr。**

**注:strtok只能用于同一个源字符串,strtok\_r可用于同一个源字符串、也可以用于不同源字符串，因为源字符串是通过saveptr传入的，可以是不同的源。**

举个例子，还记得前文提到的提取结构体的例子么？我们可以使用strtok\_r，以双重循环的形式提取出每个人的信息。

1. **int** in=0;
2. **char** buffer[INFO\_MAX\_SZ]="Fred male 25,John male 62,Anna female 16";
3. **char** \*p[20];
4. **char** \*buf=buffer;
5. **char** \*outer\_ptr=NULL;
6. **char** \*inner\_ptr=NULL;
7. **while**((p[in] = strtok\_r(buf, ",", &outer\_ptr))!=NULL)  {
8. buf=p[in];
9. **while**((p[in]=strtok\_r(buf, " ", &inner\_ptr))!=NULL)  {
10. in++;
11. buf=NULL;
12. }
13. buf=NULL;
14. }
15. printf("Here we have %d strings/n",in);
16. **for** (**int** j=0; j<in; j++)  {
17. printf(">%s</n",p[j]);
18. }



调用strtok\_r的代码比调用strtok的代码多了两个指针，outer\_ptr和inner\_ptr。outer\_ptr用于标记每个人的提取位置，即外循环；inner\_ptr用于标记每个人内部每项信息的提取位置，即内循环。具体过程如下：

（1）第1次外循环，**outer\_ptr忽略**，对整个源串提取，提取出"Fred male 25"，分隔符',' 被修改为了'\0’，outer\_ptr返回指向'J’。

（2）第一次内循环，**inner\_ptr忽略**，**对第1次外循环的提取结果**"Fred male 25"进行提取，提取出了"Fred"，分隔符' '被修改为了'\0'，inner\_ptr返回指向'm'。

（3）第二次内循环，传递第一次内循环返回的inner\_ptr，第一个参数为NULL，从inner\_ptr指向的位置'm'开始提取，提取出了"male"，分隔符  ' '被修改为了'\0'，inner\_ptr返回指向'2'。

（4）第三次内循环，传递第二次内循环返回的inner\_ptr，第一个参数为NULL，从inner\_ptr指向的位置'2'开始提取，提取出了"25"，因为没有找到' '，inner\_ptr返回指向25后的'\0'。

（5）第四次内循环，传递第三次内循环返回的inner\_ptr，第一个参数为NULL，因为inner\_ptr指向的位置为'\0'，无法提取，返回空值。结束内循环。

（6）第2次外循环，**传递第1次外循环返回的outer\_ptr**，第一个参数为NULL，从outer\_ptr指向的位置'J'开始提取，提取出"John male 62"，分隔符',’被修改为了'\0’，outer\_ptr返回指向'A’。（**调用strtok则卡死在了这一步**）

……以此类推，外循环一次提取一个人的全部信息，内循环从外循环的提取结果中，二次提取个人单项信息。

**可以看到strtok\_r将原内部指针显示化，提供了saveptr这个参数。增加了函数的灵活性和安全性。**

**3.strtok和strtok\_r的源代码**

这两个函数的实现，由众多的版本。我strtok\_r来自于GNU C Library，strtok则调用了strtok\_r。因此先给出strtok\_r的源代码。

1. /\* Parse S into tokens separated by characters in DELIM.
2. If S is NULL, the saved pointer in SAVE\_PTR is used as
3. the next starting point.  For example:
4. char s[] = "-abc-=-def";
5. char \*sp;
6. x = strtok\_r(s, "-", &sp);      // x = "abc", sp = "=-def"
7. x = strtok\_r(NULL, "-=", &sp);  // x = "def", sp = NULL
8. x = strtok\_r(NULL, "=", &sp);   // x = NULL
9. // s = "abc\0-def\0"
10. \*/
11. **char** \*strtok\_r(**char** \*s, **const** **char** \*delim, **char** \*\*save\_ptr) {
12. **char** \*token;
14. **if (s == NULL) s = \*save\_ptr;** //**以s作为源字符串,在s传入NULL时会以\*save\_ptr作为源字符串**
15. //可知,第1次调用时s也可以设置为NULL,此时会以\*sava\_ptr作为源
16. /\* Scan leading delimiters.  \*/
17. s += strspn(s, delim);
18. **if** (\*s == '\0')
19. **return** NULL;
21. /\* Find the end of the token.  \*/
22. token = s;
23. s = strpbrk(token, delim);
24. **if** (s == NULL)
25. /\* This token finishes the string.  \*/
26. \*save\_ptr = strchr(token, '\0');
27. **else** {
28. /\* Terminate the token and make \*SAVE\_PTR point past it.  \*/
29. \*s = '\0';
30. \*save\_ptr = s + 1;
31. }
33. **return** token;
34. }

代码整体的流程如下：

（1）判断参数s是否为NULL，如果是NULL就以传递进来的save\_ptr作为起始分解位置；若不是NULL，则以s开始切分。

（2）跳过待分解字符串开始的所有分界符。

（3）判断当前待分解的位置是否为'\0'，若是则返回NULL（联系到（一）中所说对返回值为NULL的解释）；不是则继续。

（4）保存当前的待分解串的指针token，调用strpbrk在token中找分界符：如果找不到，则将save\_ptr赋值为待分解串尾部'\0'所在的位置，token没有发生变化；若找的到则将分界符所在位置赋值为'\0'，token相当于被截断了（提取出来），save\_ptr指向分界符的下一位。

（5）函数的最后（无论找到还是没找到）都将返回。

**对于函数strtok来说，可以理解为用一个内部的静态变量将strtok\_r中的save\_ptr给保存起来，对调用者不可见。**其代码如下：

1. **char** \*strtok(**char** \*s, **const** **char** \*delim)  { //**第1次调用,s传入待分割字符串(源)**
2. **static** **char** \*last;
3. **return** strtok\_r(s, delim, &last);  //**可知,第2次及以后,s必须指定为NULL**
4. }

 有了上述两个函数的实现代码，再理解（一）（二）中所讲的一些要点也就不困难了。

花那么多篇幅总结这两个函数，一来是因为很对人对于strtok的误解比较深，网上很少有对于其非常详细的讨论，因此总结一份比较全面的材料，是有必要的；二来这也是自己不断学习的一个过程，总结会得到远比两个函数重要很多的信息。

### 1.代码(参考)

参考代码: WFA9.0/lib/wfa\_cmdproc.c:: xcCmdProcStaSetIpConfig()

功能:**字符串解析**，输入字符串pcmdStr为:

sta\_set\_ip\_config,interface,interfaceId\_1,dhcp,0,ip,192.168.1.101,mask,255.255.255.0,primary-dns,192.168.1.1,secondary-dns,192.168.1.101。

参考：Sigma\_Control\_API\_Specification\_v8.0.0.pdf

int xcCmdProcStaSetIpConfig(char \*pcmdStr, BYTE \*aBuf, int \*aLen){

dutCommand\_t staSetIpConfig;

caStaSetIpConfig\_t \*setip = (caStaSetIpConfig\_t \*)&staSetIpConfig.cmdsu.ipconfig;

caStaSetIpConfig\_t defparams = {"", 0, "", "", "", "", ""};

char \*str;

if(aBuf == NULL)

return WFA\_FAILURE;

memset(aBuf, 0, \*aLen);

memcpy(setip, &defparams, sizeof(caStaSetIpConfig\_t));

**for(;;){**

str = strtok\_r(NULL, ",", &pcmdStr);

if(str == NULL || str[0] == '\0')

break;

if(strcasecmp(str, "interface") == 0) {

str = strtok\_r(NULL, ",", &pcmdStr);

strncpy(setip->intf, str, 15);

DPRINT\_INFO(WFA\_OUT, "interface %s\n", setip->intf);

}

else if(strcasecmp(str, "dhcp") == 0) {

str = strtok\_r(NULL, ",", &pcmdStr);

setip->isDhcp = atoi(str);

DPRINT\_INFO(WFA\_OUT, "dhcp %i\n", setip->isDhcp);

}

else if(strcasecmp(str, "ip") == 0) {

str = strtok\_r(NULL, ",", &pcmdStr);

strncpy(setip->ipaddr, str, 15);

DPRINT\_INFO(WFA\_OUT, "ip %s\n", setip->ipaddr);

}

else if(strcasecmp(str, "mask") == 0) {

str = strtok\_r(NULL, ",", &pcmdStr);

strncpy(setip->mask, str, 15);

DPRINT\_INFO(WFA\_OUT, "mask %s\n", setip->mask);

}

else if(strcasecmp(str, "defaultGateway") == 0) {

str = strtok\_r(NULL, ",", &pcmdStr);

strncpy(setip->defGateway, str, 15);

DPRINT\_INFO(WFA\_OUT, "gw %s\n", setip->defGateway);

}

else if(strcasecmp(str, "primary-dns") == 0) {

str = strtok\_r(NULL, ",", &pcmdStr);

strncpy(setip->pri\_dns, str, 15);

DPRINT\_INFO(WFA\_OUT, "dns p %s\n", setip->pri\_dns);

}

else if(strcasecmp(str, "secondary-dns") == 0) {

str = strtok\_r(NULL, ",", &pcmdStr);

strncpy(setip->sec\_dns, str, 15);

DPRINT\_INFO(WFA\_OUT, "dns s %s\n", setip->sec\_dns);

}

else {

DPRINT\_ERR(WFA\_ERR, "invalid command %s\n",str);

return WFA\_FAILURE;

}

}

wfaEncodeTLV(WFA\_STA\_SET\_IP\_CONFIG\_TLV, sizeof(staSetIpConfig), (BYTE \*)&staSetIpConfig, aBuf);

\*aLen = 4+sizeof(staSetIpConfig);

return WFA\_SUCCESS;

}

点评:将strtok\_r()放在for循环中，在for里面进行if判断，很好。

我之前写的代码是在for里面逐个的key/val取值并赋值给传入的指针参数，**在for循环之外if判断**，**浪费for的逐次循环**。

**应该利用for的逐次循环、在for里面逐个的比较判断。SGK20160408**

### C语言很多的陷阱，稍不注意就出错。

C语言很多的陷阱，稍不注意就出错。20160119

# 文件操作

## creat(创建文件)

|  |  |
| --- | --- |
| **creat(创建文件)** | |
| 相关函数 | read,write,fcntl,close,link,stat,umask,unlink,fopen |
| 表头文件 | #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h> |
| 定义函数 | int creat(const char \*pathname, mode\_t mode); |
| 函数说明 | 参数pathname指向欲建立的文件路径。creat()相当于使用下列的调用方式调用open()：  open(const char \*pathname ,(O\_CREAT|O\_WRONLY|O\_TRUNC)); |
| 错误代码 | 关于参数mode请参考open()函数。 |
| 返回值 | creat() 会返回新的文件描述符,若有错误发生则会返回-1,并把错误代码设给errno。  EEXIST 参数pathname所指的文件已存在。  EACCESS 参数pathname所指定的文件不符合所要求测试的权限  EROFS 欲打开写入权限的文件存在于只读文件系统内  EFAULT 参数pathname指针超出可存取的内存空间  EINVAL 参数mode不正确。  ENAMETOOLONG 参数pathname太长。  ENOTDIR 参数pathname为一目录  ENOMEM 核心内存不足  ELOOP 参数pathname有过多符号连接问题。  EMFILE 已达到进程可同时打开的文件数上限  ENFILE 已达到系统可同时打开的文件数上限 |
| 附加说明 | creat() 无法建立特别的装置文件,如果需要请使用mknod()。 |
| 范例 |  |
| 请参考open()。 | |

## open(打开文件)

|  |  |
| --- | --- |
| **open(打开文件)** | |
| 相关函数 | read,write,fcntl,close,link,stat,umask,unlink,fopen |
| 表头文件 | #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h> |
| 定义函数 | int open( const char \*pathname, int flags);  int open( const char \*pathname,int flags, mode\_t mode); |
| 函数说明 | 参数pathname指向欲打开的文件路径字符串。  下列是参数flags所能使用的旗标：  O\_RDONLY 以只读方式打开文件  O\_WRONLY 以只写方式打开文件  O\_RDWR 以可读写方式打开文件。上述三种旗标是互斥的,也就是不可同时使用,但可与下列的旗标利用OR(|)运算符组合。  O\_CREAT 若欲打开的文件不存在则自动建立该文件。  O\_EXCL 如果O\_CREAT也被设置,此指令会去检查文件是否存在。文件若不存在则建立该文件,否则将导致打开文件错误。此外,若O\_CREAT与O\_EXCL同时设置,并且欲打开的文件为符号连接,则会打开文件失败。  O\_NOCTTY 如果欲打开的文件为终端机设备时,则不会将该终端机当成进程控制终端机。  O\_TRUNC 若文件存在并且以可写的方式打开时,此旗标会令文件长度清为0,而原来存于该文件的资料也会消失。  O\_APPEND 当读写文件时会从文件尾开始移动,也就是所写入的数据会以附加的方式加入到文件后面。  O\_NONBLOCK 以不可阻断的方式打开文件,也就是无论有无数据读取或等待,都会立即返回进程之中。  O\_NDELAY 同O\_NONBLOCK。  O\_SYNC 以同步的方式打开文件。  O\_NOFOLLOW 如果参数pathname所指的文件为一符号连接,则会令打开文件失败。  O\_DIRECTORY 如果参数pathname所指的文件并非为一目录,则会令打开文件失败。此为Linux2.2以后特有的旗标,以避免一些系统安全问题。  参数mode则有下列数种组合,只有在建立新文件时才会生效,此外真正建文件时的权限会受到umask值所影响,因此该文件权限应该为(mode-umaks).  S\_IRWXU,00700权限,代表该文件所有者具有可读、可写及可执行的权限。  S\_IRUSR 或S\_IREAD,00400权限,代表该文件所有者具有可读取的权限。  S\_IWUSR 或S\_IWRITE,00200权限,代表该文件所有者具有可写入的权限。  S\_IXUSR 或S\_IEXEC,00100权限,代表该文件所有者具有可执行的权限。  S\_IRWXG 00070权限,代表该文件用户组具有可读、可写及可执行的权限。  S\_IRGRP 00040权限,代表该文件用户组具有可读的权限。  S\_IWGRP 00020权限,代表该文件用户组具有可写入的权限。  S\_IXGRP 00010权限,代表该文件用户组具有可执行的权限。  S\_IRWXO 00007权限,代表其他用户具有可读、可写及可执行的权限。  S\_IROTH 00004权限,代表其他用户具有可读的权限  S\_IWOTH 00002权限,代表其他用户具有可写入的权限。  S\_IXOTH 00001权限,代表其他用户具有可执行的权限。 |
| 返回值 | 若所有欲核查的权限都通过了检查则返回0值,表示成功,只要有一个权限被禁止则返回-1。 |
| 错误代码 | EEXIST 参数pathname所指的文件已存在,却使用了O\_CREAT和O\_EXCL旗标  EACCESS 参数pathname所指的文件不符合所要求测试的权限。  EROFS 欲测试写入权限的文件存在于只读文件系统内。  EFAULT 参数pathname指针超出可存取内存空间。  EINVAL 参数mode不正确。  ENAMETOOLONG 参数pathname太长。  ENOTDIR 参数pathname不是目录。  ENOMEM 核心内存不足。  ELOOP 参数pathname有过多符号连接问题。  EIO I/O存取错误。 |
| 附加说明 | 使用access()作用户认证方面的判断要特别小心,例如在access()后再作open()空文件可能会造成系统安全上的问题。 |
| 范例 |  |
| #include <unistd.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  main(){  int fd,size;  char s [ ]=”Linux Programmer!\n”,buffer[80];  fd=open(“/tmp/temp”,O\_WRONLY|O\_CREAT);  write(fd,s,sizeof(s));  close(fd);  fd=open(“/tmp/temp”,O\_RDONLY);  size=read(fd,buffer,sizeof(buffer));  close(fd);  printf(“%s”,buffer);  } | |
| 执行 |  |
| Linux Programmer! | |

## close(关闭文件)

|  |  |
| --- | --- |
| **close(关闭文件)** | |
| 相关函数 | open,fcntl,shutdown,unlink,fclose |
| 表头文件 | #include <unistd.h> |
| 定义函数 | int close(int fd); |
| 函数说明 | 当使用完文件后若已不再需要则可使用close()关闭该文件,二close()会让数据写回磁盘,并释放该文件所占用的资源。参数fd为先前由open()或creat()所返回的文件描述符。 |
| 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非有效的文件描述符或该文件已关闭。 |
| 附加说明 | 虽然在进程结束时,系统会自动关闭已打开的文件,但仍建议自行关闭文件,并确实检查返回值。 |
| 范例 |  |
| 参考open() | |

## read(由已打开的文件读取数据)

|  |  |
| --- | --- |
| **read(由已打开的文件读取数据)** | |
| 相关函数 | readdir,write,fcntl,close,lseek,readlink,fread |
| 表头文件 | #include <unistd.h> |
| 定义函数 | ssize\_t read(int fd,void \*buf ,size\_t count); |
| 函数说明 | read()会把参数fd所指的文件传送count个字节到buf指针所指的内存中。若参数count为0,则read()不会有作用并返回0。返回值为实际读取到的字节数,如果返回0,表示已到达文件尾或是无可读取的数据,此外文件读写位置会随读取到的字节移动。 |
| 附加说明 | 如果顺利read()会返回实际读到的字节数,最好能将返回值与参数count作比较,若返回的字节数比要求读取的字节数少,则有可能读到了文件尾、从管道(pipe)或终端机读取,或者是read()被信号中断了读取动作。当有错误发生时则返回-1,错误代码存入errno中,而文件读写位置则无法预期。 |
| 错误代码 | EINTR 此调用被信号所中断。  EAGAIN 当使用不可阻断I/O时(O\_NONBLOCK),若无数据可读取则返回此值。  EBADF 参数fd非有效的文件描述符,或该文件已关闭。 |
| 范例 |  |
| 参考open()。 | |

## write(将数据写入已打开的文件内)

|  |  |
| --- | --- |
| **write(将数据写入已打开的文件内)** | |
| 相关函数 | open,read,fcntl,close,lseek,sync,fsync,fwrite |
| 表头文件 | #include <unistd.h> |
| 定义函数 | ssize\_t write(int fd,const void \*buf,size\_t count); |
| 函数说明 | write()会把参数buf所指的内存写入count个字节到参数fd所指的文件内。当然,文件读写位置也会随之移动。 |
| 返回值 | 如果顺利write()会返回实际写入的字节数。当有错误发生时则返回-1,错误代码存入errno中。 |
| 错误代码 | EINTR 此调用被信号所中断。  EAGAIN 当使用不可阻断I/O时(O\_NONBLOCK),若无数据可读取则返回此值。  EADF 参数fd非有效的文件描述符,或该文件已关闭。 |
| 范例 |  |
| 请参考open()。 | |

## dup(复制文件描述符)

|  |  |
| --- | --- |
| **dup(复制文件描述符)** | |
| 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所指的文件描述符,并返回新的文件描述符。此新的文件描述符和参数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()对某个文件描述符作用时,另一个文件描述符的读写位置也会随着改变。  不过,文件描述符之间并不共享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**文件描述符。若有错误则返回-1,errno会存放错误代码。 |
| 错误代码 | EBADF 参数fd非有效的文件描述符,或该文件已关闭。 |

## dup2(复制文件描述符)

|  |  |
| --- | --- |
| **dup2(复制文件描述符)** | |
| 相关函数 | open,close,fcntl,dup |
| 表头文件 | #include <unistd.h> |
| 定义函数 | int dup2(int odlfd,int newfd); |
| 函数说明 | dup2()用来复制参数oldfd所指的文件描述符,并将它拷贝至参数newfd。若参数newfd为一已打开的文件描述符,则newfd所指的文件会先被关闭。dup2()所复制的文件描述符,与原来的文件描述符共享各种文件状态,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符。若有错误则返回-1,errno会存放错误代码。 |
| 附加说明 | dup2()相当于调用fcntl(oldfd,F\_DUPFD,newfd)；请参考fcntl()。 |
| 错误代码 | EBADF 参数fd非有效的文件描述符,或该文件已关闭 |

### 1.dup/dup2解析

**int dup(int filedes) / int dup2(int filedes, int filedes2)**

**关键是区分:文件,文件描述符,文件描述符的指向(filedes:file description)**。

文件描述符代表一个打开的文件，指向文件。文件描述符操作的对象是其代表的文件。

Linux一切皆文件,文件打开后产生一个int型文件描述符。值为0/1/2的文件描述符默认为shell打开,即标准输入/输出/错误输出。STDIN\_FILENO=0标准输入、STDOUT\_FILENO=1标准输出、STDERR\_FILENO=2标准错误输出。

ret=int dup(int filedes)的**作用**:**产生一个新的文件描述符ret,将现有文件描述符filedes复制到新的文件描述符ret**。这样,ret和filedes**都指向同一个文件**,即**原filedes所指向的文件**。(多个文件描述符指向同一个文件，相当于同一个文件有不同的名字)。

dup/dup2后对新的ret/filedes2的操作和对旧的filedes的操作,**效果是一样的**,**都是在操作它们所指向的同一个文件**。

dup/dup2区别:dup产生新的文件描述符的值由系统分配，而dup2可以由调用者指定新的文件描述符的值:filedes2。

*参考: 1.《Linux程序设计(第4版)Neil Matthew,陈健译》13.5.2把管道用作标准输入和标准输出*

*2.《UNIX环境高级编程(第三版)[美]理查德·史蒂文斯》3.12 dup和dup2函数*

**例**

int fd, fd2;    
mode\_t fd\_mode = S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IROTH;    
void redir\_stdout(const char \*filename)  {

fd2=dup(STDOUT\_FILENO);  //dup后，fd2表示标准输出

fd = open(filename, O\_WRONLY|O\_CREAT, fd\_mode);

dup2(fd, STDOUT\_FILENO);

close(fd);

}

将文件描述符fd复制到了stdout，从而，之后对stdout的操作等效于对fd的操作，也就是对文件filename的操作，因此，以后输出到stdout的数据都会输出到fd，看上去就是将stdout重定向到了文件filename。

**从文件描述符的角度看是复制，从输入输出的角度看是重定向。**

如果我们想把刚刚定向到fd的输出,再定向回标准输出,我们可以用下面的代码实现：

void resume\_stdout() {//恢复输出,把标准输出定向到fd2,fd2代表的是标准输出

dup2(fd2, STDOUT\_FILENO);

close(fd2);    
}

**dup()函数**

利用函数dup,我们可以复制一个描述符。传给该函数一个既有的描述符,它就会返回一个新的描述符,这个新的描述符是传给它的描述符的拷贝。这意味着,这两个描述符**共享同一个数据结构**。例如,如果我们对一个文件描述符执行lseek操作,得到的第一个文件的位置和第二个是一样的。下面是用来说明dup函数使用方法的代码片段：

**int** fd1, fd2;

...

fd2 = dup( fd1 );

    需要注意的是,我们可以在调用fork之前建立一个描述符,这与调用dup建立描述符的效果是一样的,子进程也同样会收到一个复制出来的描述符。

**dup2()函数**

    dup2函数跟dup函数相似,但dup2函数允许调用者指定一个有效描述符值作为目标描述符。dup2函数成功返回时,目标描述符（dup2函数的第二个参数）将变成源描述符（dup2函数的第一个参数）的复制品,换句话说,两个文件描述符现在都**指向同一个文件**,并且是函数第一个参数指向的文件。下面我们用一段代码加以说明：

**int** oldfd;

oldfd = open("app\_log",(O\_RDWR | O\_CREATE), 0644 );

dup2( oldfd, 1 );

close( oldfd );

    在本例中,我们打开了一个新文件,称为“app\_log”,并收到一个文件描述符,该描述符叫做fd1。我们调用dup2函数,参数为oldfd和1,这会导致用我们新打开的文件描述符替换掉由1代表的文件描述符（即stdout,因为标准输出文件的id为1）。任何写到stdout的东西,现在都将改为写入名为“app\_log”的文件中。需要注意的是,dup2函数在复制了oldfd之后,会立即将其关闭,但不会关掉新近打开的文件描述符,因为文件描述符1现在也指向它。

**例子**

下面我们介绍一个更加深入的示例代码。回忆一下命令行管道,我们可以将ls –l命令的标准输出作为标准输入连接到wc –l命令。接下来,我们就用一个C程序来加以说明这个过程的实现。代码如下所示。

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main() {

int pfds[2];

if(pipe(pfds) == 0 ) {

if(fork() == 0) {

close(1);

dup2( pfds[1], 1 );

close( pfds[0] );

execlp( "ls", "ls", "-l", NULL );

} else {

close(0);

dup2( pfds[0], 0 );

close( pfds[1] );

execlp( "wc", "wc", "-l", NULL );

}

}

return 0;

}

    在示例代码中,首先在第9行代码中建立一个管道,然后将应用程序分成两个进程：一个子进程（第10–14行）和一个父进程（第16–19行）。接下来,在子进程中首先关闭stdout描述符（第13行）,然后提供了ls –l命令功能,不过它不是写到stdout（第13行）,而是写到我们建立的管道的输出端,这是通过dup2函数来完成重定向的。在第14行,使用dup2 函数把stdout重定向到管道（pfds[1]）。之后,马上关掉管道的输入端。然后,使用execlp函数把子进程的映像替换为命令ls –l的进程映像,一旦该命令执行,它的任何输出都将发给管道的输入端。

    现在来研究一下管道的接收端。从代码中可以看出,管道的接收端是由父进程来担当的。首先关闭stdin描述符（第16行）,因为我们不会从机器的键盘等标准设备文件来接收数据的输入,而是从其它程序的输出中接收数据。然后,再一次用到dup2函数（第17行）,让管道的输入端作为输入,这是通过让文件描述符0（即常规的stdin）重定向到pfds[0]实现的。关闭管道的stdout端（pfds[1]）,因为在这里用不到它。最后,使用 execlp函数把父进程的映像替换为命令wc -l的进程映像,命令wc -l把管道的内容作为它的输入（第23行）。

## fcntl(文件描述符操作)

|  |  |
| --- | --- |
| **fcntl(文件描述符操作)** | |
| 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()用来操作文件描述符的一些特性。参数fd代表欲设置的文件描述符,参数cmd代表欲操作的指令,有以下几种情况：  F\_DUPFD 用来查找大于或等于参数arg的最小且仍未使用的文件描述符,并且复制参数fd  的文件描述符。执行成功则返回新复制的文件描述符。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 取得文件描述符状态旗标,此旗标为open()的参数flags。  F\_SETFL 设置文件描述符状态旗标,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flcok结构的l\_type值必须是F\_RDLCK、F\_WRLCK或F\_UNLCK。如果无法建立锁定,则返回-1,错误代码为EACCES或EAGAIN。  F\_SETLKW F\_SETLK作用相同,但是无法建立锁定时,此调用会一直等到锁定动作成功为止。若在等待锁定的过程中被信号中断时,会立即返回-1,错误代码为EINTR。参数lock指针为flock结构指针,定义如下：  struct flock {  short int l\_type; /\*锁定的状态\*/  short int l\_whence;/\*决定l\_start位置\*/  off\_t l\_start; /\*锁定区域的开头位置\*/  off\_t l\_len; /\*锁定区域的大小\*/  pid\_t l\_pid; /\*锁定动作的进程\*/  };  l\_type 有三种状态：  F\_RDLCK 建立一个供读取用的锁定  F\_WRLCK 建立一个供写入用的锁定  F\_UNLCK 删除之前建立的锁定  l\_whence也有三种方式：  SEEK\_SET 以文件开头为锁定的起始位置。  SEEK\_CUR 以目前文件读写位置为锁定的起始位置  SEEK\_END 以文件结尾为锁定的起始位置。 |
| 返回值 | 成功则返回0,若有错误则返回-1,错误原因存于errno. |

### ioctl和fcntl函数区别及作用

http://blog.itpub.net/14973561/viewspace-750222/

ioctl - control device

ioctl() performs a variety of control functions on devices and STREAMS. For non-STREAMS files, the functions performed by this call are device-specific control functions. Request and an optional third argument with varying type are passed to the file designated by fildes and are interpreted by the device driver.

The fcntl() function provides control of open file descriptors. It is similar to ioctl().

so：

正如名字所暗示的

1. ioctl用于设备控制，控制命令与参数都与设备高度相关，通常也与系统高度相关。

不过IEEE Std 1003.1对流设备的ioctl进行了标准化。

引用

The ioctl() function performs a variety of control functions on STREAMS devices. For non-STREAMS devices, the functions performed by this call are unspecified. The request argument and an optional third argument (with varying type) are passed to and interpreted by the appropriate part of the STREAM associated with fildes.

2. fcntl用于文件控制。

标准化的操作有

F\_DUPFD

F\_GETFD

F\_SETFD

F\_GETFL

F\_SETFL

F\_GETLK

F\_SETLK

F\_SETLKW

F\_GETOWN

F\_SETOWN

### fcntl之F\_SETFL,F\_GETFL

fcntl即F\_SETFL,F\_GETFL的使用，设置文件的flags,阻塞设置成非阻塞，非阻塞设置成阻塞(这连个在server开发中可以封装为基本函数)。

http://blog.163.com/xychenbaihu@yeah/blog/static/132229655201010265577965/

**1、获取文件的flags，即open函数的第二个参数:**

**flags = fcntl(fd,F\_GETFL,0);**

**2、设置文件的flags:**

**fcntl(fd,F\_SETFL,flags);**

**3、增加文件的某个flags，比如文件是阻塞的，想设置成非阻塞:**

**flags = fcntl(fd,F\_GETFL,0);**

**flags |= O\_NONBLOCK;**

**fcntl(fd,F\_SETFL,flags);**

**4、取消文件的某个flags，比如文件是非阻塞的，想设置成为阻塞:**

**flags = fcntl(fd,F\_GETFL,0);**

**flags &= ~O\_NONBLOCK;**

**fcntl(fd,F\_SETFL,flags);**

**获取和设置文件flags举例:**

**#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <fcntl.h>  
#include <error.h>**

**char buf[500000];**

**int main(int argc,char \*argv[]){  
        int ntowrite,nwrite;  
        const char \*ptr ;  
        int flags;  
      
        ntowrite = read(STDIN\_FILENO,buf,sizeof(buf));  
        if(ntowrite <0) {     
                perror("read STDIN\_FILENO fail:");  
                exit(1);  
        }     
        fprintf(stderr,"read %d bytes\n",ntowrite);  
      
        if((flags = fcntl(STDOUT\_FILENO,F\_GETFL,0))==-1) {     
                perror("fcntl F\_GETFL fail:");  
                exit(1);  
        }     
        flags |= O\_NONBLOCK;  
        if(fcntl(STDOUT\_FILENO,F\_SETFL,flags)==-1) {     
                perror("fcntl F\_SETFL fail:");  
                exit(1);  
        }     
      
        ptr = buf;  
        while(ntowrite > 0) {     
                nwrite = write(STDOUT\_FILENO,ptr,ntowrite);  
                if(nwrite == -1) {**

**perror("write file fail:");  
                }     
                if(nwrite > 0)  {     
                        ptr += nwrite;  
                        ntowrite -= nwrite;  
                }     
        }     
      
        flags &= ~O\_NONBLOCK;  
        if(fcntl(STDOUT\_FILENO,F\_SETFL,flags)==-1) {     
                perror("fcntl F\_SETFL fail2:");  
        }     
        return 0;  
}**

### fcntl之F\_SETFD,F\_GETFD

http://blog.chinaunix.net/uid-233938-id-162601.html

上文书说到对于这两个参数还没有搞定是如何用的，结果斗胆在CU发了一个帖子，终于弄明白了~~

[root@FC5 tmp]# cat fcntl.c   
#include <stdio.h>  
#include <fcntl.h>  
#include <unistd.h>  
int set\_fd(int);  
int main(){  
        int fd,ret;  
        char f[20];  
        fd=open("/tmp/pighead.txt",O\_RDWR|O\_CREAT);  
        set\_fd(fd);  
        sprintf(f,"%d",fd);  
        ret=execl("/tmp/test",f,(char \*)0);  
        printf("ret:%d\n",ret);  
        return 0;  
}

int set\_fd(int fd){  
        unsigned int flag;  
        flag=fcntl(fd,F\_GETFD);  
        flag|=~FD\_CLOEXEC; //去掉'~'看看  
        fcntl(fd,F\_SETFD,flag);  
        return 0;  
}  
[root@FC5 tmp]# cat test.c  
#include <stdio.h>  
#include <fcntl.h>  
#include <string.h>  
int main(int argc,char \*\*argv){  
 int fd,ret;  
 system("echo abc >te.log");  
 printf("%s:argv[1]:%s\n",argv[0],argv[1]);  
 fd=atoi(argv[0]);  
 char str[16];  
 strcpy(str,"hello,world~\n");

printf("%s\n",str);

ret=write(fd,str,strlen(str));

printf("ret:%d\n",ret);

return 0;  
}

**结果**

[root@FC5 tmp]# gcc -o test test.c

[root@FC5 tmp]# gcc -o fcnt fcntl.c

[root@FC5 tmp]# ls pighead.txt

ls: pighead.txt: No such file or directory

[root@FC5 tmp]# ./fcnt

3:argv[1]:(null)

hello,world~

ret:13

[root@FC5 tmp]# cat pighead.txt

hello,world~

### fcntl之FD\_CLOEXEC

http://blog.csdn.net/ustc\_dylan/article/details/6930189

通过fcntl设置FD\_CLOEXEC标志有什么用？  
close on exec, not on-fork, 意为如果对描述符设置了FD\_CLOEXEC，使用execl执行的程序里，此描述符被关闭，不能再使用它，但是在使用fork调用的子进程中，此描述符并不关闭，仍可使用。  
eg:  
jamie@jamie-laptop:~$ cat test.c  
#include <fcntl.h>  
#include <unistd.h>  
#include <stdio.h>  
#include <string.h>  
  
int main(void){  
        int fd,pid;  
        char buffer[20];  
        fd=open("wo.txt",O\_RDONLY);  
        printf("%d/n",fd);  
        int val=fcntl(fd,F\_GETFD);  
        val|=FD\_CLOEXEC;  
        fcntl(fd,F\_SETFD,val);  
  
        pid=fork();  
        if(pid==0) {  
                //子进程中，此描述符并不关闭，仍可使用  
                char child\_buf[2];  
                memset(child\_buf,0,sizeof(child\_buf) );  
                ssize\_t bytes = read(fd,child\_buf,sizeof(child\_buf)-1 );  
                printf("child, bytes:%d,%s/n/n",bytes,child\_buf);  
  
                //execl执行的程序里，此描述符被关闭，不能再使用它  
                char fd\_str[5];  
                memset(fd\_str,0,sizeof(fd\_str));  
                sprintf(fd\_str,"%d",fd);  
                int ret = execl("./exe1","exe1",fd\_str,NULL);  
                if(-1 == ret)  
                        perror("ececl fail:");  
        }          
  
        waitpid(pid,NULL,0);  
        memset(buffer,0,sizeof(buffer) );  
        ssize\_t bytes = read(fd,buffer,sizeof(buffer)-1 );  
        printf("parent, bytes:%d,%s/n/n",bytes,buffer);  
}  
  
jamie@jamie-laptop:~$ cat exe1.c  
#include <fcntl.h>  
#include <stdio.h>  
#include <assert.h>  
#include <string.h>  
  
int main(int argc, char \*\*args){  
        char buffer[20];  
        int fd = atoi(args[1]);  
        memset(buffer,0,sizeof(buffer) );  
        ssize\_t bytes = read(fd,buffer,sizeof(buffer)-1);  
        if(bytes < 0) {  
                perror("exe1: read fail:");  
                return -1;  
        } else {  
                printf("exe1: read %d,%s/n/n",bytes,buffer);  
        }  
        return 0;  
}  
  
jamie@jamie-laptop:~$ gcc -o exe1 exe1.c  
jamie@jamie-laptop:~$ gcc -o test test.c  
jamie@jamie-laptop:~$ cat wo.txt  
this is a test  
jamie@jamie-laptop:~$ ./test  
3  
child, bytes:1,t                               //子进程中可使用fd  
exe1: read fail:: Bad file descriptor  //execl调用的程序中不能使用fd  
parent, bytes:14,his is a test          //父进程中当然能使用fd

## flock(锁定文件或解除锁定)

|  |  |
| --- | --- |
| **flock(锁定文件或解除锁定)** | |
| 相关函数 | open,fcntl |
| 表头文件 | #include <sys/file.h> |
| 定义函数 | int flock(int fd,int operation); |
| 函数说明 | flock()会依参数operation所指定的方式对参数fd所指的文件做各种锁定或解除锁定的动作。此函数只能锁定整个文件,无法锁定文件的某一区域。  参数operation有下列四种情况：  LOCK\_SH 建立共享锁定。多个进程可同时对同一个文件作共享锁定。  LOCK\_EX 建立互斥锁定。一个文件同时只有一个互斥锁定。  LOCK\_UN 解除文件锁定状态。  LOCK\_NB 无法建立锁定时,此操作可不被阻断,马上返回进程。通常与LOCK\_SH或LOCK\_EX做OR(|)组合。  单一文件无法同时建立共享锁定和互斥锁定,而当使用dup()或fork()时文件描述符不会继承此种锁定。 |
| 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。 |

## sync(将缓冲区数据写回磁盘)

|  |  |
| --- | --- |
| **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。 |

## fsync(将缓冲区数据写回磁盘)

|  |  |
| --- | --- |
| **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)

|  |  |
| --- | --- |
| **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符,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offet允许负值的出现。  下列是教特别的使用方式：  1) 欲将读写位置移到文件开头时：lseek(int fildes,0,SEEK\_SET)；  2) 欲将读写位置移到文件尾时：lseek(int fildes,0,SEEK\_END)；  3) 想要取得目前文件位置时：lseek(int fildes,0,SEEK\_CUR)； |
| 返回值 | 当调用成功时则返回目前的读写位置,也就是距离文件开头多少个字节。若有错误则返回-1,errno会存放错误代码。 |
| 附加说明 | Linux系统不允许lseek()对tty装置作用,此项动作会令lseek()返回ESPIPE。 |
| 范例 |  |
| 参考本函数说明 | |

## mkstemp(建立唯一的临时文件)

|  |  |
| --- | --- |
| **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符会返回。  文件顺利打开后返回可读写的文件描述符。如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 |  |
| template = template-lgZcbo | |

# 文件内容操作

## fopen(打开文件)

|  |  |
| --- | --- |
| **fopen(打开文件)** | |
| 相关函数 | open,fclose |
| 表头文件 | #include <stdio.h> |
| 定义函数 | FILE \*fopen(const char \*path,const char \*mode); |
| 函数说明 | 参数path字符串包含欲打开的文件路径及文件名,参数mode字符串则代表着流形态。mode有下列几种形态字符串：  r 打开只读文件,该文件必须存在。  r+ 打开可读写的文件,该文件必须存在。  w 打开只写文件,若文件存在则文件长度清为0,即该文件内容会消失。若文件不存在则  建立该文件。  w+ 打开可读写文件,若文件存在则文件长度清为零,即该文件内容会消失。若文件不存在则建立该文件。  a 以附加的方式打开只写文件。若文件不存在,则会建立该文件,如果文件存在,写入的数据会被加到文件尾,即文件原先的内容会被保留。  a+ 以附加方式打开可读写的文件。若文件不存在,则会建立该文件,如果文件存在,写入的数据会被加到文件尾后,即文件原先的内容会被保留。  上述的形态字符串都可以再加一个b字符,如rb、w+b或ab＋等组合,加入b字符用来告诉函数库打开的文件为二进制文件,而非纯文字文件。不过在POSIX系统,包含Linux都会忽略该字符。由fopen()所建立的新文件会具有S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IWGRP|S\_IROTH|S\_IWOTH(0666)权限,此文件权限也会参考umask值。 |
| 返回值 | 文件顺利打开后,指向该流的文件指针就会被返回。若果文件打开失败则返回NULL,并把错误代码存在errno中。 |
| 附加说明 | 一般而言,开文件后会作一些文件读取或写入的动作,若开文件失败,接下来的读写动作也无法顺利进行,所以在fopen()后请作错误判断及处理。 |
| 范例 |  |
| #include <stdio.h>  main(){  FILE \*fp;  fp=fopen(“noexist”,”a+”);  if(fp= =NULL) return;  fclose(fp);  } | |

### fopen/open区别

http://blog.sina.com.cn/s/blog\_61080d360100jnt2.html

UNIX环境下的C对二进制流文件的读写有两套班子：1)fopen,fread,fwrite; 2)open,read,write  
这里简单的介绍一下他们的区别。  
1.fopen系列是标准的**C库函数**；open系列是UNIX系统里的**系统调用**，是POSIX定义的。

fopen系列更具有可移植性(抽象层)；而open系列只能用在 POSIX的操作系统上。

2.使用fopen系列函数时要定义一个指代文件的对象，被称为“文件句柄”(file handler)，是一个FILE\*结构体；而open系列使用的是一个被称为“文件描述符”(file descriptor)的int型整数。

3.fopen系列是级别较高的I/O，读写时使用缓冲；而open系列相对低层，更接近操作系统，读写时没有缓冲。由于能更多地与操作系统打交道，open系列可以访问更改一些fopen系列无法访问的信息，如查看文件的读写权限。这些额外的功能通常因系统而异。

4.使用fopen系列函数需要"#include <sdtio.h>"；使用open系列函数需要"#include <fcntl.h>"，链接时要使用libc(-lc)。

小结：  
总的来说，为了使程序获得更好的可移植性，未到非得使用一些open系列无法实现的功能的情况下，fopen系列是首选。

**open和fopen的区别：**

1,open返回一个**文件描述符(int)**，fopen返回一个**文件指针(FILE\*)**。  
2,open是**POSIX中定义**的，fopen是**标准c里定义**的。  
3,open无缓冲，fopen有缓冲。open属于低级IO，fopen是高级IO。  
4,open与read/write等配合使用，fopen与fread/fwrite等配合使用。  
5,fopen不能指定要创建文件的权限，open可以指定权限。  
6,fread可以读一个结构，read在linux/unix中读二进制与普通文件没有区别。  
7,linux/unix中任何设备都是文件,都可以用open,read。

8,fopen是在open的基础上扩充而来的，在大多数情况下，用fopen。

如果文件的大小是8k。  
你如果用read/write，且只分配了2k的缓存，则要将此文件读出需要做4次系统调用来实际从磁盘上读出。  
如果你用fread/fwrite，则系统自动分配缓存，则读出此文件只要一次系统调用从磁盘上读出。  
也就是用read/write要读4次磁盘，而用fread/fwrite则只要读1次磁盘。效率比read/write要高4倍。  
如果程序对内存有限制，则用read/write比较好。  
都用fread 和fwrite,它自动分配缓存,速度会很快,比自己来做要简单。如果要处理一些特殊的描述符,用read和write,如套接口,管道之类的系统调用write的效率取决于你buf的大小和你要写入的总数量，如果buf太小，你进入内核空间的次数大增，效率就低下。而fwrite会替你做缓存，减少了实际出现的系统调用，所以效率比较高。

## fclose(关闭文件)

|  |  |
| --- | --- |
| **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()。 | |

## freopen(打开文件)

|  |  |
| --- | --- |
| **freopen(打开文件)** | |
| 相关函数 | fopen,fclose |
| 表头文件 | #include <stdio.h> |
| 定义函数 | FILE \*freopen(const char \*path,const char \*mode,FILE \*stream); |
| 函数说明 | 参数path字符串包含欲打开的文件路径及文件名,参数mode请参考fopen()说明。参数stream为已打开的文件指针。freopen()会将原stream所打开的文件流关闭,然后打开参数path的文件。 |
| 返回值 | 文件顺利打开后,指向该流的文件指针就会被返回。如果文件打开失败则返回NULL,并把错误代码存在errno中。 |
| 范例 |  |
| #include <stdio.h>  main(){  FILE \*fp;  fp=fopen(“/etc/passwd”,”r”);  fp=freopen(“/etc/group”,”r”,fp);  fclose(fp);  } | |

## fdopen(将文件描述符转为文件指针)

|  |  |
| --- | --- |
| **fdopen(将文件描述符转为文件指针)** | |
| 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的文件描述符,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符读写模式相同。关于mode字符串格式请参考fopen()。 |
| 返回值 | 转换成功时返回指向该流的文件指针。失败则返回NULL,并把错误代码存在errno中。 |
| 范例 |  |
| #include <stdio.h>  main(){  FILE \*fp =fdopen(0,”w+”);  fprintf(fp,”%s\n”,”hello!”);  fclose(fp);  } | |
| 执行 |  |
| hello! | |

## fread(从文件流读取数据)

|  |  |
| --- | --- |
| **fread(从文件流读取数据)** | |
| 相关函数 | fopen,fwrite,fseek,fscanf |
| 表头文件 | #include <stdio.h> |
| 定义函数 | size\_t fread(void \*ptr,size\_t size,size\_t nmemb,FILE \*stream); |
| 函数说明 | fread()用来从文件流中读取数据。参数stream为已打开的文件指针,参数ptr指向欲存放读取进来的数据空间,读取的字符数以参数size\_t nmemb来决定。fread()会返回实际读取到的nmemb数目,如果此值比参数nmemb来得小,则代表可能读到了文件尾或有错误发生,这时必须用feof()或ferror()来决定发生什么情况。 |
| 返回值 | 返回实际读取到的nmemb数目。 |
| 附加说明 |  |
| 范例 |  |
| #include <stdio.h>  #define nmemb 3  struct test{  char name[20];  int size;  }s[nmemb];  main(){  FILE \*stream;  int i;  stream = fopen(“/tmp/fwrite”,”r”);  fread(s,sizeof(struct test),nmemb,stream);  fclose(stream);  for(i=0;i<nmemb;i++)  printf(“name[%d]=%-20s:size[%d]=%d\n”,i,s[i].name,i,s[i].size);  } | |
| 执行 |  |
| name[0]=Linux! size[0]=6  name[1]=FreeBSD! size[1]=8  name[2]=Windows2000 size[2]=11 | |

## fwrite(将数据写至文件流)

|  |  |
| --- | --- |
| **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\_t 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) {strcoy(s[x].name,y);s[x].size=strlen(y);}  #define nmemb 3  struct test{  char name[20];  int size;  }s[nmemb];  main(){  FILE \*stream;  set\_s(0,”Linux!”);  set\_s(1,”FreeBSD!”);  set\_s(2,”Windows2000.”);  stream=fopen(“/tmp/fwrite”,”w”);  fwrite(s,sizeof(struct test),nmemb,stream);  fclose(stream);  } | |
| 执行 |  |
| 参考fread()。 | |

## fseek(移动文件流的读写位置)

|  |  |
| --- | --- |
| **fseek(移动文件流的读写位置)** | |
| 相关函数 | rewind,ftell,fgetpos,fsetpos,lseek |
| 表头文件 | #include <stdio.h> |
| 定义函数 | int fseek(FILE \*stream,long offset,int whence); |
| 函数说明 | fseek()用来移动文件流的读写位置。参数stream为已打开的文件指针,参数offset为根据参数whence来移动读写位置的位移数。参数whence为下列其中一种：SEEK\_SET从距文件开头offset位移量为新的读写位置。SEEK\_CUR以目前的读写位置往后增加offset个位移量。SEEK\_END将读写位置指向文件尾后再增加offset个位移量。  当whence值为SEEK\_CUR或SEEK\_END时,参数offset允许负值的出现。下列是较特别的使用方式：  1)欲将读写位置移动到文件开头时：fseek(FILE \*stream,0,SEEK\_SET);  2)欲将读写位置移动到文件尾时：fseek(FILE \*stream,0,0SEEK\_END); |
| 返回值 | 当调用成功时则返回0,若有错误则返回-1,errno会存放错误代码。 |
| 附加说明 | fseek()不像lseek()会返回读写位置,因此必须使用ftell()来取得目前读写的位置。 |
| 范例 |  |
| #include <stdio.h>  main(){  FILE \*stream;  long offset;  fpos\_t pos;  stream=fopen(“/etc/passwd”,”r”);  fseek(stream,5,SEEK\_SET);  printf(“offset=%d\n”,ftell(stream));  rewind(stream);  fgetpos(stream,&pos);  printf(“offset=%d\n”,pos);  pos=10;  fsetpos(stream,&pos);  printf(“offset = %d\n”,ftell(stream));  fclose(stream);  } | |
| 执行 |  |
| offset = 5  offset =0  offset=10 | |

## ftell(取得文件流的读取位置)

|  |  |
| --- | --- |
| **ftell(取得文件流的读取位置)** | |
| 相关函数 | fseek,rewind,fgetpos,fsetpos |
| 表头文件 | #include <stdio.h> |
| 定义函数 | long ftell(FILE \*stream); |
| 函数说明 | ftell()用来取得文件流目前的读写位置。参数stream为已打开的文件指针。 |
| 返回值 | 当调用成功时则返回目前的读写位置,若有错误则返回-1,errno会存放错误代码。 |
| 错误代码 | EBADF 参数stream无效或可移动读写位置的文件流。 |
| 范例 |  |
| 参考fseek()。 | |

## rewind(重设文件流的读写位置为文件开头)

|  |  |
| --- | --- |
| **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 |  |
| 参考fseek() | |

## feof(检查文件流是否读到了文件尾)

|  |  |
| --- | --- |
| **feof(检查文件流是否读到了文件尾)** | |
| 相关函数 | fopen,fgetc,fgets,fread |
| 表头文件 | #include <stdio.h> |
| 定义函数 | int feof(FILE \*stream); |
| 函数说明 | feof()用来侦测是否读取到了文件尾,尾数stream为fopen()所返回之文件指针。如果已到文件尾则返回非零值,其他情况返回0。 |
| 返回值 | 返回非零值代表已到达文件尾。 |

## fileno(返回文件流所使用的文件描述符)

|  |  |
| --- | --- |
| **fileno(返回文件流所使用的文件描述符)** | |
| 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符。 |
| 返回值 | 返回文件描述符。 |
| 范例 |  |
| #include <stdio.h>  main(){  FILE \*fp;  int fd;  fp=fopen(“/etc/passwd”,”r”);  fd=fileno(fp);  printf(“fd=%d\n”,fd);  fclose(fp);  } | |
| 执行 |  |
| fd=3 | |

## fflush(更新缓冲区)

|  |  |
| --- | --- |
| **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()。 |

## clearerr(清除文件流的错误旗标)

|  |  |
| --- | --- |
| **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 |

## fgetc(由文件中读取一个字符)

|  |  |
| --- | --- |
| **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 |  |
| #include <stdio.h>  main(){  FILE \*fp;  int c;  fp=fopen(“exist”,”r”);  while((c=fgetc(fp))!=EOF)  printf(“%c”,c);  fclose(fp);  } | |

## fgets(由文件中读取一字符串)

|  |  |
| --- | --- |
| **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则表示有错误发生。 |
| 范例 |  |
| #include <stdio.h>  main(){  char s[80];  fputs(fgets(s,80,stdin),stdout);  } | |
| 执行 |  |
| this is a test /\*输入\*/  this is a test /\*输出\*/ | |

## fputc(将一指定字符写入文件流中)

|  |  |
| --- | --- |
| **fputc(将一指定字符写入文件流中)** | |
| 相关函数 | fopen,fwrite,fscanf,putc |
| 表头文件 | #include <stdio.h> |
| 定义函数 | int fputc(int c,FILE \*stream); |
| 函数说明 | fputc会将参数c转为unsigned char后写入参数stream指定的文件中。 |
| 返回值 | fputc()会返回写入成功的字符,即参数c。若返回EOF则代表写入失败。 |
| 范例 |  |
| #include <stdio.h>  main(){  FILE \*fp;  char a[26]=”abcdefghijklmnopqrstuvwxyz”;  int i;  fp= fopen(“noexist”,”w”);  for(i=0;i<26;i++)  fputc(a[i],fp);  fclose(fp); } | |
|  | |

## fputs(将一指定的字符串写入文件内)

|  |  |
| --- | --- |
| **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()。 | |

## getc(由文件中读取一个字符)

|  |  |
| --- | --- |
| **getc(由文件中读取一个字符)** | |
| 相关函数 | read,fopen,fread,fgetc |
| 表头文件 | #include <stdio.h> |
| 定义函数 | int getc(FILE \*stream); |
| 函数说明 | getc()用来从参数stream所指的文件中读取一个字符。若读到文件尾而无数据时便返回EOF。虽然getc()与fgetc()作用相同,但getc()为宏定义,非真正的函数调用。 |
| 返回值 | getc()会返回读取到的字符,若返回EOF则表示到了文件尾。 |
| 范例 |  |
| 参考fgetc()。 | |

## putc(将一指定字符写入文件中)

|  |  |
| --- | --- |
| **putc(将一指定字符写入文件中)** | |
| 相关函数 | fopen,fwrite,fscanf,fputc |
| 表头文件 | #include <stdio.h> |
| 定义函数 | int putc(int c,FILE \*stream); |
| 函数说明 | putc()会将参数c转为unsigned char后写入参数stream指定的文件中。虽然putc()与fputc()作用相同,但putc()为宏定义,非真正的函数调用。 |
| 返回值 | putc()会返回写入成功的字符,即参数c。若返回EOF则代表写入失败。 |
| 范例 |  |
| 参考fputc()。 | |

## ungetc(将指定字符写回文件流中)

|  |  |
| --- | --- |
| **ungetc(将指定字符写回文件流中)** | |
| 相关函数 | fputc,getchar,getc |
| 表头文件 | #include <stdio.h> |
| 定义函数 | int ungetc(int c,FILE \*stream); |
| 函数说明 | ungetc()将参数c字符写回参数stream所指定的文件流。这个写回的字符会由下一个读取文件流的函数取得。 |
| 返回值 | 成功则返回c字符,若有错误则返回EOF。 |

## getchar(由标准输入设备内读进一字符)

|  |  |
| --- | --- |
| **getchar(由标准输入设备内读进一字符)** | |
| 相关函数 | fopen,fread,fscanf,getc |
| 表头文件 | #include <stdio.h> |
| 定义函数 | int getchar(void); |
| 函数说明 | getchar()用来从标准输入设备中读取一个字符。然后将该字符从unsigned char转换成int后返回。 |
| 返回值 | getchar()会返回读取到的字符,若返回EOF则表示有错误发生。 |
| 附加说明 | getchar()非真正函数,而是getc(stdin)宏定义。 |
| 范例 |  |
| #include <stdio.h>  main(){  int c,i;  for(i=0li<5;i++) {  c=getchar();  putchar(c);  }  } | |
| 执行 |  |
| 1234 /\*输入\*/  1234 /\*输出\*/ | |

## putchar(将指定的字符写到标准输出设备)

|  |  |
| --- | --- |
| **putchar(将指定的字符写到标准输出设备)** | |
| 相关函数 | fopen,fwrite,fscanf,fputc |
| 表头文件 | #include <stdio.h> |
| 定义函数 | int putchar(int c); |
| 函数说明 | putchar()用来将参数c字符写到标准输出设备。 |
| 返回值 | putchar()会返回输出成功的字符,即参数c。若返回EOF则代表输出失败。 |
| 附加说明 | putchar()非真正函数,而是putc(c,stdout)宏定义。 |
| 范例 |  |
| 参考getchar()。 | |

## gets(由标准输入设备内读进一字符串)

|  |  |
| --- | --- |
| **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()用来从标准设备读入字符并存到参数s所指的内存空间,直到出现换行字符或读到文件尾为止,最后加上NULL作为字符串结束。 |
| 返回值 | gets()若成功则返回s指针,返回NULL则表示有错误发生。 |
| 附加说明 | 由于gets()无法知道字符串s的大小,必须遇到换行字符或文件尾才会结束输入,因此容易造成缓冲溢出的安全性问题。建议使用fgets()取代。 |
| 范例 |  |
| 参考fgets() | |
|  | |

## mktemp(产生唯一的临时文件名)

|  |  |
| --- | --- |
| **mktemp(产生唯一的临时文件名)** | |
| 相关函数 | tmpfile |
| 表头文件 | #include <stdlib.h> |
| 定义函数 | char \*mktemp(char \*template); |
| 函数说明 | mktemp()用来产生唯一的临时文件名。参数template所指的文件名称字符串中最后六个字符必须是XXXXXX。产生后的文件名会借字符串指针返回。 |
| 返回值 | 文件顺利打开后,指向该流的文件指针就会被返回。如果文件打开失败则返回NULL,并把错误代码存在errno中。 |
| 附加说明 | 参数template所指的文件名称字符串必须声明为数组,如 char template[ ]=”template-XXXXXX”；不可用char \*template=”template-XXXXXX”； |
| 范例 |  |
| #include <stdlib.h>  main(){  char template[ ]=”template-XXXXXX”;  mktemp(template);  printf(“template=%s\n”,template);  } | |

## setbuf(设置文件流的缓冲区)

|  |  |
| --- | --- |
| **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)

|  |  |
| --- | --- |
| **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)

|  |  |
| --- | --- |
| **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)

|  |  |
| --- | --- |
| **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 |

# 文件属性操作

## access(判断是否具有存取文件的权限)

|  |  |
| --- | --- |
| **access(判断是否具有存取文件的权限)** | |
| 相关函数 | stat,open,chmod,chown,setuid,setgid |
| 表头文件 | #include <unistd.h> |
| 定义函数 | int access(const char \*pathname,int mode); |
| 函数说明 | access()会检查是否可以读/写某一已存在的文件。参数mode有几种情况组合,R\_OK, W\_OK,X\_OK和F\_OK。R\_OK,W\_OK与X\_OK用来检查文件是否具有读取、写入和执行的权限。F\_OK则是用来判断该文件是否存在。由于access()只作权限的核查,并不理会文件形态或文件内容,因此,如果一目录表示为“可写入”,表示可以在该目录中建立新文件等操作,而非意味此目录可以被当做文件处理。例如,你会发现DOS的文件都具有“可执行”权限,但用execve()执行时则会失败。 |
| 返回值 | 若所有欲查核的权限都通过了检查则返回0值,表示成功,只要有一权限被禁止则返回-1。 |
| 错误代码 | EACCESS 参数pathname所指定的文件不符合所要求测试的权限。  EROFS 欲测试写入权限的文件存在于只读文件系统内。  EFAULT 参数pathname指针超出可存取内存空间。  EINVAL 参数mode不正确。  ENAMETOOLONG 参数pathname太长。  ENOTDIR 参数pathname为一目录。  ENOMEM 核心内存不足  ELOOP 参数pathname有过多符号连接问题。  EIO I/O存取错误。 |
| 附加说明 | 使用access()作用户认证方面的判断要特别小心,例如在access()后再做open()的空文件可能会造成系统安全上的问题。 |
| 范例 |  |
| /\* 判断是否允许读取/etc/passwd \*/  #include <unistd.h>  int main(){  if(access(“/etc/passwd”,R\_OK) = =0)  printf(“/etc/passwd can be read\n”);  } | |
| 执行 |  |
| /etc/passwd can be read | |

## chdir(改变当前的工作目录)

|  |  |
| --- | --- |
| **chdir(改变当前的工作目录)** | |
| 相关函数 | getcwd,chroot |
| 表头文件 | #include <unistd.h> |
| 定义函数 | int chdir(const char \*path); |
| 函数说明 | chdir()用来将当前的工作目录改变成以参数path所指的目录。 |
| 返回值 | 执行成功则返回0,失败返回-1,errno为错误代码。 |
| 范例 |  |
| #include <unistd.h>  main(){  chdir(“/tmp”);  printf(“current working directory: %s\n”,getcwd(NULL,NULL));  } | |
| 执行 |  |
| current working directory :/tmp | |

## chmod(改变文件的权限)

|  |  |
| --- | --- |
| **chmod(改变文件的权限)** | |
| 相关函数 | fchmod,stat,open,chown |
| 表头文件 | #include <sys/types.h>  #include <sys/stat.h> |
| 定义函数 | int chmod(const char \*path,mode\_t mode); |
| 函数说明 | chmod()会依参数mode权限来更改参数path指定文件的权限。参数mode有下列数种组合：  S\_ISUID 04000 文件的(set user-id on execution)位  S\_ISGID 02000 文件的(set group-id on execution)位  S\_ISVTX 01000 文件的sticky位  S\_IRUSR(S\_IREAD) 00400 文件所有者具可读取权限  S\_IWUSR(S\_IWRITE)00200 文件所有者具可写入权限  S\_IXUSR(S\_IEXEC) 00100 文件所有者具可执行权限  S\_IRGRP 00040 用户组具可读取权限  S\_IWGRP 00020 用户组具可写入权限  S\_IXGRP 00010 用户组具可执行权限  S\_IROTH 00004 其他用户具可读取权限  S\_IWOTH 00002 其他用户具可写入权限  S\_IXOTH 00001 其他用户具可执行权限  只有该文件的所有者或有效用户识别码为0,才可以修改该文件权限。基于系统安全,如果欲将数据写入一执行文件,而该执行文件具有S\_ISUID或S\_ISGID权限,则这两个位会被清除。如果一目录具有S\_ISUID位权限,表示在此目录下只有该文件的所有者或root可以删除该文件。 |
| 返回值 | 权限改变成功返回0,失败返回-1,错误原因存于errno。 |
| 错误代码 | EPERM 进程的有效用户识别码与欲修改权限的文件拥有者不同,而且也不具root权限。  EACCESS 参数path所指定的文件无法存取。  EROFS 欲写入权限的文件存在于只读文件系统内。  EFAULT 参数path指针超出可存取内存空间。  EINVAL 参数mode不正确  ENAMETOOLONG 参数path太长  ENOENT 指定的文件不存在  ENOTDIR 参数path路径并非一目录  ENOMEM 核心内存不足  ELOOP 参数path有过多符号连接问题。  EIO I/O存取错误 |
| 范例 |  |
| /\*将/etc/passwd文件权限设成S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IROTH \*/  #include <sys/types.h>  #include <sys/stat.h>  main(){  chmod(“/etc/passwd”,S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IROTH);  } | |

## umask(设置建立新文件时的权限遮罩)

|  |  |
| --- | --- |
| **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--。  注:**umask是从权限中"拿走"相应的位,即被屏蔽的权限,真正的权限是mode&~umask。** |
| 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。 |

## chown(改变文件的所有者)

|  |  |
| --- | --- |
| **chown(改变文件的所有者)** | |
| 相关函数 | fchown,lchown,chmod |
| 表头文件 | #include <sys/types.h>  #include <unistd.h> |
| 定义函数 | int chown(const char \*path, uid\_t owner,gid\_t group); |
| 函数说明 | chown()会将参数path指定文件的所有者变更为参数owner代表的用户,而将该文件的组变更为参数group组。如果参数owner或group为-1,对应的所有者或组不会有所改变。root与文件所有者皆可改变文件组,但所有者必须是参数group组的成员。当root用chown()改变文件所有者或组时,该文件若具有S\_ISUID或S\_ISGID权限,则会清除此权限位,此外如果具有S\_ISGID权限但不具S\_IXGRP位,则该文件会被强制锁定,文件模式会保留。 |
| 返回值 | 成功则返回0,失败返回-1,错误原因存于errno。 |
| 错误代码 | 参考chmod()。 |
| 范例 |  |
| /\*将/etc/passwd的所有者和组都设为root \*/  #include <sys/types.h>  #include <unistd.h>  main(){  chown(“/etc/passwd”,0,0);  } | |

## chroot(改变根目录)

|  |  |
| --- | --- |
| **chroot(改变根目录)** | |
| 相关函数 | chdir |
| 表头文件 | #include <unistd.h> |
| 定义函数 | int chroot(const char \*path); |
| 函数说明 | chroot()用来改变根目录为参数path所指定的目录。只有超级用户才允许改变根目录,子进程将继承新的根目录。 |
| 返回值 | 调用成功则返回0,失败则返-1,错误代码存于errno。 |
| 错误代码 | EPERM 权限不足,无法改变根目录。  EFAULT 参数path指针超出可存取内存空间。  ENAMETOOLONG 参数path太长。  ENOTDIR 路径中的目录存在但却非真正的目录。  EACCESS 存取目录时被拒绝  ENOMEM 核心内存不足。  ELOOP 参数path有过多符号连接问题。  EIO I/O存取错误。 |
| 范例 |  |
| /\*将根目录改为/tmp ,并将工作目录切换至/tmp \*/  #include <unistd.h>  main(){  chroot(“/tmp”);  chdir(“/”);  } | |

## closedir(关闭目录)

|  |  |
| --- | --- |
| **closedir(关闭目录)** | |
| 相关函数 | opendir |
| 表头文件 | #include <sys/types.h>  #include <dirent.h> |
| 定义函数 | int closedir(DIR \*dir); |
| 函数说明 | closedir()关闭参数dir所指的目录流。 |
| 返回值 | 关闭成功则返回0,失败返回-1,错误原因存于errno中。 |
| 错误代码 | EBADF 参数dir为无效的目录流 |
| 范例 |  |
| 参考readir()。 | |

## fchdir(改变当前的工作目录)

|  |  |
| --- | --- |
| **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所指的文件描述符。 |
| 返回值 | 执行成功则返回0,失败返回-1,errno为错误代码。 |
| 附加说明 |  |
| 范例 |  |
| #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <unistd.h>  main(){  int fd;  fd = open(“/tmp”,O\_RDONLY);  fchdir(fd);  printf(“current working directory : %s \n”,getcwd(NULL,NULL));  close(fd);  } | |
| 执行 |  |
| current working directory : /tmp | |

## fchmod(改变文件的权限)

|  |  |
| --- | --- |
| **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符。  参数mode请参考chmod()。 |
| 返回值 | 权限改变成功则返回0,失败返回-1,错误原因存于errno。 |
| 错误原因 | EBADF 参数fildes为无效的文件描述符。  EPERM 进程的有效用户识别码与欲修改权限的文件所有者不同,而且也不具root权限。  EROFS 欲写入权限的文件存在于只读文件系统内。  EIO I/O存取错误。 |
| 范例 |  |
| #include <sys/stat.h>  #include <fcntl.h>  main(){  int fd;  fd = open(“/etc/passwd”,O\_RDONLY);  fchmod(fd,S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IROTH);  close(fd);  } | |

## fchown(改变文件的所有者)

|  |  |
| --- | --- |
| **fchown(改变文件的所有者)** | |
| 相关函数 | chown,lchown,chmod |
| 表头文件 | #include <sys/types.h>  #include <unistd.h> |
| 定义函数 | int fchown(int fd,uid\_t owner,gid\_t group); |
| 函数说明 | fchown()会将参数fd指定文件的所有者变更为参数owner代表的用户,而将该文件的组变更为参数group组。如果参数owner或group为-1,对映的所有者或组有所改变。参数fd为已打开的文件描述符。当root用fchown()改变文件所有者或组时,该文件若具S\_ISUID或S\_ISGID权限,则会清除此权限位。 |
| 返回值 | 成功则返回0,失败则返回-1,错误原因存于errno。 |
| 错误代码 | EBADF 参数fd文件描述符为无效的或该文件已关闭。  EPERM 进程的有效用户识别码与欲修改权限的文件所有者不同,而且也不具root权限,或是参数owner、group不正确。  EROFS 欲写入的文件存在于只读文件系统内。  ENOENT 指定的文件不存在  EIO I/O存取错误 |
| 范例 |  |
| #include <sys/types.h>  #include <unistd.h>  #include <fcntl.h>  main(){  int fd;  fd = open(“/etc/passwd”,O\_RDONLY);  chown(fd,0,0);  close(fd);  } | |

## ftruncate(改变文件大小)

|  |  |
| --- | --- |
| **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符,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文件描述符为无效的或该文件已关闭。  EINVAL 参数fd为一socket并非文件,或是该文件并非以写入模式打开。 |

## getcwd(取得当前的工作目录)

|  |  |
| --- | --- |
| **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程度来决定所配置的内存大小,进程可以在使用完此字符串后利用free()来释放此空间。 |
| 返回值 | 执行成功则将结果复制到参数buf所指的内存空间,或是返回自动配置的字符串指针。失败返回NULL,错误代码存于errno。 |
| 范例 |  |
| #include <unistd.h>  main(){  char buf[80];  getcwd(buf,sizeof(buf));  printf(“current working directory : %s\n”,buf);  } | |
| 执行 |  |
| current working directory :/tmp | |

## link(建立文件连接)

|  |  |
| --- | --- |
| **link(建立文件连接)** | |
| 相关函数 | symlink,unlink |
| 表头文件 | #include <unistd.h> |
| 定义函数 | int link(const char \*oldpath,const char \*newpath); |
| 函数说明 | link()以参数newpath指定的名称来建立一个新的连接(硬连接)到参数oldpath所指定的已存在文件。如果参数newpath指定的名称为一已存在的文件则不会建立连接。 |
| 返回值 | 成功则返回0,失败返回-1,错误原因存于errno。 |
| 附加说明 | link()所建立的硬连接无法跨越不同文件系统,如果需要请改用symlink()。 |
| 错误代码 | EXDEV 参数oldpath与newpath不是建立在同一文件系统。  EPERM 参数oldpath与newpath所指的文件系统不支持硬连接  EROFS 文件存在于只读文件系统内  EFAULT 参数oldpath或newpath指针超出可存取内存空间。  ENAMETOLLONG 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”);  } | |

## symlink(建立文件符号连接)

|  |  |
| --- | --- |
| **symlink(建立文件符号连接)** | |
| 相关函数 | link,unlink |
| 表头文件 | #include <unistd.h> |
| 定义函数 | int symlink( const char \*oldpath,const char \*newpath); |
| 函数说明 | symlink()以参数newpath指定的名称来建立一个新的连接(符号连接)到参数oldpath所指定的已存在文件。参数oldpath指定的文件不一定要存在,如果参数newpath指定的名称为一已存在的文件则不会建立连接。 |
| 返回值 | 成功则返回0,失败返回-1,错误原因存于errno |
| 错误代码 | EPERM 参数oldpath与newpath所指的文件系统不支持符号连接  EROFS 欲测试写入权限的文件存在于只读文件系统内  EFAULT 参数oldpath或newpath指针超出可存取内存空间。  ENAMETOOLONG 参数oldpath或newpath太长  ENOMEM 核心内存不足  EEXIST 参数newpath所指的文件名已存在。  EMLINK 参数oldpath所指的文件已达到最大连接数目  ELOOP 参数pathname有过多符号连接问题  ENOSPC 文件系统的剩余空间不足  EIO I/O存取错误 |
| 范例 |  |
| #include <unistd.h>  main() {  symlink(“/etc/passwd”,”pass”);  } | |

## unlink(删除文件)

|  |  |
| --- | --- |
| **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符皆关闭后才会删除。如果参数pathname为一符号连接,则此连接会被删除。 |
| 返回值 | 成功则返回0,失败返回-1,错误原因存于errno |
| 错误代码 | EROFS 文件存在于只读文件系统内。  EFAULT 参数pathname指针超出可存取内存空间  ENAMETOOLONG 参数pathname太长  ENOMEM 核心内存不足  ELOOP 参数pathname有过多符号连接问题  EIO I/O存取错误 |

## readlink(取得符号连接所指的文件)

|  |  |
| --- | --- |
| **readlink(取得符号连接所指的文件)** | |
| 相关函数 | stat,lstat,symlink |
| 表头文件 | #include <unistd.h> |
| 定义函数 | int readlink(const char \*path ,char \*buf,size\_t bufsiz); |
| 函数说明 | readlink()会将参数path的符号连接内容存到参数buf所指的内存空间,返回的内容不是以NULL作字符串结尾,但会将字符串的字符数返回。若参数bufsiz小于符号连接的内容长度,过长的内容会被截断。 |
| 返回值 | 执行成功则传符号连接所指的文件路径字符串,失败则返回-1,错误代码存于errno。 |
| 错误代码 | EACCESS 取文件时被拒绝,权限不够  EINVAL 参数bufsiz为负数  EIO I/O存取错误。  ELOOP 欲打开的文件有过多符号连接问题。  ENAMETOOLONG 参数path的路径名称太长  ENOENT 参数path所指定的文件不存在  ENOMEM 核心内存不足  ENOTDIR 参数path路径中的目录存在但却非真正的目录。 |

## stat(取得文件状态)

|  |  |
| --- | --- |
| **stat(取得文件状态)** | |
| 相关函数 | fstat,lstat,chmod,chown,readlink,utime |
| 表头文件 | #include <sys/stat.h>  #include <unistd.h> |
| 定义函数 | int stat(const char \*file\_name,struct stat \*buf); |
| 函数说明 | stat()用来将参数file\_name所指的文件状态,复制到参数buf所指的结构中。下面是struct stat内各参数的说明：  struct stat{  dev\_t st\_dev; /\*device\*/  ino\_t st\_ino; /\*inode\*/  mode\_t st\_mode; /\*protection\*/  nlink\_t st\_nlink; /\*number of hard links \*/  uid\_t st\_uid; /\*user ID of owner\*/  gid\_t st\_gid; /\*group ID of owner\*/  dev\_t st\_rdev; /\*device type \*/  off\_t st\_size; /\*total size, in bytes\*/  unsigned long st\_blksize; /\*blocksize for filesystem I/O \*/  unsigned long st\_blocks; /\*number of blocks allocated\*/  time\_t st\_atime; /\*time of lastaccess\*/  time\_t st\_mtime; /\*time of last modification \*/  time\_t st\_ctime; /\*time of last change \*/  };  st\_dev 文件的设备编号  st\_ino 文件的i-node  st\_mode 文件的类型和存取的权限  st\_nlink 连到该文件的硬连接数目,刚建立的文件值为1。  st\_uid 文件所有者的用户识别码  st\_gid 文件所有者的组识别码  st\_rdev 若此文件为装置设备文件,则为其设备编号  st\_size 文件大小,以字节计算  st\_blksize 文件系统的I/O缓冲区大小。  st\_blcoks 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tructate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 scoket  S\_IFLNK 0120000 符号连接  S\_IFREG 0100000 一般文件  S\_IFBLK 0060000 区块装置  S\_IFDIR 0040000 目录  S\_IFCHR 0020000 字符装置  S\_IFIFO 0010000 先进先出  S\_ISUID 04000 文件的(set user-id on execution)位  S\_ISGID 02000 文件的(set group-id on execution)位  S\_ISVTX 01000 文件的sticky位  S\_IRUSR(S\_IREAD) 00400 文件所有者具可读取权限  S\_IWUSR(S\_IWRITE)00200 文件所有者具可写入权限  S\_IXUSR(S\_IEXEC) 00100 文件所有者具可执行权限  S\_IRGRP 00040 用户组具可读取权限  S\_IWGRP 00020 用户组具可写入权限  S\_IXGRP 00010 用户组具可执行权限  S\_IROTH 00004 其他用户具可读取权限  S\_IWOTH 00002 其他用户具可写入权限  S\_IXOTH 00001 其他用户具可执行权限  上述的文件类型在POSIX中定义了检查这些类型的宏定义：  S\_ISLNK (st\_mode) 判断是否为符号连接  S\_ISREG (st\_mode) 是否为一般文件  S\_ISDIR (st\_mode)是否为目录  S\_ISCHR (st\_mode)是否为字符装置文件  S\_ISBLK (s3e) 是否为先进先出  S\_ISSOCK (st\_mode) 是否为socket  若一目录具有sticky位(S\_ISVTX),则表示在此目录下的文件只能被该文件所有者、此目录所有者或root来删除或改名。 |
| 返回值 | 执行成功则返回0,失败返回-1,错误代码存于errno |
| 错误代码 | ENOENT 参数file\_name指定的文件不存在  ENOTDIR 路径中的目录存在但却非真正的目录  ELOOP 欲打开的文件有过多符号连接问题,上限为16符号连接  EFAULT 参数buf为无效指针,指向无法存在的内存空间  EACCESS 存取文件时被拒绝  ENOMEM 核心内存不足  ENAMETOOLONG 参数file\_name的路径名称太长 |
| 范例 |  |
| #include <sys/stat.h>  #include <unistd.h>  mian(){  struct stat buf;  stat(“/etc/passwd”,&buf);  printf(“/etc/passwd file size = %d \n”,buf.st\_size);  } | |
| 执行 |  |
| /etc/passwd file size = 705 | |

## lstat(由文件描述符取得文件状态)

|  |  |
| --- | --- |
| **lstat(由文件描述符取得文件状态)** | |
| 相关函数 | stat,fstat,chmod,chown,readlink,utime |
| 表头文件 | #include <sys/stat.h>  #include <unistd.h> |
| 定义函数 | int lstat(const char \*file\_name.struct stat \*buf); |
| 函数说明 | lstat()与stat()作用完全相同,都是取得参数file\_name所指的文件状态,其差别在于,当文件为符号连接时,lstat()会返回该link本身的状态。  详细内容请参考stat()。 |
| 返回值 | 执行成功则返回0,失败返回-1,错误代码存于errno。 |
| 范例 |  |
| 参考stat()。 | |

## fstat(由文件描述符取得文件状态)

|  |  |
| --- | --- |
| **fstat(由文件描述符取得文件状态)** | |
| 相关函数 | stat,lstat,chmod,chown,readlink,utime |
| 表头文件 | #include <sys/stat.h>  #include <unistd.h> |
| 定义函数 | int fstat(int fildes,struct stat \*buf); |
| 函数说明 | fstat()用来将参数fildes所指的文件状态,复制到参数buf所指的结构中(struct stat)。fstat()与stat()作用完全相同,不同处在于传入的参数为已打开的文件描述符。详细内容请参考stat()。 |
| 返回值 | 执行成功则返回0,失败返回-1,错误代码存于errno。 |
| 范例 |  |
| #include <sys/stat.h>  #include <unistd.h>  #include <fcntk.h>  main(){  struct stat buf;  int fd;  fd = open(“/etc/passwd”,O\_RDONLY);  fstat(fd,&buf);  printf(“/etc/passwd file size +%d\n “,buf.st\_size);  } | |
| 执行 |  |
| /etc/passwd file size = 705 | |

## opendir(打开目录)

|  |  |
| --- | --- |
| **opendir(打开目录)** | |
| 相关函数 | open,readdir,closedir,rewinddir,seekdir,telldir,scandir |
| 表头文件 | #include <sys/types.h>  #include <dirent.h> |
| 定义函数 | DIR \*opendir(const char \*name); |
| 函数说明 | opendir()用来打开参数name指定的目录,并返回DIR\*形态的目录流,和open()类似,接下来对目录的读取和搜索都要使用此返回值 |
| 返回值 | 成功则返回DIR\*型态的目录流,打开失败则返回NULL |
| 错误代码 | EACCESS 权限不足  EMFILE 已达到进程可同时打开的文件数上限。  ENFILE 已达到系统可同时打开的文件数上限。  ENOTDIR 参数name非真正的目录  ENOENT 参数name指定的目录不存在,或是参数name为一空字符串。  ENOMEM 核心内存不足。 |

## readdir(读取目录)

|  |  |
| --- | --- |
| **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 |  |
| #include <sys/types.h>  #include <dirent.h>  #include <unistd.h>  main(){  DIR \*dir;  struct dirent \*ptr;  int i;  dir =opendir(“/etc/rc.d”);  while((ptr = readdir(dir))!=NULL){  printf(“d\_name: %s\n”,ptr->d\_name);  }  closedir(dir);  } | |
| 执行 |  |
| d\_name:.  d\_name:..  d\_name:init.d  d\_name:rc0.d  d\_name:rc1.d  d\_name:rc2.d  d\_name:rc3.d  d\_name:rc4.d  d\_name:rc5.d  d\_name:rc6.d  d\_name:rc  d\_name:rc.local  d\_name:rc.sysinit | |

## remove(删除文件)

|  |  |
| --- | --- |
| **remove(删除文件)** | |
| 相关函数 | link,rename,unlink |
| 表头文件 | #include <stdio.h> |
| 定义函数 | int remove(const char \*pathname); |
| 函数说明 | remove()会删除参数pathname指定的文件。如果参数pathname为一文件,则调用unlink()处理,若参数pathname为一目录,则调用rmdir()来处理。请参考unlink()与rmdir()。 |
| 返回值 | 成功则返回0,失败则返回-1,错误原因存于errno |
| 错误代码 | EROFS 欲写入的文件存在于只读文件系统内  EFAULT 参数pathname指针超出可存取内存空间。  ENAMETOOLONG 参数pathname太长  ENOMEM 核心内存不足  ELOOP 参数pathname有过多符号连接问题  EIO I/O存取错误。 |

## rename(更改文件名称或位置)

|  |  |
| --- | --- |
| **rename(更改文件名称或位置)** | |
| 相关函数 | link,unlink,symlink |
| 表头文件 | #include <stdio.h> |
| 定义函数 | int rename(const char \*oldpath,const char \*newpath); |
| 函数说明 | rename()会将参数oldpath所指定的文件名称改为参数newpath所指的文件名称。若newpath所指定的文件已存在,则会被删除。 |
| 返回值 | 执行成功则返回0,失败返回-1,错误原因存于errno |
| 范例 |  |
| /\*设计一个DOS下的rename指令：rename 旧文件名 新文件名\*/  #include <stdio.h>  void main(int argc,char \*\*argv){  if(argc<3){  printf(“Usage: %s old\_name new\_name\n”,argv[0]);  return;  }  printf(“%s=>%s”,argc[1],argv[2]);  if(rename(argv[1],argv[2]<0)  printf(“error!\n”);  else  printf(“ok!\n”);  } | |

## rewinddir(重设读取目录的位置为开头位置)

|  |  |
| --- | --- |
| **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);  printf(“readdir again!\n”);  while((ptr = readdir(dir))!=NULL) {  printf(“d\_name: %s\n”,ptr->d\_name);  }  closedir(dir);  } | |
| 执行 |  |
| d\_name:.  d\_name:..  d\_name:init.d  d\_name:rc0.d  d\_name:rc1.d  d\_name:rc2.d  d\_name:rc3.d  d\_name:rc4.d  d\_name:rc5.d  d\_name:rc6.d  d\_name:rc  d\_name:rc.local  d\_name:rc.sysinit  readdir again!  d\_name:.  d\_name:..  d\_name:init.d  d\_name:rc0.d  d\_name:rc1.d  d\_name:rc2.d  d\_name:rc3.d  d\_name:rc4.d  d\_name:rc5.d  d\_name:rc6.d  d\_name:rc  d\_name:rc.local  d\_name:rc.sysinit | |

## seekdir(设置下回读取目录的位置)

|  |  |
| --- | --- |
| **seekdir(设置下回读取目录的位置)** | |
| 相关函数 | open,opendir,closedir,rewinddir,telldir,readdir,scandir |
| 表头文件 | #include <dirent.h> |
| 定义函数 | void seekdir(DIR \*dir,off\_t offset); |
| 函数说明 | seekdir()用来设置参数dir目录流目前的读取位置,在调用readdir()时便从此新位置开始读取。参数offset代表距离目录文件开头的偏移量。 |
| 返回值 | 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;  printf(“d\_name :%s offset :%d \n”,ptr->d\_name,offset);  }  seekdir(dir offset\_5);  printf(“Readdir again!\n”);  while((ptr = readdir(dir))!=NULL){  offset = telldir(dir);  printf(“d\_name :%s offset :%d\n”,ptr->d\_name.offset);  }  closedir(dir);  } | |
| 执行 |  |
| d\_name : . offset :12  d\_name : .. offset:24  d\_name : init.d offset 40  d\_name : rc0.d offset :56  d\_name :rc1.d offset :72  d\_name:rc2.d offset :88  d\_name:rc3.d offset 104  d\_name:rc4.d offset:120  d\_name:rc5.d offset:136  d\_name:rc6.d offset:152  d\_name:rc offset 164  d\_name:rc.local offset :180  d\_name:rc.sysinit offset :4096  readdir again!  d\_name:rc2.d offset :88  d\_name:rc3.d offset 104  d\_name:rc4.d offset:120  d\_name:rc5.d offset:136  d\_name:rc6.d offset:152  d\_name:rc offset 164  d\_name:rc.local offset :180  d\_name:rc.sysinit offset :4096 | |

## telldir(取得目录流的读取位置)

|  |  |
| --- | --- |
| **telldir(取得目录流的读取位置)** | |
| 相关函数 | open,opendir,closedir,rewinddir,seekdir,readdir,scandir |
| 表头文件 | #include <dirent.h> |
| 定义函数 | off\_t telldir(DIR \*dir); |
| 函数说明 | telldir()返回参数dir目录流目前的读取位置。此返回值代表距离目录文件开头的偏移量 |
| 返回值 | 返回下个读取位置,有错误发生时返回-1。 |
| 错误代码 | EBADF 参数dir为无效的目录流。 |
| 范例 |  |
| #include <sys/types.h>  #include <dirent.h>  #include <unistd.h>  main(){  DIR \*dir;  struct dirent \*ptr;  int offset;  dir = opendir(“/etc/rc.d”);  while((ptr = readdir(dir))!=NULL) {  offset = telldir(dir);  printf(“d\_name : %s offset :%d\n”, ptr->d\_name,offset);  }  closedir(dir);  } | |
| 执行 |  |
| d\_name : . offset :12  d\_name : .. offset:24  d\_name : init.d offset 40  d\_name : rc0.d offset :56  d\_name :rc1.d offset :72  d\_name:rc2.d offset :88  d\_name:rc3.d offset 104  d\_name:rc4.d offset:120  d\_name:rc5.d offset:136  d\_name:rc6.d offset:152  d\_name:rc offset 164  d\_name:rc.local offset :180  d\_name:rc.sysinit offset :4096 | |

## truncate(改变文件大小)

|  |  |
| --- | --- |
| **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。 |

## utime(修改文件的存取时间和更改时间)

|  |  |
| --- | --- |
| **utime(修改文件的存取时间和更改时间)** | |
| 相关函数 | utimes,stat |
| 表头文件 | #include <sys/types.h>  #include <utime.h> |
| 定义函数 | int utime(const char \*filename,struct utimbuf \*buf); |
| 函数说明 | utime()用来修改参数filename文件所属的inode存取时间。结构utimbuf定义如下：  struct utimbuf{  time\_t actime;  time\_t modtime;  };  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)

|  |  |
| --- | --- |
| **utimes(修改文件的存取时间和更改时间)** | |
| 相关函数 | utime,stat |
| 表头文件 | #include <sys/types.h>  #include <utime.h> |
| 定义函数 | int utimes(char \*filename.struct timeval \*tvp); |
| 函数说明 | utimes()用来修改参数filename文件所属的inode存取时间和修改时间,结构timeval定义如下：  struct timeval {  long tv\_sec;  long tv\_usec; /\*微妙\*/  };  参数tvp指向两个timeval结构空间,和utime()使用的utimebuf结构比较,tvp[0].tc\_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 |

## alphasort(依字母顺序排序目录结构)

|  |  |
| --- | --- |
| **alphasort(依字母顺序排序目录结构)** | |
| 相关函数 | scandir,qsort |
| 表头文件 | #include <dirent.h> |
| 定义函数 | int alphasort(const struct dirent \*\*a,const struct dirent \*\*b); |
| 函数说明 | alphasort()为scandir()最后调用qsort()函数时传给qsort()作为判断的函数,详细说明请参考scandir()及qsort()。 |
| 返回值 | 参考qsort()。 |
| 范例 |  |
| /\*读取/目录下所有的目录结构,并依字母顺序排列 \*/  main(){  struct dirent \*\*namelist;  int i,total;  total = scandir(“/”,&namelist ,0,alphasort);  if(total <0)  perror(“scandir”);  else{  for(i=0;i<total;i++)  printf(“%s\n”,namelist[i]->d\_name);  printf(“total = %d\n”,total);  }  } | |
| 执行 |  |
| ..  .gnome  .gnome\_private  ErrorLog  Weblog  bin  boot  dev  dosc  dosd  etc  home  lib  lost+found  misc  mnt  opt  proc  root  sbin  tmp  usr  var  total = 24 | |

# 进程操作

## system(执行shell命令)

|  |  |
| --- | --- |
| **system(执行shell命令)** | |
| 相关函数 | fork,execve,waitpid,popen |
| 表头文件 | #include <stdlib.h> |
| 定义函数 | int system(const char \*string); |
| 函数说明 | system()会调用fork()产生子进程,由子进程来调用/bin/sh-c string来执行参数string字符串所代表的命令,此命令执行完后随即返回原调用的进程。在调用system()期间SIGCHLD信号会被暂时搁置,SIGINT和SIGQUIT信号则会被忽略。 |
| 返回值 | 如果system()在调用/bin/sh时失败则返回127,其他失败原因返回-1。若参数string为空指针(NULL),则返回非零值。如果system()调用成功则最后会返回执行shell命令后的返回值,但是此返回值也有可能为system()调用/bin/sh失败所返回的127,因此最好能再检查errno来确认执行成功。 |
| 附加说明 | 在编写具有SUID/SGID权限的程序时请勿使用system(),system()会继承环境变量,通过环境变量可能会造成系统安全的问题。 |
| 范例 |  |
| #include <stdlib.h>  main(){  system(“ls -al /etc/passwd /etc/shadow”);  } | |
| 执行 |  |
| -rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd  -r--------- 1 root root 572 Sep 2 15 :34 /etc/shadow | |

## execl(执行文件)

|  |  |
| --- | --- |
| **execl(执行文件)** | |
| 相关函数 | fork,execle,execlp,execv,execve,execvp |
| 表头文件 | #include <unistd.h> |
| 定义函数 | int execl(const char \*path,const char \*arg,....); |
| 函数说明 | execl()用来执行参数path字符串所代表的文件路径,接下来的参数代表执行该文件时传递过去的argv〔0〕、argv[1]­……,最后一个参数必须用空指针(NULL)作结束。 |
| 返回值 | 如果执行成功则函数不会返回,执行失败则直接返回-1,失败原因存于errno中。 |
| 范例 |  |
| #include <unistd.h>  main(){  execl(“/bin/ls”,”ls”,”-al”,”/etc/passwd”,(char \*)0);  } | |
| 执行 |  |
| /\*执行/bin/ls -al /etc/passwd \*/  -rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd | |

## execlp(从PATH环境变量中查找文件并执行)

|  |  |
| --- | --- |
| **execlp(从PATH环境变量中查找文件并执行)** | |
| 相关函数 | fork,execl,execle,execv,execve,execvp |
| 表头文件 | #include <unistd.h> |
| 定义函数 | int execlp(const char \*file,const char \*arg,……)； |
| 函数说明 | execlp()会从PATH环境变量所指的目录中查找符合参数file的文件名,找到后便执行该文件,然后将第二个以后的参数当做该文件的argv[0]、argv[1]……,最后一个参数必须用空指针(NULL)作结束。 |
| 返回值 | 如果执行成功则函数不会返回,执行失败则直接返回-1,失败原因存于errno中。 |
| 错误代码 | 参考execve()。 |
| 范例 |  |
| /\*执行 ls -al /etc/passwd execlp()会依PATH变量中的/bin找到/bin/ls \*/  #include <unistd.h>  main(){  execlp(“ls”,”ls”,”-al”,”/etc/passwd”,(char \*)0);  } | |
| 执行 |  |
| -rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd | |

## execv(执行文件)

|  |  |
| --- | --- |
| **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\*) }};  execv(“/bin/ls”,argv);  } | |
| 执行 |  |
| -rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd | |

## execve(执行文件)

|  |  |
| --- | --- |
| **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,执行者并不具有root权限,欲执行的文件具有SUID或SGID位。  2.欲执行的文件所属的文件系统是以nosuid方式挂上,欲执行的文件具有SUID  或SGID位元,但执行者并不具有root权限。  E2BIG 参数数组过大  ENOEXEC 无法判断欲执行文件的执行文件格式,有可能是格式错误或无法在此平台执  行。  EFAULT 参数filename所指的字符串地址超出可存取空间范围。  ENAMETOOLONG 参数filename所指的字符串太长。  ENOENT 参数filename字符串所指定的文件不存在。  ENOMEM 核心内存不足  ENOTDIR 参数filename字符串所包含的目录路径并非有效目录  EACCES 参数filename字符串所包含的目录路径无法存取,权限不足  ELOOP 过多的符号连接  ETXTBUSY 欲执行的文件已被其他进程打开而且正把数据写入该文件中  EIO I/O存取错误  ENFILE 已达到系统所允许的打开文件总数。  EMFILE 已达到系统所允许单一进程所能打开的文件总数。  EINVAL 欲执行文件的ELF执行格式不只一个PT\_INTERP节区  EISDIR ELF翻译器为一目录  ELIBBAD ELF翻译器有问题。 |
| 范例 |  |
| #include <unistd.h>  main(){  char \*argv[ ]={“ls”,”-al”,”/etc/passwd”,(char \*)0};  char \*envp[ ]={“PATH=/bin”,0}  execve(“/bin/ls”,argv,envp);  } | |
| 执行 |  |
| -rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd | |

## execvp(执行文件)

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

## atexit(设置程序正常结束前调用的函数)

|  |  |
| --- | --- |
| **atexit(设置程序正常结束前调用的函数)** | |
| 相关函数 | \_exit,exit,on\_exit |
| 表头文件 | #include <stdlib.h> |
| 定义函数 | int atexit(void(\*function)(void)); |
| 函数说明 | atexit()用来设置一个程序正常结束前调用的函数。当程序通过调用exit()或从main中返回时,参数function所指定的函数会先被调用,然后才真正由exit()结束程序。 |
| 返回值 | 如果执行成功则返回0,否则返回-1,失败原因存于errno中。 |
| 范例 |  |
| #include <stdlib.h>  void my\_exit(void){  printf(“before exit() !\n”);  }  main(){  atexit(my\_exit);  exit(0);  } | |
| 执行 |  |
| before exit()! | |

## exit(正常结束进程)

|  |  |
| --- | --- |
| **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 |  |
| 参考wait() | |

## \_exit(结束进程执行)

|  |  |
| --- | --- |
| **\_exit(结束进程执行)** | |
| 相关函数 | exit,wait,abort |
| 表头文件 | #include <unistd.h> |
| 定义函数 | void \_exit(int status); |
| 函数说明 | \_exit()用来立刻结束目前进程的执行,并把参数status返回给父进程,并关闭未关闭的文件。此函数调用后不会返回,并且会传递SIGCHLD信号给父进程,父进程可以由wait函数取得子进程结束状态。 |
| 返回值 | 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()。 |
|  | |

## vfork(建立一个新的进程)

|  |  |
| --- | --- |
| **vfork(建立一个新的进程)** | |
| 相关函数 | wait,execve |
| 表头文件 | #include <unistd.h> |
| 定义函数 | pid\_t vfork(void); |
| 函数说明 | vfork()会产生一个新的子进程,其子进程会复制父进程的数据与堆栈空间,并继承父进程的用户代码,组代码,环境变量、已打开的文件代码、工作目录和资源限制等。Linux使用copy-on-write(COW)技术,只有当其中一进程试图修改欲复制的空间时才会做真正的复制动作,由于这些继承的信息是复制而来,并非指相同的内存空间,因此子进程对这些变量的修改和父进程并不会同步。此外,子进程不会继承父进程的文件锁定和未处理的信号。注意,Linux不保证子进程会比父进程先执行或晚执行,因此编写程序时要留意死锁或竞争条件的发生。 |
| 返回值 | 如果vfork()成功则在父进程会返回新建立的子进程代码(PID),而在新建立的子进程中则返回0。如果vfork失败则直接返回-1,失败原因存于errno中。 |
| 错误代码 | EAGAIN 内存不足  ENOMEM 内存不足,无法配置核心所需的数据结构空间。 |
| 范例 |  |
| #include <unistd.h>  main(){  if(vfork() = =0){  printf(“This is the child process\n”);  }else{  printf(“This is the parent process\n”);  }  } | |
| 执行 |  |
| this is the parent process  this is the child process | |

## getpgid(取得进程组识别码)

|  |  |
| --- | --- |
| **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>  mian(){  printf(“init gid = %d\n”,getpgid(1));  } | |
| 执行 |  |
| init gid = 0 | |

## getpgrp(取得进程组识别码)

|  |  |
| --- | --- |
| **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());  } | |
| 执行 |  |
| my gid =29546 | |

## getpid(取得进程识别码)

|  |  |
| --- | --- |
| **getpid(取得进程识别码)** | |
| 相关函数 | fork,kill,getpid |
| 表头文件 | #include <unistd.h> |
| 定义函数 | pid\_t getpid(void); |
| 函数说明 | getpid()用来取得目前进程的进程识别码,许多程序利用取到的此值来建立临时文件,以避免临时文件相同带来的问题。 |
| 返回值 | 目前进程的进程识别码 |
| 范例 |  |
| #include <unistd.h>  main(){  printf(“pid=%d\n”,getpid());  } | |
| 执行 |  |
| pid=1494 /\*每次执行结果都不一定相同\*/ | |

## getppid(取得父进程的进程识别码)

|  |  |
| --- | --- |
| **getppid(取得父进程的进程识别码)** | |
| 相关函数 | fork,kill,getpid |
| 表头文件 | #include <unistd.h> |
| 定义函数 | pid\_t getppid(void); |
| 函数说明 | getppid()用来取得目前进程的父进程识别码。 |
| 返回值 | 目前进程的父进程识别码。 |
| 范例 |  |
| #include <unistd.h>  main(){  printf(“My parent ‘pid =%d\n”,getppid());  } | |
| 执行 |  |
| My parent ‘pid =463 | |

## getpriority(取得程序进程执行优先权)

|  |  |
| --- | --- |
| **getpriority(取得程序进程执行优先权)** | |
| 相关函数 | setpriority,nice |
| 表头文件 | #include <sys/time.h>  #include <sys/resource.h> |
| 定义函数 | int getpriority(int which,int who); |
| 函数说明 | getpriority()可用来取得进程、进程组和用户的进程执行优先权。参数which有三种数值,参数who则依which值有不同定义：  which who代表的意义  PRIO\_PROCESS who为进程识别码  PRIO\_PGRP who为进程的组识别码  PRIO\_USER who为用户识别码  此函数返回的数值介于-20至20之间,代表进程执行优先权,数值越低代表有较高的优先次序,执行会较频繁。 |
| 返回值 | 返回进程执行优先权,如有错误发生返回值则为-1且错误原因存于errno。 |
| 附加说明 | 由于返回值有可能是-1,因此要同时检查errno是否存有错误原因。最好在调用次函数前先清除errno变量。 |
| 错误代码 | ESRCH 参数which或who可能有错,而找不到符合的进程  EINVAL 参数which值错误。 |

## nice(改变进程优先顺序)

|  |  |
| --- | --- |
| **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)

|  |  |
| --- | --- |
| **on\_exit(设置程序正常结束前调用的函数)** | |
| 相关函数 | \_exit,atexit,exit |
| 表头文件 | #include <stdlib.h> |
| 定义函数 | int on\_exit(void(\*function)(int, void\*),void \*arg); |
| 函数说明 | on\_exit()用来设置一个程序正常结束前调用的函数。当程序通过调用exit()或从main中返回时,参数function所指定的函数会先被调用,然后才真正由exit()结束程序。参数arg指针会传给参数function函数,详细情况请见范例。 |
| 返回值 | 如果执行成功则返回0,否则返回-1,失败原因存于errno中。 |
| 附加说明 |  |
| 范例 |  |
| #include <stdlib.h>  void my\_exit(int status,void \*arg){  printf(“before exit()!\n”);  printf(“exit(%d)\n”,status);  printf(“arg = %s\n”,(char\*)arg);  }  main(){  char \*str=”test”;  on\_exit(my\_exit,(void \*)str);  exit(1234);  } | |
| 执行 |  |
| before exit()!  exit(1234)  arg = test | |

onexit()、at\_exit()有什么区别？

at\_exit()可以给函数传参数，onexit()不可以传参数。

## setpgid(设置进程组识别码)

|  |  |
| --- | --- |
| **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pgid小于0。  EPERM 进程权限不足,无法完成调用。  ESRCH 找不到符合参数pid指定的进程。 |

## setpgrp(设置进程组识别码)

|  |  |
| --- | --- |
| **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)

|  |  |
| --- | --- |
| **setpriority(设置程序进程执行优先权)** | |
| 相关函数 | getpriority,nice |
| 表头文件 | #include <sys/time.h>  #include <sys/resource.h> |
| 定义函数 | int setpriority(int which,int who, int prio); |
| 函数说明 | setpriority()可用来设置进程、进程组和用户的进程执行优先权。参数which有三种数值,参数who则依which值有不同定义：  which who代表的意义  PRIO\_PROCESS who为进程识别码  PRIO\_PGRP who为进程的组识别码  PRIO\_USER who为用户识别码  参数prio介于-20至20之间。代表进程执行优先权,数值越低代表有较高的优先次序,执行会较频繁。此优先权默认是0,而只有超级用户(root)允许降低此值。 |
| 返回值 | 执行成功则返回0,如果有错误发生返回值则为-1,错误原因存于errno。  ESRCH 参数which或who可能有错,而找不到符合的进程  EINVAL 参数which值错误。  EPERM 权限不够,无法完成设置  EACCES 一般用户无法降低优先权 |

## wait(等待子进程中断或结束)

|  |  |
| --- | --- |
| **wait(等待子进程中断或结束)** | |
| 相关函数 | waitpid,fork |
| 表头文件 | #include <sys/types.h>  #include <sys/wait.h> |
| 定义函数 | pid\_t wait(int \*status); |
| 函数说明 | wait()会暂时停止目前进程的执行,直到有信号来到或子进程结束。如果在调用wait()时子进程已经结束,则wait()会立即返回子进程结束状态值。子进程的结束状态值会由参数status返回,而子进程的进程识别码也会一快返回。如果不在意结束状态值,则参数status可以设成NULL。子进程的结束状态值请参考waitpid()。 |
| 返回值 | 如果执行成功则返回子进程识别码(PID),如果有错误发生则返回-1。失败原因存于errno中。 |
| 附加说明 |  |
| 范例 |  |
| #include <stdlib.h>  #include <unistd.h>  #include <sys/types.h>  #include <sys/wait.h>  main(){  pid\_t pid;  int status,i;  if(fork()= =0){  printf(“This is the child process .pid =%d\n”,getpid());  exit(5);  }else{  sleep(1);  printf(“This is the parent process ,wait for child...\n”;  pid=wait(&status);  i=WEXITSTATUS(status);  printf(“child’s pid =%d .exit status=^d\n”,pid,i);  }  } | |
| 执行 |  |
| This is the child process.pid=1501  This is the parent process .wait for child...  child’s pid =1501,exit status =5 | |

## waitpid(等待子进程中断或结束)

|  |  |
| --- | --- |
| **waitpid(等待子进程中断或结束)** | |
| 相关函数 | wait,fork |
| 表头文件 | #include <sys/types.h>  #include <sys/wait.h> |
| 定义函数 | pid\_t waitpid(pid\_t pid,int \*status,int options); |
| 函数说明 | waitpid()会暂时停止目前进程的执行,直到有信号来到或子进程结束。如果在调用wait()时子进程已经结束,则wait()会立即返回子进程结束状态值。子进程的结束状态值会由参数status返回,而子进程的进程识别码也会一快返回。如果不在意结束状态值,则参数status可以设成NULL。参数pid为欲等待的子进程识别码,其他数值意义如下：  pid<-1 等待进程组识别码为pid绝对值的任何子进程。  pid=-1 等待任何子进程,相当于wait()。  pid=0 等待进程组识别码与目前进程相同的任何子进程。  pid>0 等待任何子进程识别码为pid的子进程。  参数option可以为0或下面的OR组合：  WNOHANG 如果没有任何已经结束的子进程则马上返回,不予以等待。  WUNTRACED 如果子进程进入暂停执行情况则马上返回,但结束状态不予以理会。  子进程的结束状态返回后存于status,底下有几个宏可判别结束情况：  WIFEXITED(status)如果子进程正常结束则为非0值。  WEXITSTATUS(status)取得子进程exit()返回的结束代码,一般会先用WIFEXITED来判断是否正常结束才能使用此宏。  WIFSIGNALED(status)如果子进程是因为信号而结束则此宏值为真  WTERMSIG(status) 取得子进程因信号而中止的信号代码,一般会先用WIFSIGNALED来判断后才使用此宏。  WIFSTOPPED(status) 如果子进程处于暂停执行情况则此宏值为真。一般只有使用WUNTRACED时才会有此情况。  WSTOPSIG(status) 取得引发子进程暂停的信号代码,一般会先用WIFSTOPPED来判断后才使用此宏。 |
| 返回值 | 如果执行成功则返回子进程识别码(PID),如果有错误发生则返回-1。失败原因存于errno中。 |
| 范例 |  |
| 参考wait()。 | |

# 进程间通信&socket编程

## pipe(建立管道)

|  |  |
| --- | --- |
| **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,并将文件描述符由参数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符最大量  ENFILE 系统已无文件描述符可用。  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));  } else { /\*子进程\*/  read(filedes[0],buffer,80);  printf(“%s”,buffer);  }  } | |
| 执行 |  |
| hello! | |

## popen(建立管道I/O)

|  |  |
| --- | --- |
| **popen(建立管道I/O)** | |
| 相关函数 | pipe,mkfifo,pclose,fork,system,fopen |
| 表头文件 | #include <stdio.h> |
| 定义函数 | FILE\*popen( const char \*command,const char \*type); |
| 函数说明 | popen()会调用fork()产生子进程,然后从子进程中调用/bin/sh -c来执行参数command的指令。参数type可使用“r”代表读取,“w”代表写入。依照此type值,popen()会建立管道连到子进程的标准输出设备或标准输入设备,然后返回一个文件指针。随后进程便可利用此文件指针来读取子进程的输出设备或是写入到子进程的标准输入设备中。此外,所有使用文件指针(FILE\*)操作的函数也都可以使用,除了fclose()以外。 |
| 返回值 | 若成功则返回文件指针,否则返回NULL,错误原因存于errno中。 |
| 错误代码 | EINVAL 参数type不合法。 |
| 注意事项 | 在编写具SUID/SGID权限的程序时请尽量避免使用popen(),popen()会继承环境变量,通过环境变量可能会造成系统安全的问题。 |
| 范例 |  |
| #include <stdio.h>  main(){  FILE \*fp;  char buffer[80];  fp=popen(“cat /etc/passwd”,”r”);  fgets(buffer,sizeof(buffer),fp);  printf(“%s”,buffer);  pclose(fp);  } | |
| 执行 |  |
| root :x:0 0: root: /root: /bin/bash | |

## pclose(关闭管道I/O)

|  |  |
| --- | --- |
| **pclose(关闭管道I/O)** | |
| 相关函数 | popen |
| 表头文件 | #include <stdio.h> |
| 定义函数 | int pclose(FILE \*stream); |
| 函数说明 | pclose()用来关闭由popen所建立的管道及文件指针。参数stream为先前由popen()所返回的文件指针。 |
| 返回值 | 返回子进程的结束状态。如果有错误则返回-1,错误原因存于errno中。 |
| 错误代码 | ECHILD pclose()无法取得子进程的结束状态。 |
| 范例 |  |
| 参考popen()。 | |

## mkfifo(建立具名管道)

|  |  |
| --- | --- |
| **mkfifo(建立具名管道)** | |
| 相关函数 | pipe,popen,open,umask |
| 表头文件 | #include <sys/types.h>  #include <sys/stat.h> |
| 定义函数 | int mkfifo(const char \*pathname,mode\_t mode); |
| 函数说明 | mkfifo()会依参数pathname建立特殊的FIFO文件,该文件必须不存在,而参数mode为该文件的权限(mode%~umask),因此umask值也会影响到FIFO文件的权限。mkfifo()建立的FIFO文件其他进程都可以用读写一般文件的方式存取。当使用open()来打开FIFO文件时,O\_NONBLOCK旗标会有影响：  1.当使用O\_NONBLOCK旗标时,打开FIFO文件来读取的操作会立刻返回,但是若还没有其他进程打开FIFO文件来读取,则写入的操作会返回ENXIO错误代码。  2.没有使用O\_NONBLOCK旗标时,打开FIFO来读取的操作会等到其他进程打开FIFO文件来写入才正常返回。同样地,打开FIFO文件来写入的操作会等到其他进程打开FIFO文件来读取后才正常返回。 |
| 返回值 | 若成功则返回0,否则返回-1,错误原因存于errno中。 |
| 错误代码 | EACCESS 参数pathname所指定的目录路径无可执行的权限  EEXIST 参数pathname所指定的文件已存在。  ENAMETOOLONG 参数pathname的路径名称太长。  ENOENT 参数pathname包含的目录不存在  ENOSPC 文件系统的剩余空间不足  ENOTDIR 参数pathname路径中的目录存在但却非真正的目录。  EROFS 参数pathname指定的文件存在于只读文件系统内。 |
| 范例 |  |
| #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  main(){  char buffer[80];  int fd;  unlink(FIFO);  mkfifo(FIFO,0666);  if(fork()>0){  char s[ ] = “hello!\n”;  fd = open(FIFO,O\_WRONLY);  write(fd,s,sizeof(s));  close(fd);  } else {  fd= open(FIFO,O\_RDONLY);  read(fd,buffer,80);  printf(“%s”,buffer);  close(fd);  }  } | |
| 执行 |  |
| hello! | |

## socket(建立一个socket通信)

|  |  |
| --- | --- |
| **socket(建立一个socket通信)** | |
| 相关函数 | accept,bind,connect,listen |
| 表头文件 | #include <sys/types.h>  #include <sys/socket.h> |
| 定义函数 | int socket(int domain,int type,int protocol); |
| 函数说明 | socket()用来建立一个新的socket,也就是向系统注册,通知系统建立一通信端口。参数domain指定使用何种的地址类型,完整的定义在/usr/include/bits/socket.h内,底下是常见的协议：  PF\_UNIX/PF\_LOCAL/AF\_UNIX/AF\_LOCAL UNIX进程通信协议  PF\_INET/AF\_INET Ipv4网络协议  PF\_INET6/AF\_INET6 Ipv6网络协议  PF\_IPX/AF\_IPX IPX-Novell协议  PF\_NETLINK/AF\_NETLINK 核心用户接口装置  PF\_X25/AF\_X25 ITU-T X.25/ISO-8208协议  PF\_AX25/AF\_AX25 业余无线AX.25协议  PF\_ATMPVC/AF\_ATMPVC 存取原始ATM PVCs  PF\_APPLETALK/AF\_APPLETALK appletalk(DDP)协议  PF\_PACKET/AF\_PACKET 初级封包接口  参数type有下列几种数值：  SOCK\_STREAM 提供双向连续且可信赖的数据流,即TCP。支持OOB机制,在所有数据传送前必须使用connect()来建立连线状态。  SOCK\_DGRAM 使用不连续不可信赖的数据包连接,UDP  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指定的类型不支持参数type或protocol指定的协议  ENFILE 核心内存不足,无法建立新的socket结构  EMFILE 进程文件表溢出,无法再建立新的socket  EACCESS 权限不足,无法建立type或protocol指定的协议  ENOBUFS/ENOMEM 内存不足  EINVAL 参数domain/type/protocol不合法 |
| 范例 |  |
| 参考connect()。 | |

## bind(对socket定位)

|  |  |
| --- | --- |
| **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**绑定本地地址和端口号**。  bind()用来设置给参数sockfd的socket一个名称。此名称由参数my\_addr指向一sockaddr结构,对于不同的socket domain定义了一个通用的数据结构：  struct sockaddr{  unsigned short int sa\_family;  char sa\_data[14];  };  sa\_family 为调用socket()时的domain参数,即AF\_xxxx值。  sa\_data 最多使用14个字符长度。  此sockaddr结构会因使用不同的socket domain而有不同结构定义,例如使用AF\_INET domain,其socketaddr结构定义便为：  struct socketaddr\_in{  unsigned short int sin\_family;  uint16\_t sin\_port;  struct in\_addr sin\_addr;  unsigned char sin\_zero[8];  };  struct in\_addr{  uint32\_t s\_addr;  };  sin\_family 即为sa\_family  sin\_port 为使用的port编号  sin\_addr.s\_addr 为IP地址  sin\_zero 未使用。  参数addrlen为sockaddr的结构长度。 |
| 返回值 | 成功则返回0,失败返回-1,错误原因存于errno中。 |
| 错误代码 | EBADF 参数sockfd非合法socket处理代码。  EACCESS 权限不足  ENOTSOCK 参数sockfd为一文件描述符,非socket。 |
| 范例 |  |
| 参考listen() | |

## listen(等待连接)

|  |  |
| --- | --- |
| **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。listen()并未开始接收连线,只是设置socket为listen模式,真正接收client端连线的是accept()。通常listen()会在socket()、 bind()之后调用,接着才调用accept()。 |
| 返回值 | 成功则返回0,失败返回-1,错误原因存于errno |
| 附加说明 | listen()只适用SOCK\_STREAM或SOCK\_SEQPACKET的socket类型。如果socket为AF\_INET则参数backlog最大值可设至128。 |
| 错误代码 | EBADF 参数sockfd非合法socket处理代码  EACCESS 权限不足  EOPNOTSUPP 指定的socket并未支援listen模式。 |
| 范例 |  |
| #include <sys/types.h>  #include <sys/socket.h>  #include <netinet/in.h>  #include <arpa/inet.h>  #include <unistd.h>  #define PORT 1234  #define MAXSOCKFD 10  main(){  int sockfd,newsockfd,is\_connected[MAXSOCKFD],fd;  struct sockaddr\_in addr;  int addr\_len = sizeof(struct sockaddr\_in);  fd\_set readfds;  char buffer[256];  char msg[ ] =”Welcome to server!”;  if((sockfd = socket(AF\_INET,SOCK\_STREAM,0))<0){  perror(“socket”);  exit(1);  }  bzero(&addr,sizeof(addr));  addr.sin\_family =AF\_INET;  addr.sin\_port = htons(PORT);  addr.sin\_addr.s\_addr = htonl(INADDR\_ANY);  if(bind(sockfd,&addr,sizeof(addr))<0){  perror(“connect”);  exit(1);  }  if(listen(sockfd,3)<0){  perror(“listen”);  exit(1);  }  for(fd=0;fd<MAXSOCKFD;fd++)  is\_connected[fd]=0;  while(1){  FD\_ZERO(&readfds);  FD\_SET(sockfd,&readfds);  for(fd=0;fd<MAXSOCKFD;fd++)  if(is\_connected[fd]) FD\_SET(fd,&readfds);  if(!select(MAXSOCKFD,&readfds,NULL,NULL,NULL))continue;  for(fd=0;fd<MAXSOCKFD;fd++)  if(FD\_ISSET(fd,&readfds)){  if(sockfd = =fd){  if((newsockfd = accept(sockfd,&addr,&addr\_len))<0)  perror(“accept”);  write(newsockfd,msg,sizeof(msg));  is\_connected[newsockfd] =1;  printf(“cnnect from %s\n”,inet\_ntoa(addr.sin\_addr));  }else{  bzero(buffer,sizeof(buffer));  if(read(fd,buffer,sizeof(buffer))<=0){  printf(“connect closed.\n”);  is\_connected[fd]=0;  close(fd);  }else  printf(“%s”,buffer);  }  }  }  } | |
| 执行 |  |
| $ ./listen  connect from 127.0.0.1  hi I am client  connected closed. | |

## accept(接受socket连线)

|  |  |
| --- | --- |
| **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scokaddr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为一文件描述符,非socket。  EOPNOTSUPP 指定的socket并非SOCK\_STREAM。  EPERM 防火墙拒绝此连线  ENOBUFS 系统的缓冲内存不足  ENOMEM 核心内存不足。 |
| 范例 |  |
| 参考listen()。 | |

## connect(建立socket连线)

|  |  |
| --- | --- |
| **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为一文件描述符,非socket。  EISCONN 参数sockfd的socket已是连线状态  ECONNREFUSED 连线要求被server端拒绝。  ETIMEDOUT 企图连线的操作超过限定时间仍未有响应。  ENETUNREACH 无法传送数据包至指定的主机。  EAFNOSUPPORT sockaddr结构的sa\_family不正确。  EALREADY socket为不可阻断且先前的连线操作还未完成。 |
| 范例 |  |
| /\*利用socket的TCP client  此程序会连线TCP server,并将键盘输入的字符串传送给server。  TCP server范例请参考listen()。  \*/  #include <sys/stat.h>  #include <fcntl.h>  #include <unistd.h>  #include <sys/types.h>  #include <sys/socket.h>  #include <netinet/in.h>  #include <arpa/inet.h>  #define PORT 1234  #define SERVER\_IP “127.0.0.1”  main(){  int s;  struct sockaddr\_in addr;  char buffer[256];  if((s = socket(AF\_INET,SOCK\_STREAM,0))<0){  perror(“socket”);  exit(1);  }  /\*填写sockaddr\_in结构\*/  bzero(&addr,sizeof(addr));  addr.sin\_family = AF\_INET;  addr.sin\_port=htons(PORT);  addr.sin\_addr.s\_addr = inet\_addr(SERVER\_IP);  /\*尝试连线\*/  if(connect(s,&addr,sizeof(addr))<0){  perror(“connect”);  exit(1);  }  /\*接收由server端传来的信息\*/  recv(s,buffer,sizeof(buffer),0);  printf(“%s\n”,buffer);  while(1){  bzero(buffer,sizeof(buffer));  read(STDIN\_FILENO,buffer,sizeof(buffer)); //从标准输入设备取得字符串  if(send(s,buffer,sizeof(buffer),0)<0){ //将字符串传给server端  perror(“send”);  exit(1);  }  }  } | |
| 执行 |  |
| $ ./connect  Welcome to server!  hi I am client! /\*键盘输入\*/  /\*<Ctrl+C>中断程序\*/ | |

## send(经socket传送数据)

|  |  |
| --- | --- |
| **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 falgs); |
| 函数说明 | send()用来将数据由指定的socket传给对方主机。参数s为已建立好连接的socket。参数msg指向欲连线的数据内容,参数len则为数据长度。参数flags一般设0,其他数值定义如下：  MSG\_OOB 传送的数据以out-of-band送出。  MSG\_DONTROUTE 取消路由表查询  MSG\_DONTWAIT 设置为不可阻断运作  MSG\_NOSIGNAL 此动作不愿被SIGPIPE信号中断。 |
| 返回值 | 成功则返回实际传送出去的字符数,失败返回-1。错误原因存于errno |
| 错误代码 | EBADF 参数s非合法的socket处理代码。  EFAULT 参数中有一指针指向无法存取的内存空间  ENOTSOCK 参数s为一文件描述符,非socket。  EINTR 被信号所中断。  EAGAIN 此操作会令进程阻断,但参数s的socket为不可阻断。  ENOBUFS 系统的缓冲内存不足  ENOMEM 核心内存不足  EINVAL 传给系统调用的参数不正确。 |
| 范例 |  |
| 参考connect() | |

## recv(经socket接收数据)

|  |  |
| --- | --- |
| **recv(经socket接收数据)** | |
| 相关函数 | recvfrom,recvmsg,send,sendto,socket |
| 表头文件 | #include <sys/types.h>  #include <sys/socket.h> |
| 定义函数 | int recv(int s,void \*buf,int len,unsigned int flags); |
| 函数说明 | recv()用来接收远端主机经指定的socket传来的数据,并把数据存到由参数buf指向的内存空间,参数len为可接收数据的最大长度。  参数flags一般设0。其他数值定义如下：  MSG\_OOB 接收以out-of-band送出的数据。  MSG\_PEEK 返回来的数据并不会在系统内删除,如果再调用recv()会返回相同的数据内容。  MSG\_WAITALL 强迫接收到len大小的数据后才能返回,除非有错误或信号产生。  MSG\_NOSIGNAL 此操作不愿被SIGPIPE信号中断 |
| 返回值 | 成功则返回接收到的字符数,失败返回-1,错误原因存于errno中。 |
| 错误代码 | EBADF 参数s非合法的socket处理代码  EFAULT 参数中有一指针指向无法存取的内存空间  ENOTSOCK 参数s为一文件描述符,非socket。  EINTR 被信号所中断  EAGAIN 此动作会令进程阻断,但参数s的socket为不可阻断  ENOBUFS 系统的缓冲内存不足。  ENOMEM 核心内存不足  EINVAL 传给系统调用的参数不正确。 |
| 范例 |  |
| 参考listen()。 | |

## sendto(经socket传送数据)

|  |  |
| --- | --- |
| **sendto(经socket传送数据)** | |
| 相关函数 | send, sendmsg,recv , recvfrom , socket |
| 表头文件 | #include < sys/types.h >  #include < sys/socket.h > |
| 定义函数 | int sendto(int s, const void \*msg, int len, unsigned int flags, const struct sockaddr \*to , int tolen ) ; |
| 函数说明 | sendto()用来将数据由指定的socket传给对方主机。参数s为已建好连线的socket,如果利用UDP协议则不需经过连线connect操作。参数msg指向欲连线的数据内容,参数flags一般设0,详细描述请参考send()。参数to用来指定欲传送的网络地址,结构sockaddr请参考bind().参数tolen为sockaddr的结果长度。 |
| 返回值 | 成功则返回实际传送出去的字符数,失败返回－1,错误原因存于errno 中。 |
| 错误代码 | EBADF 参数s非法的socket处理代码。  EFAULT 参数中有一指针指向无法存取的内存空间。  WNOTSOCK canshu s为一文件描述符,非socket。  EINTR 被信号所中断。  EAGAIN 此动作会令进程阻断,但参数s的soket为补课阻断的。  ENOBUFS 系统的缓冲内存不足。  EINVAL 传给系统调用的参数不正确。 |
| 范例 |  |
| #include < sys/types.h >  #include < sys/socket.h >  # include <netinet.in.h>  #include <arpa.inet.h>  #define PORT 2345 /\*使用的port\*/  main(){  int sockfd,len;  struct sockaddr\_in addr;  char buffer[256];    /\*建立socket\*/  if(sockfd=socket(AF\_INET,SOCK\_DGRAM,0))<0){  perror(“socket”);  exit(1);  }  /\*填写sockaddr\_in 结构\*/  bzero( &addr, sizeof(addr) );  addr.sin\_family=AF\_INET;  addr.sin\_port=htons(PORT);  addr.sin\_addr=hton1(INADDR\_ANY) ;  if(bind(sockfd, &addr, sizeof(addr))<0){  perror(“connect”);  exit(1);  }  while(1){  bezro(buffer,sizeof(buffer));  len = recvfrom(socket,buffer,sizeof(buffer), 0 , &addr, &addr\_len);  /\*显示client端的网络地址\*/  printf(“receive from %s\n “ , inet\_ntoa( addr.sin\_addr));  /\*将字串返回给client端\*/  sendto(sockfd,buffer,len,0,&addr,addr\_len);”  }  } | |
| 执行 |  |
| 请参考 recvfrom() | |

## recvfrom(经socket接收数据)

|  |  |
| --- | --- |
| **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 | recvfrom()用来接收远程主机经指定的socket传来的数据,并把数据存到由参数buf指向的内存空间,参数len为可接收数据的最大长度。参数flags一般设0,其他数值定义请参考recv()。参数from用来指定欲传送的网络地址,结构sockaddr请参考bind()。参数fromlen为sockaddr的结构长度。 |
| 返回值 | 成功则返回接收到的字符数,失败则返回-1,错误原因存于errno中。 |
| 错误代码 | EBADF 参数s非合法的socket处理代码  EFAULT 参数中有一指针指向无法存取的内存空间。  ENOTSOCK 参数s为一文件描述符,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给server。  UDP server 范例请参考sendto()。  \*/  #include <sys/stat.h>  #include <fcntl.h>  #include <unistd.h>  #include <sys/typs.h>  #include <sys/socket.h>  #include <netinet/in.h>  #include <arpa/inet.h>  #define PORT 2345  #define SERVER\_IP “127.0.0.1”  main() {  int s,len;  struct sockaddr\_in addr;  int addr\_len =sizeof(struct sockaddr\_in);  char buffer[256];  /\*建立socket\*/  if((s = socket(AF\_INET,SOCK\_DGRAM,0))<0){  perror(“socket”);  exit(1);  }  /\*填写sockaddr\_in\*/  bzero(&addr,sizeof(addr));  addr.sin\_family = AF\_INET;  addr.sin\_port = htons(PORT);  addr.sin\_addr.s\_addr = inet\_addr(SERVER\_IP);  while(1){  bzero(buffer,sizeof(buffer));  /\*从标准输入设备取得字符串\*/  len =read(STDIN\_FILENO,buffer,sizeof(buffer));  /\*将字符串传送给server端\*/  sendto(s,buffer,len,0,&addr,addr\_len);  /\*接收server端返回的字符串\*/  len = recvfrom(s,buffer,sizeof(buffer),0,&addr,&addr\_len);  printf(“receive: %s”,buffer);  }  } | |
| 执行 |  |
| (先执行udp server再执行udp client)  hello /\*从键盘输入字符串\*/  receive: hello /\*server端返回来的字符串\*/ | |

## sendmsg(经socket传送数据)

|  |  |
| --- | --- |
| **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 strcut 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 addres data \*/  strcut iovec \*msg\_iov; /\*Vector of data to send/receive into \*/  size\_t msg\_iovlen; /\*Number of elements in the vector \*/  void \*msg\_control; /\*Ancillary dat \*/  size\_t msg\_controllen; /\*Ancillary data buffer length \*/  int msg\_flags; /\*Flags on received message \*/  }; |
| 返回值 | 成功则返回实际传送出去的字符数,失败返回-1,错误原因存于errno |
| 错误代码 | EBADF 参数s非合法的socket处理代码。  EFAULT 参数中有一指针指向无法存取的内存空间  ENOTSOCK 参数s为一文件描述符,非socket。  EINTR 被信号所中断。  EAGAIN 此操作会令进程阻断,但参数s的socket为不可阻断。  ENOBUFS 系统的缓冲内存不足  ENOMEM 核心内存不足  EINVAL 传给系统调用的参数不正确。 |
| 范例 |  |
| 参考sendto()。 | |

## recvmsg(经socket接收数据)

|  |  |
| --- | --- |
| **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/socktet.h> |
| 定义函数 | int recvmsg(int s,struct msghdr \*msg,unsigned int flags); |
| 函数说明 | recvmsg()用来接收远程主机经指定的socket传来的数据。参数s为已建立好连线的socket,如果利用UDP协议则不需经过连线操作。参数msg指向欲连线的数据结构内容,参数flags一般设0,详细描述请参考send()。关于结构msghdr的定义请参考sendmsg()。 |
| 返回值 | 成功则返回接收到的字符数,失败则返回-1,错误原因存于errno中。 |
| 错误代码 | EBADF 参数s非合法的socket处理代码。  EFAULT 参数中有一指针指向无法存取的内存空间  ENOTSOCK 参数s为一文件描述符,非socket。  EINTR 被信号所中断。  EAGAIN 此操作会令进程阻断,但参数s的socket为不可阻断。  ENOBUFS 系统的缓冲内存不足  ENOMEM 核心内存不足  EINVAL 传给系统调用的参数不正确。 |
| 范例 |  |
| 参考recvfrom()。 | |

## setsockopt(设置socket状态)

|  |  |
| --- | --- |
| **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形态。  SO\_ERROR 返回socket已发生的错误原因  SO\_DONTROUTE 送出的数据包不要利用路由设备来传输。  SO\_BROADCAST 使用广播方式传送  SO\_SNDBUF 设置送出的暂存区大小  SO\_RCVBUF 设置接收的暂存区大小  SO\_KEEPALIVE 定期确定连线是否已终止。  SO\_OOBINLINE 当接收到OOB数据时会马上送至标准输入设备  SO\_LINGER 确保数据安全且可靠的传送出去。  参数optval代表欲设置的值,参数optlen则为optval的长度。 |
| 返回值 | 成功则返回0,若有错误则返回-1,错误原因存于errno。 |
| 附加说明 | EBADF 参数s并非合法的socket处理代码  ENOTSOCK 参数s为一文件描述符,非socket  ENOPROTOOPT 参数optname指定的选项不正确。  EFAULT 参数optval指针指向无法存取的内存空间。 |
| 范例 |  |
| 参考getsockopt()。 | |

## getsockopt(取得socket状态)

|  |  |
| --- | --- |
| **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为一文件描述符,非socket  ENOPROTOOPT 参数optname指定的选项不正确  EFAULT 参数optval指针指向无法存取的内存空间 |
| 范例 |  |
| #include <sys/types.h>  #include <sys/socket.h>  main()  {  int s,optval,optlen = sizeof(int);  if((s = socket(AF\_INET,SOCK\_STREAM,0))<0) perror(“socket”);  getsockopt(s,SOL\_SOCKET,SO\_TYPE,&optval,&optlen);  printf(“optval = %d\n”,optval);  close(s); } | |
| 执行 |  |
| optval = 1 /\*SOCK\_STREAM的定义正是此值\*/ | |

## setprotoent(打开网络协议的数据文件)

|  |  |
| --- | --- |
| **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。 |

## endprotoent(结束网络协议数据的读取)

|  |  |
| --- | --- |
| **endprotoent(结束网络协议数据的读取)** | |
| 相关函数 | getprotoent,getprotobyname,getprotobynumber,setprotoent |
| 表头文件 | #include <netdb.h> |
| 定义函数 | void endprotoent(void); |
| 函数说明 | endprotoent()用来关闭由getprotoent()打开的文件。 |
| 返回值 | 无 |
| 范例 |  |
| 参考getprotoent() | |

## setservent(打开主机网络服务的数据文件)

|  |  |
| --- | --- |
| **setservent(打开主机网络服务的数据文件)** | |
| 相关函数 | getservent, getservbyname, getservbyport, endservent |
| 表头文件 | #include < netdb.h > |
| 定义函数 | void setservent(int stayopen); |
| 函数说明 | setservent() 用来打开/etc/services,如果参数stayopen值为1,则接下来的getservbyname()或getservbyport()将补回自动关闭文件。 |

## endservent(结束网络服务数据的读取)

|  |  |
| --- | --- |
| **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 |  |
| 参考getservent()。 | |

## shutdown(终止socket通信)

|  |  |
| --- | --- |
| **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为一文件描述符,非socket  ENOTCONN 参数s指定的socket并未连线 |

## htonl(将32位主机字符顺序转换成网络字符顺序)

|  |  |
| --- | --- |
| **htonl(将32位主机字符顺序转换成网络字符顺序)** | |
| 相关函数 | htons,ntohl,ntohs |
| 表头文件 | #include <netinet/in.h> |
| 定义函数 | unsigned long int htonl(unsigned long int hostlong); |
| 函数说明 | htonl()用来将参数指定的32位hostlong转换成网络字符顺序。 |
| 返回值 | 返回对应的网络字符顺序。 |
| 范例 |  |
| 参考getservbyport()或connect()。 | |

## htons(将16位主机字符顺序转换成网络字符顺序)

|  |  |
| --- | --- |
| **htons(将16位主机字符顺序转换成网络字符顺序)** | |
| 相关函数 | htonl,ntohl,ntohs |
| 表头文件 | #include <netinet/in.h> |
| 定义函数 | unsigned short int htons(unsigned short int hostshort); |
| 函数说明 | htons()用来将参数指定的16位hostshort转换成网络字符顺序。 |
| 返回值 | 返回对应的网络字符顺序。 |
| 范例 |  |
| 参考connect()。 | |

## inet\_addr(将网络地址转成二进制的数字)

|  |  |
| --- | --- |
| **inet\_addr(将网络地址转成二进制的数字)** | |
| 相关函数 | inet\_aton,inet\_ntoa |
| 表头文件 | #include <sys/socket.h>  #include <netinet/in.h>  #include <arpa/inet.h> |
| 定义函数 | unsigned long int inet\_addr(const char \*cp); |
| 函数说明 | inet\_addr()用来将参数cp所指的网络地址字符串转换成网络所使用的二进制数字。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)

|  |  |
| --- | --- |
| **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/scoket.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使用的二进制的数字,然后存于参数inp所指的in\_addr结构中。  结构in\_addr定义如下：  struct in\_addr{  unsigned long int s\_addr;  }; |
| 返回值 | 成功则返回非0值,失败则返回0。 |

## inet\_ntoa(将网络二进制的数字转换成网络地址)

|  |  |
| --- | --- |
| **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。 |
| 返回值 | 成功则返回字符串指针,失败则返回NULL。 |

### inet\_addr和inet\_aton区别

两个函数都把点分十进制数转换为网络字节序的IPV4地址，

int inet\_aton(const char \*strptr,struct in\_addr \*addptr)；

in\_addr\_t inet\_addr(const char \*strptr)；

第一个函数将strptr指向的c字符串转换为32位的网络字节序二进制值，存储在addrptr中，如果执行成功返回1，否则返回0.如果addrptr指针为空，函数仍能对输入的字符串做有效性检查，但是不存储任何结果。

第二个函数进行相同转换，但是有一个历史遗留问题，所有的2的32次方个可能的二进制值都是有效的IP地址，，但是该函数出错时返回值为INADDR\_NONE（通常是个32位均为1的值），这意味着点分十进制数串255.255.255.255不能由该函数处理，因为他的二进制值用来表示函数失败。

inet\_aton,inet\_addr和inet\_ntoa在点分十进制数串（如，“192.168.1.10")与他的32位网络字节二进制值之前转换IPV4地址，有2个比较新的函数inet\_pton和inet\_ntop，这2个对IPV4和IPV6地址都能处理

int inet\_aton(const char \*cp, struct in\_addr \*inp);

in\_addr\_t inet\_addr(const char \*cp);

char \*inet\_ntoa(struct in\_addr in);

inet\_aton() converts the Internet host address cp from the standard numbers-and-dots notation into binary data and stores it in the structure that inp points to. inet\_aton() returns non-zero if the address is valid, zero if not.

inet\_aton() 转换网络主机地址cp为二进制数值，并存储在struct in\_addr结构中，即第二个参数\*inp,函数返回非0表示cp主机有地有效，返回0表示主机地址无效。

The inet\_addr() function converts the Internet host address cp from numbers-and-dots notation into binary data in network byte order. If the input is invalid, INADDR\_NONE (usually -1) is returned. This is an

obsolete interface to inet\_aton(), described immediately above; it is obsolete because -1 is a valid address (255.255.255.255), and inet\_aton() provides a cleaner way to indicate error return.

inet\_addr函数转换网络主机地址（如192.168.1.10)为网络字节序二进制值，如果参数char \*cp无效，函数返回-1(INADDR\_NONE),这个函数在处理地址为255.255.255.255时也返回 －1,255.255.255.255是一个有效的地址，不过inet\_addr无法处理;

The inet\_ntoa() function converts the Internet host address in given in network byte order to a string in standard numbers-and-dots notation. The string is returned in a statically allocated buffer, which subsequent calls will overwrite.

inet\_ntoa 函数转换网络字节排序的地址为标准的ASCII以点分开的地址,,该函数返回指向点分开的字符串地址的指针，该字符串的空间为静态分配的，这意味着在第二次调用该函数时，上一次调用将会被重写（覆盖），所以如果需要保存该串最后复制出来自己管理！

现在一般使用inet\_aton和inet\_ntoa来处理网络字节和主机字节之间的转换;

有两个更新的函数inet\_pton和inet\_ntop这2个函数能够处理ipv4和ipv6,原型如下

**int inet\_pton(int af, const char \*src, void \*dst);**

这个函数转换字符串到网络地址，第一个参数af是地址族，转换后存在dst中

inet\_pton 是inet\_addr的扩展，支持的多地址族有下列：

AF\_INET

src为指向字符型的地址，即ASCII的地址的首地址（ddd.ddd.ddd.ddd格式的），函数将该地址

转换为in\_addr的结构体，并复制在\*dst中

AF\_INET6

src为指向IPV6的地址，，函数将该地址

转换为in6\_addr的结构体，并复制在\*dst中

如果函数出错将返回一个负值，并将errno设置为EAFNOSUPPORT，如果参数af指定的地址族和src格式不对，函数将返回0。

函数inet\_ntop进行相反的转换原型如下

**const char \*inet\_ntop(int af, const void \*src, char \*dst, socklen\_t cnt);**

这个函数转换网络二进制结构到ASCII类型的地址，参数的作用和上面相同，只是多了一个参数socklen\_t cnt,他是所指向缓存区dst的大小，避免溢出，如果缓存区太小无法存储地址的值，则返回一个空指针，并将errno置为ENOSPC

例子：

int main(void){

char\* ip = "192.168.1.87";

struct in\_addr inp;

u\_int32\_t addr = 0x5701a8c0;

inet\_aton(ip, &inp);

printf("%x ", inp);

inp.s\_addr = addr;

printf("%s \n", inet\_ntoa(inp));

return 0;

}

## ntohl(将32位网络字符顺序转换成主机字符顺序)

|  |  |
| --- | --- |
| **ntohl(将32位网络字符顺序转换成主机字符顺序)** | |
| 相关函数 | htonl,htons,ntohs |
| 表头文件 | #include <netinet/in.h> |
| 定义函数 | unsigned long int ntohl(unsigned long int netlong); |
| 函数说明 | ntohl()用来将参数指定的32位netlong转换成主机字符顺序。 |
| 返回值 | 返回对应的主机字符顺序。 |
| 范例 |  |
| 参考getservent()。 | |

## ntohs(将16位网络字符顺序转换成主机字符顺序)

|  |  |
| --- | --- |
| **ntohs(将16位网络字符顺序转换成主机字符顺序)** | |
| 相关函数 | htonl,htons,ntohl |
| 表头文件 | #include <netinet/in.h> |
| 定义函数 | unsigned short int ntohs(unsigned short int netshort); |
| 函数说明 | ntohs()用来将参数指定的16位netshort转换成主机字符顺序。 |
| 返回值 | 返回对应的主机顺序。 |
| 范例 |  |
| 参考getservent()。 | |

# 信号处理

## alarm(设置信号传送闹钟)

|  |  |
| --- | --- |
| **alarm(设置信号传送闹钟)** | |
| 相关函数 | signal,sleep |
| 表头文件 | #include <unistd.h> |
| 定义函数 | unsigned int alarm(unsigned int seconds); |
| 函数说明 | alarm()用来设置信号SIGALRM在经过参数seconds指定的秒数后传送给目前的进程。如果参数seconds为0,则之前设置的闹钟会被取消,并将剩下的时间返回。 |
| 返回值 | 返回之前闹钟的剩余秒数,如果之前未设闹钟则返回0。 |
| 范例 |  |
| #include <unistd.h>  #include <signal.h>  void handler() {  printf(“hello\n”);  }  main() {  int i;  signal(SIGALRM,handler);  alarm(5);  for(i=1;i<7;i++){  printf(“sleep %d ...\n”,i);  sleep(1);  }  } | |
| 执行 |  |
| sleep 1 ...  sleep 2 ...  sleep 3 ...  sleep 4 ...  sleep 5 ...  hello  sleep 6 ... | |

## kill(传送信号给指定的进程)

|  |  |
| --- | --- |
| **kill(传送信号给指定的进程)** | |
| 相关函数 | raise,signal |
| 表头文件 | #include <sys/types.h>  #include <signal.h> |
| 定义函数 | int kill(pid\_t pid,int sig); |
| 函数说明 | kill()可以用来送参数sig指定的信号给参数pid指定的进程。参数pid有几种情况：  pid>0 将信号传给进程识别码为pid的进程。  pid=0 将信号传给和目前进程相同进程组的所有进程  pid=-1 将信号广播传送给系统内所有的进程  pid<0 将信号传给进程组识别码为pid绝对值的所有进程  参数sig代表的信号编号可参考附录D |
| 返回值 | 执行成功则返回0,如果有错误则返回-1。 |
| 错误代码 | EINVAL 参数sig不合法  ESRCH 参数pid所指定的进程或进程组不存在  EPERM 权限不够无法传送信号给指定进程 |
| 范例 |  |
| #include <unistd.h>  #include <signal.h>  #include <sys/types.h>  #include <sys/wait.h>  main(){  pid\_t pid;  int status;  if(!(pid= fork())){  printf(“Hi I am child process!\n”);  sleep(10);  return;  } else {  printf(“send signal to child process(%d) \n”,pid);  sleep(1);  kill(pid ,SIGABRT);  wait(&status);  if(WIFSIGNALED(status))  printf(“chile process receive signal %d\n”,WTERMSIG(status));  }  } | |
| 执行 |  |
| sen signal to child process(3170)  Hi I am child process!  child process receive signal 6 | |

## pause(让进程暂停直到信号出现)

|  |  |
| --- | --- |
| **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。 |

## signal(设置信号处理方式)

|  |  |
| --- | --- |
| **signal(设置信号处理方式)** | |
| 相关函数 | sigaction,kill,raise |
| 表头文件 | #include <signal.h> |
| 定义函数 | void(\*signal(int signum,void(\*handler)(int)))(int); |
| 函数说明 | signal()会依参数signum指定的信号编号来设置该信号的处理函数。当指定的信号到达时就会跳转到参数handler指定的函数执行。  如果参数handler不是函数指针,则必须是下列两个常数之一：  SIG\_IGN 忽略参数signum指定的信号。  SIG\_DFL 将参数signum指定的信号重设为核心预设的信号处理方式。  关于信号的编号和说明,请参考附录D |
| 返回值 | 返回先前的信号处理函数指针,如果有错误则返回SIG\_ERR(-1)。 |
| 附加说明 | 在信号发生跳转到自定的handler处理函数执行后,系统会自动将此处理函数换回原来系统预设的处理方式,如果要改变此操作请改用sigaction()。 |
| 范例 |  |
| 参考alarm()或raise()。 | |

## sigaction(查询或设置信号处理方式)

|  |  |
| --- | --- |
| **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启  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/SIGSTOPSIGKILL信号  EFAULT 参数act,oldact指针地址无法存取。  EINTR 此调用被中断 |
| 范例 |  |
| #include <unistd.h>  #include <signal.h>  void show\_handler(struct sigaction \*act){  switch(act->sa\_flags) {  case SIG\_DFL:printf(“Default action\n”);break;  case SIG\_IGN:printf(“Ignore the signal\n”);break;  default: printf(“0x%x\n”,act->sa\_handler);  }  }  main() {  int i;  struct sigaction act,oldact;  act.sa\_handler = show\_handler;  act.sa\_flags = SA\_ONESHOT|SA\_NOMASK;  sigaction(SIGUSR1,&act,&oldact);  for(i=5;i<15;i++) {  printf(“sa\_handler of signal %2d =”.i);  sigaction(i,NULL,&oldact);  }  } | |
| 执行 |  |
| sa\_handler of signal 5 = Default action  sa\_handler of signal 6= Default action  sa\_handler of signal 7 = Default action  sa\_handler of signal 8 = Default action  sa\_handler of signal 9 = Default action  sa\_handler of signal 10 = 0x8048400  sa\_handler of signal 11 = Default action  sa\_handler of signal 12 = Default action  sa\_handler of signal 13 = Default action  sa\_handler of signal 14 = Default action | |

## sigaddset(增加一个信号至信号集)

|  |  |
| --- | --- |
| **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)

|  |  |
| --- | --- |
| **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)

|  |  |
| --- | --- |
| **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)

|  |  |
| --- | --- |
| **sigfillset(将所有信号加入至信号集)** | |
| 相关函数 | sigempty,sigaddset,sigdelset,sigismember |
| 表头文件 | #include <signal.h> |
| 定义函数 | int sigfillset(sigset\_t \*set); |
| 函数说明 | sigfillset()用来将参数set信号集初始化,然后把所有的信号加入到此信号集里。 |
| 返回值 | 执行成功则返回0,如果有错误则返回-1。 |
| 附加说明 | EFAULT 参数set指针地址无法存取 |

## sigismember(测试某个信号是否已加入至信号集里)

|  |  |
| --- | --- |
| **sigismember(测试某个信号是否已加入至信号集里)** | |
| 相关函数 | sigemptyset,sigfillset,sigaddset,sigdelset |
| 表头文件 | #include <signal.h> |
| 定义函数 | int sigismember(const sigset\_t \*set,int signum); |
| 函数说明 | sigismember()用来测试参数signum代表的信号是否已加入至参数set信号集里。如果信号集里已有该信号则返回1,否则返回0。 |
| 返回值 | 信号集已有该信号则返回1,没有则返回0。如果有错误则返回-1。 |
| 错误代码 | EFAULT 参数set指针地址无法存取  EINVAL 参数signum 非合法的信号编号 |

## sigpending(查询被搁置的信号)

|  |  |
| --- | --- |
| **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)

|  |  |
| --- | --- |
| **sigprocmask(查询或设置信号遮罩)** | |
| 相关函数 | signal,sigaction,sigpending,sigsuspend |
| 表头文件 | #include <signal.h> |
| 定义函数 | int sigprocmask(int how,const sigset\_t \*set,sigset\_t \*oldset); |
| 函数说明 | sigprocmask()可以用来改变目前的信号遮罩,其操作依参数how来决定：  SIG\_BLOCK 新的信号遮罩由目前的信号遮罩和参数set指定的信号遮罩作联集  SIG\_UNBLOCK 将目前的信号遮罩删除掉参数set指定的信号遮罩  SIG\_SETMASK 将目前的信号遮罩设成参数set指定的信号遮罩。  如果参数oldset不是NULL指针,那么目前的信号遮罩会由此指针返回。 |
| 返回值 | 执行成功则返回0,如果有错误则返回-1。 |
| 错误代码 | EFAULT 参数set,oldset指针地址无法存取。  EINTR 此调用被中断 |

## sleep(让进程暂停执行一段时间)

|  |  |
| --- | --- |
| **sleep(让进程暂停执行一段时间)** | |
| 相关函数 | signal,alarm |
| 表头文件 | #include <unistd.h> |
| 定义函数 | unsigned int sleep(unsigned int seconds); |
| 函数说明 | sleep()会令目前的进程暂停,直到达到参数seconds所指定的时间,或是被信号所中断 |
| 返回值 | 若进程暂停到参数seconds所指定的时间则返回0,若有信号中断则返回剩余秒数。 |

# 用户组操作

## endgrent(关闭组文件)

|  |  |
| --- | --- |
| **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()与setgrent()。 | |

## endpwent(关闭密码文件)

|  |  |
| --- | --- |
| **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()与setpwent()。 | |

## endutent(关闭utmp文件)

|  |  |
| --- | --- |
| **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)

|  |  |
| --- | --- |
| **fgetgrent(从指定的文件来读取组格式)** | |
| 相关函数 | fgetpwent |
| 表头文件 | #include <grp.h>  #include <stdio.h>  #include <sys/types.h> |
| 定义函数 | struct group \*getgrent(FILE \*stream); |
| 函数说明 | fgetgrent()会从参数stream指定的文件读取一行数据,然后以group结构将该数据返回。参数stream所指定的文件必须和、etc/group相同的格式。group结构定义请参考getgrent()。 |
| 返回值 | 返回group结构数据,如果返回NULL则表示已无数据,或有错误发生。 |
| 范例 |  |
| #include <grp.h>  #include <sys/types.h>  #include <stdio.h>  main() {  struct group \*data;  FILE \*stream;  int i;  stream = fopen(“/etc/group”,”r”);  while((data = fgetgrent(stream))!=0){  i=0;  printf(“ %s :%s:%d :”,data->gr\_name,data->gr\_passwd,data->gr\_gid);  while(data->gr\_mem[i])printf(“%s,”,data->gr\_mem[i++]);  printf(“\n”);  }  fclose(stream);  } | |
| 执行 |  |
| root:x:0:root,  bin:x:1:root,bin,daemon  daemon:x:2:root,bin,daemon  sys:x:3:root,bin,adm  adm:x:4:root,adm,daemon  tty:x:5  disk:x:6:root  lp:x:7:daemon,lp  mem:x:8  kmem:x:9  wheel:x:10:root  mail:x:12:mail  news:x:13:news  uucp:x:14:uucp  man:x:15  games:x:20  gopher:x:30  dip:x:40:  ftp:x:50  nobody:x:99: | |

## fgetpwent(从指定的文件来读取密码格式)

|  |  |
| --- | --- |
| **fgetpwent(从指定的文件来读取密码格式)** | |
| 相关函数 | fgetgrent |
| 表头文件 | #include <pwd.h>  #include <stdio.h>  #include <sys/types.h> |
| 定义函数 | struct passwd \*fgetpwent(FILE \*stream); |
| 函数说明 | fgetpwent()会从参数stream指定的文件读取一行数据,然后以passwd结构将该数据返回。参数stream所指定的文件必须和/etc/passwd相同的格式。passwd结构定义请参考getpwent()。 |
| 返回值 | 返回passwd结构数据,如果返回NULL则表示已无数据,或有错误发生。 |
| 范例 |  |
| #include <pwd.h>  #include <sys/types.h>  main(){  struct passwd \*user;  FILE \*stream;  stream = fopen(“/etc/passwd”,”r”);  while((user = fgetpwent(stream))!=0){  printf(“%s:%d:%d:%s:%s:%s\n”,user->pw\_name,user->pw\_uid,user->pw\_gid,  user->pw\_gecos,user->pw\_dir,user->pw\_shell);  }  } | |
| 执行 |  |
| root:0:0:root:/root:/bin/bash  bin:1:1:bin:/bin:  daemon:2:2:daemon:/sbin:  adm:3:4:adm:/var/adm:  lp:4:7:lp:/var/spool/lpd:  sync:5:0:sync:/sbin:/bin/sync  shutdown:6:0:shutdown:/sbin:/sbin/shutdown  halt:7:0:halt:/sbin:/sbin/halt  mail:8:12:mail:/var/spool/mail:  news:9:13:news:var/spool/news  uucp:10:14:uucp:/var/spool/uucp:  operator:11:0:operator :/root:  games:12:100:games:/usr/games:  gopher:13:30:gopher:/usr/lib/gopher-data:  ftp:14:50:FTP User:/home/ftp:  nobody:99:99:Nobody:/:  xfs:100:101:X Font Server: /etc/Xll/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)

|  |  |
| --- | --- |
| **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组识别码。有效的组识别码用来决定进程执行时组的权限。 |
| 返回值 | 返回有效的组识别码。 |
| 范例 |  |
| main(){  printf(“egid is %d\n”,getegid());  } | |
| 执行 |  |
| egid is 0 /\*当使用root身份执行范例程序时\*/ | |

## geteuid(取得有效的用户识别码)

|  |  |
| --- | --- |
| **geteuid(取得有效的用户识别码)** | |
| 相关函数 | getuid,setreuid,setuid |
| 表头文件 | #include <unistd.h>  #include <sys/types.h> |
| 定义函数 | uid\_t geteuid(void) |
| 函数说明 | geteuid()用来取得执行目前进程有效的用户识别码。有效的用户识别码用来决定进程执行的权限,借由此改变此值,进程可以获得额外的权限。倘若执行文件的setID位已被设置,该文件执行时,其进程的euid值便会设成该文件所有者的uid。例如,执行文件/usr/bin/passwd的权限为-r-s--x--x,其s位即为setID(SUID)位,而当任何用户在执行passwd时其有效的用户识别码会被设成passwd所有者的uid值,即root的uid值(0)。 |
| 返回值 | 返回有效的用户识别码。 |
| 范例 |  |
| main(){  printf(“euid is %d \n”,geteuid());  } | |
| 执行 |  |
| euid is 0 /\*当使用root身份执行范例程序时\*/ | |

## getgid(取得真实的组识别码)

|  |  |
| --- | --- |
| **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 | getgid()用来取得执行目前进程的组识别码。 |
| 返回值 | 返回组识别码 |
| 范例 |  |
| main(){  printf(“gid is %d\n”,getgid());  } | |
| 执行 |  |
| gid is 0 /\*当使用root身份执行范例程序时\*/ | |

## getgrent(从组文件中取得账号的数据)

|  |  |
| --- | --- |
| **getgrent(从组文件中取得账号的数据)** | |
| 相关函数 | setgrent,endgrent |
| 表头文件 | #include <grp.h>  #include <sys/types.h> |
| 定义函数 | struct group \*getgrent(void); |
| 函数说明 | getgrent()用来从组文件(/etc/group)中读取一项组数据,该数据以group结构返回。第一次调用时会取得第一项组数据,之后每调用一次就会返回下一项数据,直到已无任何数据时返回NULL。  struct group{  char \*gr\_name**; /**\*组名称\*/  char \*gr\_passwd**; /**\*组密码\*/  gid\_t gr\_gid; /\*组识别码\*/  char \*\*gr\_mem; /\*组成员账号\*/  } |
| 返回值 | 返回group结构数据,如果返回NULL则表示已无数据,或有错误发生。 |
| 附加说明 | getgrent()在第一次调用时会打开组文件,读取数据完毕后可使用endgrent()来关闭该组文件。 |
| 错误代码 | ENOMEM 内存不足,无法配置group结构。 |
| 范例 |  |
| #include <grp.h>  #include <sys/types.h>  main(){  struct group \*data;  int i;  while((data= getgrent())!=0){  i=0;  printf(“%s:%s:%d:”,data->gr\_name,data->gr\_passwd,data->gr\_gid);  while(data->gr\_mem[i])printf(“%s,”,data->gr\_mem[i++]);  printf(“\n”);  }  endgrent();  } | |
| 执行 |  |
| root:x:0:root,  bin:x:1:root,bin,daemon,  daemon:x:2:root,bin,daemon,  sys:x:3:root,bin,adm,  adm:x:4:root,adm,daemon  tty:x:5  disk:x:6:root  lp:x:7:daemon,lp  mem:x:8  kmem:x:9:  wheel:x:10:root  mail:x:12:mail  news:x:13:news  uucp:x:14:uucp  man:x:15:  games:x:20  gopher:x:30  dip:x:40  ftp:x:50  nobody:x:99 | |

## getgrgid(从组文件中取得指定gid的数据)

|  |  |
| --- | --- |
| **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 | strcut 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(){  strcut group \*data;  int i=0;  data = getgrgid(3);  printf(“%s:%s:%d:”,data->gr\_name,data->gr\_passwd,data->gr\_gid);  while(data->gr\_mem[i])printf(“%s ,”,data->mem[i++]);  printf(“\n”);  } | |
| 执行 |  |
| sys:x:3:root,bin,adm | |

## getgrnam(从组文件中取得指定组的数据)

|  |  |
| --- | --- |
| **getgrnam(从组文件中取得指定组的数据)** | |
| 相关函数 | fgetgrent,getrent,getgruid |
| 表头文件 | #include <grp.h>  #include <sys/types.h> |
| 定义函数 | strcut group \*getgrnam(const char \*name); |
| 函数说明 | getgrnam()用来逐一搜索参数那么指定的组名称,找到时便将该组的数据以group结构返回。group结构请参考getgrent()。 |
| 返回值 | 返回group结构数据,如果返回NULL则表示已无数据,或有错误发生。 |
| 范例 |  |
| /\*取得adm的组数据\*/  #include <grp.h>  #include <sys/types.h>  main(){  strcut group \*data;  int i=0;  data = getgrnam(“adm”);  printf(“%s:%s:%d:”,data->gr\_name,data->gr\_passwd,data->gr\_gid);  while(data->gr\_mem[i])printf(“%s,”,data->gr\_mem[i++]);  printf(“\n”);  } | |
| 执行 |  |
| adm:x:4:root,adm,daemon | |

## getgroups(取得组代码)

|  |  |
| --- | --- |
| **getgroups(取得组代码)** | |
| 相关函数 | initgroups,setgroup,getgid,setgid |
| 表头文件 | #include <unistd.h>  #include <sys/types.h> |
| 定义函数 | int getgroups(int size,gid\_t list[]); |
| 函数说明 | getgroup()用来取得目前用户所属的组代码。参数size为list〔〕所能容纳的gid\_t数目。如果参数size值为零,此函数仅会返回用户所属的组数。 |
| 返回值 | 返回组识别码,如有错误则返回-1。 |
| 错误代码 | EFAULT 参数list数组地址不合法。  EINVAL 参数size值不足以容纳所有的组 |
| 范例 |  |
| #include <unistd.h>  #include <sys/types.h>  main(){  gid\_t list[500];  int x,i;  x = getgroups(0.list);  getgroups(x,list);  for(i=0;i<x;i++)  printf(“%d:%d\n”,i,list[i]);  } | |
| 执行 |  |
| 0:0  1:1  2:2  3:3  4:4  5:6  6:10 | |

## getpw(取得指定用户的密码文件数据)

|  |  |
| --- | --- |
| **getpw(取得指定用户的密码文件数据)** | |
| 相关函数 | getpwent |
| 表头文件 | #include <pwd.h>  #include <sys/types.h> |
| 定义函数 | int getpw(uid\_t uid,char \*buf); |
| 函数说明 | getpw()会从/etc/passwd中查找符合参数uid所指定的用户账号数据,找不到相关数据就返回-1。所返回的buf字符串格式如下：  账号：密码：用户识别码(uid)：组识别码(gid)：全名：根目录：shell |
| 返回值 | 返回0表示成功,有错误发生时返回-1。 |
| 附加说明 | getpw()会有潜在的安全性问题,请尽量使用别的函数取代。  使用shadow的系统已把用户密码抽出/etc/passwd,因此使用getpw()取得的密码将为“x”。 |
| 范例 |  |
| #include <pwd.h>  #include <sys/types.h>  main() {  char buffer[80];  getpw(0,buffer);  printf(“%s\n”,buffer);  } | |
| 执行 |  |
| root:x:0:0:root:/root:/bin/bash | |

## getpwent(从密码文件中取得账号的数据)

|  |  |
| --- | --- |
| **getpwent(从密码文件中取得账号的数据)** | |
| 相关函数 | getpw,fgetpwent,getpwnam,getpwuid,setpwent,endpwent |
| 表头文件 | #include <pwd.h>  #include <sys/types.h> |
| 定义函数 | strcut passwd \*getpwent(void); |
| 函数说明 | getpwent()用来从密码文件(/etc/passwd)中读取一项用户数据,该用户的数据以passwd结构返回。第一次调用时会取得第一位用户数据,之后每调用一次就会返回下一项数据,直到已无任何数据时返回NULL。  passwd结构定义如下：  struct passwd{  char \*pw\_name; /\*用户账号\*/  char \*pw\_passwd; /\*用户密码\*/  uid\_t pw\_uid; /\*用户识别码\*/  gid\_t pw\_gid; /\*组识别码\*/  char \*pw\_gecos; /\*用户全名\*/  char \*pw\_dir; /\*家目录\*/  char \*pw\_shell; /\*所使用的shell路径\*/  }; |
| 返回值 | 返回passwd结构数据,如果返回NULL则表示已无数据,或有错误发生。 |
| 附加说明 | getpwent()在第一次调用时会打开密码文件,读取数据完毕后可使用endpwent()来关闭该密码文件。 |
| 错误代码 | ENOMEM 内存不足,无法配置passwd结构。 |
| 范例 |  |
| #include <pwd.h>  #include <sys/types.h>  main(){  struct passwd \*user;  while((user = getpwent())!=0){  printf(“%s:%d:%d:%s:%s:%s\n”,user->pw\_name,user->pw\_uid,user->pw\_gid,  user->pw\_gecos,user->pw\_dir,user->pw\_shell);  }  endpwent();  } | |
| 执行 |  |
| root:0:0:root:/root:/bin/bash  bin:1:1:bin:/bin:  daemon:2:2:daemon:/sbin:  adm:3:4:adm:/var/adm:  lp:4:7:lp:/var/spool/lpd:  sync:5:0:sync:/sbin:/bin/sync  shutdown:6:0:shutdown:/sbin:/sbin/shutdown  halt:7:0:halt:/sbin:/sbin/halt  mail:8:12:mail:/var/spool/mail:  news:9:13:news:var/spool/news  uucp:10:14:uucp:/var/spool/uucp:  operator:11:0:operator :/root:  games:12:100:games:/usr/games:  gopher:13:30:gopher:/usr/lib/gopher-data:  ftp:14:50:FTP User:/home/ftp:  nobody:99:99:Nobody:/:  xfs:100:101:X Font Server: /etc/Xll/fs:/bin/false  gdm:42:42:/home/gdm:/bin/bash  kids:500:500: : /home/kids:/bin/bash | |

## getpwnam(从密码文件中取得指定账号的数据)

|  |  |
| --- | --- |
| **getpwnam(从密码文件中取得指定账号的数据)** | |
| 相关函数 | getpw,fgetpwent,getpwent,getpwuid |
| 表头文件 | #include <pwd.h>  #include <sys/types.h> |
| 定义函数 | struct passwd \*getpwnam(const char \*name); |
| 函数说明 | getpwnam()用来逐一搜索参数name指定的账号名称,找到时便将该用户的数据以passwd结构返回。passwd结构请参考getpwent()。 |
| 返回值 | 返回passwd结构数据,如果返回NULL则表示已无数据,或有错误发生。 |
| 范例 |  |
| /\*取得root账号的识别码和根目录\*/  #include <pwd.h>  #include <sys/types.h>  main(){  struct passwd \*user;  user = getpwnam(“root”);  printf(“name:%s\n”,user->pw\_name);  printf(“uid:%d\n”,user->pw\_uid);  printf(“home:%s\n”,user->pw\_dir);  } | |
| 执行 |  |
| name:root  uid:0  home:/root | |

## getpwuid(从密码文件中取得指定uid的数据)

|  |  |
| --- | --- |
| **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,或者有错误发生。 |
| 范例 |  |
| #include <pwd.h>  #include <sys/types.h>  main(){  struct passwd \*user;  user= getpwuid(6);  printf(“name:%s\n”,user->pw\_name);  printf(“uid:%d\n”,user->pw\_uid);  printf(“home:%s\n”,user->pw\_dir);  } | |
| 执行 |  |
| name:shutdown  uid:6  home:/sbin | |

## getuid(取得真实的用户识别码)

|  |  |
| --- | --- |
| **getuid(取得真实的用户识别码)** | |
| 相关函数 | geteuid,setreuid,setuid |
| 表头文件 | #include <unistd.h>  #include <sys/types.h> |
| 定义函数 | uid\_t getuid(void); |
| 函数说明 | getuid()用来取得执行目前进程的用户识别码。 |
| 返回值 | 用户识别码 |
| 范例 |  |
| main(){  printf(“uid is %d\n”,getuid());  } | |
| 执行 |  |
| uid is 0 /\*当使用root身份执行范例程序时\*/ | |

## getutent(从utmp文件中取得账号登录数据)

|  |  |
| --- | --- |
| **getutent(从utmp文件中取得账号登录数据)** | |
| 相关函数 | getutent,getutid,getutline,setutent,endutent,pututline,utmpname |
| 表头文件 | #include <utmp.h> |
| 定义函数 | struct utmp \*getutent(void); |
| 函 数说明 | getutent()用来从utmp文件(/var/run/utmp)中读取一项登录数据,该数据以utmp结构返回。第一次调用时会取得第一位用户数据,之后每调用一次就会返回下一项数据,直到已无任何数据时返回NULL。  utmp结构定义如下：  struct utmp{  short int ut\_type; /\*登录类型\*/  pid\_t ut\_pid; /\*login进程的pid\*/  char ut\_line[UT\_LINESIZE];/\*登录装置名,省略了“/dev/”\*/  char ut\_id[4]; /\*Inittab ID\*/  char ut\_user[UT\_NAMESIZE];/\*登录账号\*/  char ut\_host[UT\_HOSTSIZE];/\*登录账号的远程主机名称\*/  struxt exit\_status ut\_exit;/\*当类型为DEAD\_PROCESS时进程的结束状态\*/  long int ut\_session; /\*Sessioc ID\*/  struct timeval ut\_tv; /\*时间记录\*/  int32\_t ut\_addr\_v6[4]; /\*远程主机的网络地址\*/  char \_\_unused[20]; /\*保留未使用\*/  };  ut\_type有以下几种类型：  EMPTY 此为空的记录。  RUN\_LVL 记录系统run－level的改变  BOOT\_TIME 记录系统开机时间  NEW\_TIME 记录系统时间改变后的时间  OLD\_TINE 记录当改变系统时间时的时间。  INIT\_PROCESS 记录一个由init衍生出来的进程。  LOGIN\_PROCESS 记录login进程。  USER\_PROCESS 记录一般进程。  DEAD\_PROCESS 记录一结束的进程。  ACCOUNTING 目前尚未使用。  exit\_status结构定义：  struct exit\_status {  short int e\_termination; /\*进程结束状态\*/  short int e\_exit; /\*进程退出状态\*/  };  timeval的结构定义请参考gettimeofday()。  相关常数定义如下：  UT\_LINESIZE 32  UT\_NAMESIZE 32  UT\_HOSTSIZE 256 |
| 返回值 | 返回utmp结构数据,如果返回NULL则表示已无数据,或有错误发生。 |
| 附加说明 | getutent()在第一次调用时会打开utmp文件,读取数据完毕后可使用endutent()来关闭该utmp文件。 |
| 范例 |  |
| #include <utmp.h>  main() {  struct utmp \*u;  while((u=getutent())){  if(u->ut\_type = = USER\_PROCESS)  printf(“%d %s %s %s \n”,u->ut\_type,u->ut\_user,u->ut\_line,u->ut\_host);  }  endutent();  } | |
| 执行 |  |
| /\*表示有三个root账号分别登录/dev/pts/0,/dev/pts/1,/dev/pts/2 \*/  7 root pts/0  7 root pts/1  7 root pts/2 | |

## getutid(从utmp文件中查找特定的记录)

|  |  |
| --- | --- |
| **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 | strcut utmp \*getutid(strcut utmp \*ut); |
| 函数说明 | getutid()用来从目前utmp文件的读写位置逐一往后搜索参数ut指定的记录,如果ut->ut\_type为RUN\_LVL,BOOT\_TIME,NEW\_TIME,OLD\_TIME其中之一则查找与ut->ut\_type相符的记录；  若ut->ut\_type为INIT\_PROCESS,LOGIN\_PROCESS,USER\_PROCESS或DEAD\_PROCESS其中之一,则查找与ut->ut\_id相符的记录。找到相符的记录便将该数据以utmp结构返回。utmp结构请参考getutent()。 |
| 返回值 | 返回utmp结构数据,如果返回NULL则表示已无数据,或有错误发生。 |
| 范例 |  |
| #include <utmp.h>  main(){  struct utmp ut,\*u;  ut.ut\_type=RUN\_LVL;  while((u= getutid(&ut))){  printf(“%d %s %s %s\n”,u->ut\_type,u->ut\_user,u->ut\_line,u->ut\_host);  }  } | |
| 执行 |  |
| 1 runlevel - | |

## getutline(从utmp文件中查找特定的记录)

|  |  |
| --- | --- |
| **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 |  |
| 7 root pts/1 | |

## initgroups(初始化组清单)

|  |  |
| --- | --- |
| **initgroups(初始化组清单)** | |
| 相关函数 | setgrent,endgrent |
| 表头文件 | #include <grp.h>  #include <sys/types.h> |
| 定义函数 | int initgroups(const char \*user,gid\_t group); |
| 函数说明 | initgroups()用来从组文件(/etc/group)中读取一项组数据,若该组数据的成员中有参数user时,便将参数group组识别码加入到此数据中。 |
| 返回值 | 执行成功则返回0,失败则返回-1,错误码存于errno。 |
|  | |

## pututline(将utmp记录写入文件)

|  |  |
| --- | --- |
| **pututline(将utmp记录写入文件)** | |
| 相关函数 | getutent,getutid,getutline |
| 表头文件 | #include <utmp.h> |
| 定义函数 | void pututline(struct utmp \*ut); |
| 函数说明 | pututline()用来将参数ut的utmp结构记录到utmp文件中。此函数会先用getutid()来取得正确的写入位置,如果没有找到相符的记录则会加入到utmp文件尾,utmp结构请参考getutent()。 |
| 返回值 | 无 |
| 附加说明 | 需要有写入/var/run/utmp的权限 |
| 范例 |  |
| #include <utmp.h>  main(){  struct utmp ut;  ut.ut\_type =USER\_PROCESS;  ut.ut\_pid=getpid();  strcpy(ut.ut\_user,”kids”);  strcpy(ut.ut\_line,”pts/1”);  strcpy(ut.ut\_host,”www.gnu.org”);  pututline(&ut);  } | |
| 执行 |  |
| /\*执行范例后用指令who -l 观察\*/  root pts/0 dec9 19:20  kids pts/1 dec12 10:31(www.gnu.org)  root pts/2 dec12 13:33 | |

## seteuid(设置有效的用户识别码)

|  |  |
| --- | --- |
| **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)

|  |  |
| --- | --- |
| **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。 |
| 附加说明 | 此函数为Linux特有。 |
| 错误代码 | EPERM 权限不够,无法完成设置。 |
|  | |

## setfsuid(设置文件系统的用户识别码)

|  |  |
| --- | --- |
| **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 |
| 附加说明 | 此函数为Linux特有 |
| 错误代码 | EPERM 权限不够,无法完成设置。 |
|  | |

## setgid(设置真实的组识别码)

|  |  |
| --- | --- |
| **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)

|  |  |
| --- | --- |
| **setgrent(从头读取组文件中的组数据)** | |
| 相关函数 | getgrent,endgrent |
| 表头文件 | #include <grp.h>  #include <sys/types.h> |
| 定义函数 | void setgrent(void); |
| 函数说明 | setgrent()用来将getgrent()的读写地址指回组文件开头。 |
| 返回值 | 无 |
| 附加说明 | 请参考setpwent()。 |
|  | |

## setgroups(设置组代码)

|  |  |
| --- | --- |
| **setgroups(设置组代码)** | |
| 相关函数 | initgroups,getgroup,getgid,setgid |
| 表头文件 | #include <grp.h> |
| 定义函数 | int setgroups(size\_t size,const gid\_t \*list); |
| 函数说明 | setgroups()用来将list数组中所标明的组加入到目前进程的组设置中。参数size为list〔〕的gid\_t数目,最大值为NGROUP(32)。 |
| 返回值 | 设置成功则返回0,如有错误则返回-1。 |
| 错误代码 | EFAULT 参数list数组地址不合法。  EPERM 权限不足,必须是root权限  EINVAL 参数size值大于NGROUP(32)。 |
|  | |

## setpwent(从头读取密码文件中的账号数据)

|  |  |
| --- | --- |
| **setpwent(从头读取密码文件中的账号数据)** | |
| 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();  user=getpwent();  printf(“%s :%d :%d :%s:%s:%s\n”,user->pw\_name,user->pw\_uid,user->pw\_gid,  user->pw\_gecos,user->pw\_dir,user->pw\_shell);  endpwent();  } | |
| 执行 |  |
| root:0:0:root:/root:/bin/bash  bin:1:1:bin:/bin  daemon:2:2:daemon:/sbin  adm:3:4:adm:/var/adm  root:0:0:root:/root:/bin/bash | |

## setregid(设置真实及有效的组识别码)

|  |  |
| --- | --- |
| **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)

|  |  |
| --- | --- |
| **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)

|  |  |
| --- | --- |
| **setuid(设置真实的用户识别码)** | |
| 相关函数 | getuid,setreuid,seteuid,setfsuid |
| 表头文件 | #include <unistd.h> |
| 定义函数 | int setuid(uid\_t uid) |
| 函数说明 | setuid()用来重新设置执行目前进程的用户识别码。不过,要让此函数有作用,其有效的用户识别码必须为0(root)。在Linux下,当root使用setuid()来变换成其他用户识别码时,root权限会被抛弃,完全转换成该用户身份,也就是说,该进程往后将不再具有可setuid()的权利,如果只是向暂时抛弃root权限,稍后想重新取回权限,则必须使用seteuid()。 |
| 返回值 | 执行成功则返回0,失败则返回-1,错误代码存于errno。 |
| 附加说明 | 一般在编写具setuid root 的程序时,为减少此类程序带来的系统安全风险,在使用完root权限后建议马上执行setuid(getuid());来抛弃root权限。此外,进程uid和euid不一致时Linux系统将不会产生core dump。 |
|  | |

## setutent(从头读取utmp文件中的登录数据)

|  |  |
| --- | --- |
| **setutent(从头读取utmp文件中的登录数据)** | |
| 相关函数 | getutent,endutent |
| 表头文件 | #include <utmp.h> |
| 定义函数 | void setutent(void); |
| 函数说明 | setutent()用来将getutent()的读写地址指回utmp文件开头。 |
| 附加说明 | 请参考setpwent()或setgrent()。 |
|  | |

## utmpname(设置utmp文件路径)

|  |  |
| --- | --- |
| **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 |

# 数据结构及算法

## crypt(将密码或数据编码)

|  |  |
| --- | --- |
| **crypt(将密码或数据编码)** | |
| 相关函数 | getpass |
| 表头文件 | #define \_XOPEN\_SOURCE  #include <unistd.h> |
| 定义函数 | char \*crypt(const char \*key,const char \*salt); |
| 函数说明 | crypt()将使用Data Encryption Standard(DES)演算法将参数key所指的字符串加以编码,key字符串长度仅取前8个字符,超过此长度的字符没有意义。参数salt为两个字符组成的字符串,由a-z、A-Z、0-9,“.”和“/”所组成,用来决定使用4096种不同内建表格的哪一个。函数执行成功后会返回指向编码过的字符串指针,参数key所指的字符串不会有所更动。编码过的字符串长度为13个字符,前两个字符为参数salt代表的字符串。 |
| 返回值 | 返回一个指向以NULL结尾的密码字符串。 |
| 附加说明 | 使用GCC编译时需加-lcrypt。 |
| 范例 |  |
| #include <unistd.h>  main(){  char passwd[13];  char \*key;  char slat[2];  key= getpass(“Input First Password:”);  slat[0]=key[0];  slat[1]=key[1];  strcpy(passwd,crypt(key slat));  key=getpass(“Input Second Password:”);  slat[0]=passwd[0];  slat[1]=passwd[1];  printf(“After crypt(),1st passwd :%s\n”,passwd);  printf(“After crypt(),2nd passwd:%s \n”,crypt(key,slat));  } | |
| 执行 |  |
| Input First Password: /\*输入test,编码后存于passwd[ ] \*/  Input Second Password /\*输入test,密码相同编码后也会相同\*/  After crypt() 1st Passwd : teH0wLIpW0gyQ  After crypt() 2nd Passwd : teH0wLIpW0gyQ | |

## bsearch(二元搜索)

|  |  |
| --- | --- |
| **bsearch(二元搜索)** | |
| 相关函数 | qsort |
| 表头文件 | #include <stdlib.h> |
| 定义函数 | void \*bsearch(const void \*key,const void \*base,size\_t nmemb,size\_t size,int(\*compar)(const void\*,const void\*)); |
| 函数说明 | bsearch()利用二元搜索从排序好的数组中查找数据。参数key指向欲查找的关键数据,参数base指向要被搜索的数组开头地址,参数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之间的大小关系,若传给compar的第一个参数所指的元素数据大于第二个参数所指的元素数据则必须回传大于0的值,两个元素数据相等则回传0。 |
| 附加说明 | 找到关键数据则返回找到的地址,如果在数组中找不到关键数据则返回NULL。 |
| 范例 |  |
| #include <stdio.h>  #include <stdlib.h>  #define NMEMB 5  #define SIZE 10  int compar(const void \*a,const void \*b){  return(strcmp((char \*)a,(char \*)b));  }  main(){  char data[50][size]={“linux”,”freebsd”,”solaris”,”sunos”,”windows”};  char key[80],\*base ,\*offset;  int i, nmemb=NMEMB,size=SIZE;  while(1){  printf(“>”);  fgets(key,sizeof(key),stdin);  key[strlen(key)-1]=’\0’;  if(!strcmp(key,”exit”))break;  if(!strcmp(key,”list”)){  for(i=0;i<nmemb;i++)  printf(“%s\n”,data[i]);  continue;  }  base = data[0];  qsort(base,nmemb,size,compar);  offset =(char \*) bsearch(key,base,nmemb,size,compar);  if( offset = =NULL){  printf(“%s not found!\n”,key);  strcpy(data[nmemb++],key);  printf(“Add %s to data array\n”,key);  } else {  printf(“found: %s \n”,offset);  }  }  } | |
| 执行 |  |
| >hello /\*输入hello字符串\*/  hello not found! /\*找不到hello字符串\*/  add hello to data array /\*将hello字符串加入\*/  >.list /\*列出所有数据\*/  freebsd  linux  solaris  sunos  windows  hello  >hello  found: hello | |

## lfind(线性搜索)

|  |  |
| --- | --- |
| **lfind(线性搜索)** | |
| 相关函数 | lsearch |
| 表头文件 | #include <stdlib.h> |
| 定义函数 | void \*lfind(const void \*key,const void \*base,size\_t \*nmemb,size\_t size,int(\*compar)(const void \*,const void \*)); |
| 函数说明 | lfind(()利用线性搜索在数组中从头至尾一项项查找数据。参数key指向欲查找的关键数据,参数base指向要被搜索的数组开头地址,参数nmemb代表数组中的元素数量,每一元素的大小则由参数size决定,最后一项参数compar为一函数指针,这个函数用来判断两个元素是否相同,若传给compar的异地个参数所指的元素数据和第二个参数所指的元素数据相同时则返回0,两个元素数据不相同则返回非0值。lfind()与lsearch()不同点在于,当找不到关键数据时lfind()仅会返回NULL,而不会主动把该笔数据加入数组尾端。 |
| 返回值 | 找到关键数据则返回找到的该笔元素的地址,如果在数组中找不到关键数据则返回空指针(NULL)。 |
| 范例 |  |
| 参考lsearch()。 | |

## lsearch(线性搜索)

|  |  |
| --- | --- |
| **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四肢,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(comst void \*a,const void \*b){  return(strcmp((char \*) a,(char \*) b));  }  main(){  char data[NMEMB][SIZE]={“Linux”,”freebsd”,”solzris”,”sunos”,”windows”};  char key[80],\*base,\*offset;  int i, nmemb=NMEMB,size=SIZE;  for(i=1;i<5;i++){  fgets(key,sizeof9key),stdin);  key[strlen(key)-1]=’\0’;  base = data[0];  offset =(char \*)lfind(key,base,&nmemb,size,compar);  if(offset ==NULL){  printf(“%s not found!\n”,key);  offset=(char \*) lsearch(key,base,&nmemb,size,compar);  printf(“Add %s to data array\n”,offset);  }else{  printf(“found : %s \n”,offset);  }  }  } | |
| 执行 |  |
| linux  found:linux  os/2  os/2 not found!  add os/2 to data array  os/2  found : os/2 | |

## qsort(利用快速排序法排列数组)

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

## rand(产生随机数)

|  |  |
| --- | --- |
| **rand(产生随机数)** | |
| 相关函数 | srand,random,srandom |
| 表头文件 | #include <stdlib.h> |
| 定义函数 | int rand(void) |
| 函数说明 | rand()会返回一随机数值,范围在0至RAND\_MAX间。在调用此函数产生随机数前,必须先利用srand()设好随机数种子,如果未设随机数种子,rand()在调用时会自动设随机数种子为1。关于随机数种子请参考srand()。 |
| 返回值 | 返回0至RAND\_MAX之间的随机数值,RAND\_MAX定义在stdlib.h,其值为2147483647。 |
| 范例 |  |
| /\*产生介于1到10间的随机数值,此范例未设随机数种子,完整的随机数产生请参考srand()\*/  #include <stdlib.h>  main(){  int i,j;  for(i=0;i<10;i++) {  j=1+(int)(10.0\*rand()/(RAND\_MAX+1.0));  printf(“%d “,j);  }  } | |
| 执行 |  |
| 9 4 8 8 10 2 4 8 3 6  9 4 8 8 10 2 4 8 3 6 | |

## srand(设置随机数种子)

|  |  |
| --- | --- |
| **srand(设置随机数种子)** | |
| 相关函数 | rand,random srandom |
| 表头文件 | #include <stdlib.h> |
| 定义函数 | void srand(unsigned int seed); |
| 函数说明 | srand()用来设置rand()产生随机数时的随机数种子。参数seed必须是个整数,通常可以利用getpid()或time(0)的返回值来当做seed。如果每次seed都设相同值,rand()所产生的随机数值每次就会一样。 |
| 返回值 | 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));  for(i=0;i<10;i++) {  j=1+(int)(10.0\*rand()/(RAND\_MAX+1.0));  printf(“ %d “,j);  }  } | |
| 执行 |  |
| 5 8 8 8 10 2 10 8 9 9  2 9 7 4 10 3 2 10 8 7 | |

# 日期时间篇

## time(取得目前的时间)

|  |  |
| --- | --- |
| **time(取得目前的时间)** | |
| 相关函数 | ctime,ftime,gettimeofday |
| 表头文件 | #include <time.h> |
| 定义函数 | time\_t time(time\_t \*t); |
| 函数说明 | 此函数会返回从公元1970年1月1日的UTC时间从0时0分0秒算起到现在所经过的秒数。如果t并非空指针的话,此函数也会将返回值存到t指针所指的内存。 |
| 返回值 | 成功则返回秒数,失败则返回((time\_t)-1)值,错误原因存于errno中。 |
| 范例 |  |
| #include <time.h>  mian(){  int seconds= time((time\_t\*)NULL);  printf(“%d\n”,seconds);  } | |
| 执行 |  |
| 972699100 | |

## asctime(将时间和日期以字符串格式表示)

|  |  |
| --- | --- |
| **asctime(将时间和日期以字符串格式表示)** | |
| 相关函数 | time,ctime,gmtime,localtime |
| 表头文件 | #include <time.h> |
| 定义函数 | char \*asctime(const struct tm \*timeptr); |
| 函数说明 | asctime()将参数timeptr所指的tm结构中的信息转换成真实世界所使用的时间日期表示方法,然后将结果以字符串形态返回。此函数已经由时区转换成当地时间,字符串格式为：  “Wed Jun 30 21:49:08 1993\n” |
| 返回值 | 若再调用相关的时间日期函数,此字符串可能会被破坏。此函数与ctime不同处在于传入的参数是不同的结构。 |
| 附加说明 | 返回一字符串表示目前当地的时间日期。 |
| 范例 |  |
| #include <time.h>  main(){  time\_t timep;  time(&timep);  printf(“%s”,asctime(gmtime(&timep)));  } | |
| 执行 |  |
| Sat Oct 28 02:10:06 2000 | |

## ctime(将时间和日期以字符串格式表示)

|  |  |
| --- | --- |
| **ctime(将时间和日期以字符串格式表示)** | |
| 相关函数 | time,asctime,gmtime,localtime |
| 表头文件 | #include <time.h> |
| 定义函数 | char \*ctime(const time\_t \*timep); |
| 函数说明 | ctime()将参数timep所指的time\_t结构中的信息转换成真实世界所使用的时间日期表示方法,然后将结果以字符串形态返回。此函数已经由时区转换成当地时间,字符串格式为“Wed Jun 30 21 :49 08 1993\n”。若再调用相关的时间日期函数,此字符串可能会被破坏。 |
| 返回值 | 返回一字符串表示目前当地的时间日期。 |
| 范例 |  |
| #include <time.h>  main(){  time\_t timep;  time(&timep);  printf(“%s”,ctime(&timep));  } | |
| 执行 |  |
| Sat Oct 28 10 : 12 : 05 2000 | |

## settimeofday(设置目前时间)

|  |  |
| --- | --- |
| **settimeofday(设置目前时间)** | |
| 相关函数 | time,ctime,ftime,gettimeofday |
| 表头文件 | #include <sys/time.h>  #include <unistd.h> |
| 定义函数 | int settimeofday( const struct timeval \*tv,const struct timezone \*tz); |
| 函数说明 | settimeofday()会把目前时间设成由tv所指的结构信息,当地时区信息则设成tz所指的结构。详细的说明请参考gettimeofday()。注意,只有root权限才能使用此函数修改时间。 |
| 返回值 | 成功则返回0,失败返回－1,错误代码存于errno。 |
| 错误代码 | EPERM 并非由root权限调用settimeofday(),权限不够。  EINVAL 时区或某个数据是不正确的,无法正确设置时间。 |

## gettimeofday(取得目前的时间)

|  |  |
| --- | --- |
| **gettimeofday(取得目前的时间)** | |
| 相关函数 | time,ctime,ftime,settimeofday |
| 表头文件 | #include <sys/time.h>  #include <unistd.h> |
| 定义函数 | int gettimeofday( struct timeval \*tv , struct timezone \*tz ) |
| 函数说明 | gettimeofday() 会把目前的时间有tv 所指的结构返回,当地时区的信息则放到tz所指的结构中。timeval结构定义为：  struct timeval{  long tv\_sec; /\*秒\*/  long tv\_usec; /\*微秒\*/  ｝；  timezone 结构定义为：  struct timezone{  int tz\_minuteswest; /\*和Greenwich时间差了多少分钟\*/  int tz\_dsttime; /\*日光节约时间的状态\*/  ｝；  上述两个结构都定义在 /usr/include/sys/time.h。tz\_dsttime 所代表的状态如下：  DST\_NONE /\*不使用\*/  DST\_USA /\*美国\*/  DST\_AUST /\*澳洲\*/  DST\_WET /\*西欧\*/  DST\_MET /\*中欧\*/  DST\_EET /\*东欧\*/  DST\_CAN /\*加拿大\*/  DST\_GB /\*大不列颠\*/  DST\_RUM /\*罗马尼亚\*/  DST\_TUR /\*土耳其\*/  DST\_AUSTALT /\*澳洲(1986年以后)\*/ |
| 返回值 | 成功则返回0,失败返回－1,错误代码存于errno。 |
| 附加说明 | EFAULT指针tv和tz所指的内存空间超出存取权限。 |
| 范例 |  |
| #include <sys/time.h>  #include <unistd.h>  main(){  struct timeval tv;  struct timezone tz;  gettimeofday(&tv , &tz);  printf(“tv\_sec; %d\n”, tv,.tv\_sec) ;  printf(“tv\_usec; %d\n”,tv.tv\_usec);  printf(“tz\_minuteswest; %d\n”, tz.tz\_minuteswest);  printf(“tz\_dsttime, %d\n”,tz.tz\_dsttime);  } | |
| 执行 |  |
| tv\_sec: 974857339  tv\_usec:136996  tz\_minuteswest:-540  tz\_dsttime:0 | |
|  | |

## gmtime(取得目前时间和日期)

|  |  |
| --- | --- |
| **gmtime(取得目前时间和日期)** | |
| 相关函数 | time,asctime,ctime,localtime |
| 表头文件 | #include <time.h> |
| 定义函数 | struct tm\*gmtime(const time\_t\*timep); |
| 函数说明 | gmtime()将参数timep所指的time\_t结构中的信息转换成真实世界所使用的时间日期表示方法,然后将结果由结构tm返回。结构tm的定义为：  struct tm{  int tm\_sec;  int tm\_min;  int tm\_hour;  int tm\_mday;  int tm\_mon;  int tm\_year;  int tm\_wday;  int tm\_yday;  int tm\_isdst;  };  int tm\_sec 代表目前秒数,正常范围为0-59,但允许至61秒  int tm\_min 代表目前分数,范围0-59  int tm\_hour 从午夜算起的时数,范围为0-23  int tm\_mday 目前月份的日数,范围01-31  int tm\_mon 代表目前月份,从一月算起,范围从0-11  int tm\_year 从1900年算起至今的年数  int tm\_wday 一星期的日数,从星期一算起,范围为0-6  int tm\_yday 从今年1月1日算起至今的天数,范围为0-365  int tm\_isdst 日光节约时间的旗标  此函数返回的时间日期未经时区转换,而是UTC时间。 |
| 返回值 | 返回结构tm代表目前UTC时间 |
| 范例 |  |
| #include <time.h>  main(){  char \*wday[]={“Sun”,”Mon”,”Tue”,”Wed”,”Thu”,”Fri”,”Sat” };  time\_t timep;  struct tm \*p;  time(&timep);  p=gmtime(&timep);  printf(“%d%d%d”,(1900+p->tm\_year),(1+p->tm\_mon),p->tm\_mday);  printf(“%s%d;%d;%d\n”, wday[p->tm\_wday], p->tm\_hour, p->tm\_min, p->tm\_sec);  } | |
| 执行 |  |
| 2000/10/28 Sat 8:15:38 | |

## localtime(取得当地目前时间和日期)

|  |  |
| --- | --- |
| **localtime(取得当地目前时间和日期)** | |
| 相关函数 | time, asctime, ctime, gmtime |
| 表头文件 | #include <time.h> |
| 定义函数 | struct tm \*localtime(const time\_t \*timep); |
| 函数说明 | localtime()将参数timep所指的time\_t结构中的信息转换成真实世界所使用的时间日期表示方法,然后将结果由结构tm返回。结构tm的定义请参考gmtime()。此函数返回的时间日期已经转换成当地时区。 |
| 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),( l+p->tm\_mon), p->tm\_mday);  printf(“%s%d:%d:%d\n”, wday[p->tm\_wday],p->tm\_hour, p->tm\_min, p->tm\_sec);  } | |
| 执行 |  |
| 2000/10/28 Sat 11:12:22 | |

## mktime(将时间结构数据转换成经过的秒数)

|  |  |
| --- | --- |
| **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(strcut 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mktine()将struct tm转换成原来的秒数 \*/  #include <time.h>  main(){  time\_t timep;  strcut tm \*p;  time(&timep);  printf(“time() : %d \n”,timep);  p=localtime(&timep);  timep = mktime(p);  printf(“time()->localtime()->mktime():%d\n”,timep);  } | |
| 执行 |  |
| time(): 974943297  time()->localtime()->mktime():974943297 | |

# 常用数学函数

## abs(计算整型数的绝对值)

|  |  |
| --- | --- |
| **abs(计算整型数的绝对值)** | |
| 相关函数 | labs, fabs |
| 表头文件 | #include <stdlib.h> |
| 定义函数 | int abs(int j) |
| 函数说明 | abs() 用来计算参数j的绝对值,然后将结果返回。 |
| 返回值 | 返回参数j的绝对值结果。 |
| 范例 |  |
| #ingclude <stdlib.h>  main(){  int ansert;  answer = abs(-12);  printf(“|-12| = %d\n” , answer);  } | |
| 执行 |  |
| |-12| = 12 | |

## sin(取正玄函数值)

|  |  |
| --- | --- |
| **sin(取正玄函数值)** | |
| 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的正玄值,然后将结果返回。 |
| 返回值 | 返回-1至1之间的计算结果。 |
| 附加说明 | 使用GCC编译时请加入-lm。 |
| 范例 |  |
| #include <math.h>  main(){  double answer = sin(0.5);  printf(“sin(0.5) = %f\n”,answer);  } | |
| 执行 |  |
| sin(0.5) = 0.479426 | |

## sinh(取双曲线正玄函数值)

|  |  |
| --- | --- |
| **sinh(取双曲线正玄函数值)** | |
| 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的双曲线正玄值,然后将结果返回。数学定义式为：(exp(x)-exp(-x))/2。 |
| 返回值 | 返回参数x的双曲线正玄值。 |
| 附加说明 | 使用GCC编译时请加入-lm。 |
| 范例 |  |
| #include <math.h>  main(){  double answer = sinh(0.5);  printf(“sinh(0.5) = %f\n”,answer);  } | |
| 执行 |  |
| sinh(0.5) = 0.521095 | |

## cos(取余玄函数值)

|  |  |
| --- | --- |
| **cos(取余玄函数值)** | |
| 相关函数 | acos,asin,atan,atan2,sin,tan |
| 表头文件 | #include <math.h> |
| 定义函数 | double cos(double x); |
| 函数说明 | cos()用来计算参数x的余玄值,然后将结果返回。 |
| 返回值 | 返回-1至1之间的计算结果。 |
| 附加说明 | 使用GCC编译时请加入-lm。 |
| 范例 |  |
| #include <math.h>  main(){  double answer = cos(0.5);  printf(“cos(0.5) = %f\n”,answer);  } | |
| 执行 |  |
| cos(0.5) = 0.877583 | |

## cosh(取双曲线余玄函数值)

|  |  |
| --- | --- |
| **cosh(取双曲线余玄函数值)** | |
| 相关函数 | sinh,tanh |
| 表头文件 | #include <math.h> |
| 定义函数 | double cosh(double x); |
| 函数说明 | cosh()用来计算参数x的双曲线余玄值,然后将结果返回。数学定义式为：(exp(x)+exp(-x))/2。 |
| 返回值 | 返回参数x的双曲线余玄值。 |
| 附加说明 | 使用GCC编译时请加入-lm。 |
| 范例 |  |
| #include <math.h>  main(){  double answer = cosh(0.5);  printf(“cosh(0.5) = %f\n”,answer);  } | |
| 执行 |  |
| cosh(0.5) = 1.127626 | |

## tan(取正切函数值)

|  |  |
| --- | --- |
| **tan(取正切函数值)** | |
| 相关函数 | atan,atan2,cos,sin |
| 表头文件 | #include <math.h> |
| 定义函数 | double tan(double x); |
| 函数说明 | tan()用来计算参数x的正切值,然后将结果返回。 |
| 返回值 | 返回参数x的正切值。 |
| 附加说明 | 使用GCC编译时请加入-lm。 |
| 范例 |  |
| #include <math.h>  main(){  double answer = tan(0.5);  printf(“tan(0.5) = %f\n”,answer);  } | |
| 执行 |  |
| tan(0.5) = 0.546302 | |

## tanh(取双曲线正切函数值)

|  |  |
| --- | --- |
| **tanh(取双曲线正切函数值)** | |
| 相关函数 | cosh,sinh |
| 表头文件 | #include <math.h> |
| 定义函数 | double tanh(double x); |
| 函数说明 | tanh()用来计算参数x的双曲线正切值,然后将结果返回。数学定义式为：sinh(x)/cosh(x)。 |
| 返回值 | 返回参数x的双曲线正切值。 |
| 附加说明 | 使用GCC编译时请加入-lm。 |
| 范例 |  |
| #include <math.h>  main(){  double answer = tanh(0.5);  printf(“ tanh(0.5) = %f\n”,answer);  } | |
| 执行 |  |
| tanh(0.5) = 0.462117 | |

## asin(取反正弦函数值)

|  |  |
| --- | --- |
| **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之PI/2之间的计算结果。 |
| 错误代码 | EDOM参数x超出范围 |
| 附加说明 | 使用GCC编译时请加入 －lm |
| 范例 |  |
| #include <math.h>  main(){  double angle;  angle = asin(0.5);  printf(“angle = %f\n”,angle);  } | |
| 执行 |  |
| angle = 0.523599 | |

## acos(取反余弦函数数值)

|  |  |
| --- | --- |
| **acos(取反余弦函数数值)** | |
| 相关函数 | asin , atan , atan2 , cos , sin , tan |
| 表头文件 | #include <math.h> |
| 定义函数 | double acos(double x); |
| 函数说明 | acos() 用来计算参数x的反余弦值,然后将结果返回。参数x范围为－1至1之间,超过此范围则会失败。 |
| 返回值 | 返回0至 PI之间的计算结果,单位为弧度,在函数库中角度均以弧度来表示。 |
| 错误代码 | EDOM参数x超出范围。 |
| 附加说明 | 使用GCC编译时请加入－lm。 |
| 范例 |  |
| #include <math.h>  main(){  double angle;  angle = acos(0.5);  printf(“angle = %f\n” , angle);  } | |
| 执行 |  |
| angle = 1.047198 | |

## atan(取反正切函数值)

|  |  |
| --- | --- |
| **atan(取反正切函数值)** | |
| 相关函数 | acos,asin,atan2,cos,sin,tan |
| 表头文件 | #include <math.h> |
| 定义函数 | double atan(double x); |
| 函数说明 | atan()用来计算参数x的反正切值,然后将结果返回。 |
| 返回值 | 返回-PI/2至PI/2之间的计算结果。 |
| 附加说明 | 使用GCC编译时请加入-lm |
| 范例 |  |
| #include <math.h>  main(){  double angle;  angle =atan(1);  printf(“angle = %f\n”,angle);  } | |
| 执行 |  |
| angle = 1.570796 | |

## atan2(取得反正切函数值)

|  |  |
| --- | --- |
| **atan2(取得反正切函数值)** | |
| 相关函数 | acos,asin,atan,cos,sin,tan |
| 表头文件 | #include <math.h> |
| 定义函数 | double atan2(double y,double x); |
| 函数说明 | atan2()用来计算参数y/x的反正切值,然后将结果返回。 |
| 返回值 | 返回-PI/2至PI/2之间的计算结果。 |
| 附加说明 | 使用GCC编译时请加入-lm。 |
| 范例 |  |
| #include <math.h>  main(){  double angle;  angle = atan2(1,2);  printf(“ angle = %f\n”,angle);  } | |
| 执行 |  |
| angle = 0.463648 | |

## ceil(取不小于参数的最小整型数)

|  |  |
| --- | --- |
| **ceil(取不小于参数的最小整型数)** | |
| 相关函数 | fabs |
| 表头文件 | #include <math.h> |
| 定义函数 | double ceil(double x); |
| 函数说明 | ceil()会返回不小于参数x的最小整数值,结果以double形态返回。 |
| 返回值 | 返回不小于参数x的最小整数值。 |
| 附加说明 | 使用GCC编译时请加入-lm。 |
| 范例 |  |
| #include <math.h>  main(){  double value[ ]={4.8,1.12,-2.2,0};  int i;  for(i=0;value[i]!=0;i++)  printf(“%f=>%f\n”,value[i],ceil(value[i]));  } | |
| 执行 |  |
| 4.800000=>5.000000  1.120000=>2.000000  -2.200000=>-2.000000 | |

## exp(计算指数)

|  |  |
| --- | --- |
| **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 |  |
| e^10 = 22026.465795 | |

## frexp(将浮点型数分为底数与指数)

|  |  |
| --- | --- |
| **frexp(将浮点型数分为底数与指数)** | |
| 相关函数 | ldexp,modf |
| 表头文件 | #include <math.h> |
| 定义函数 | double frexp( double x, int \*exp); |
| 函数说明 | frexp()用来将参数x的浮点型数切割成底数和指数。底数部分直接返回,指数部分则借参数exp指针返回,将返回值乘以2的exp次方即为x的值。 |
| 返回值 | 返回参数x的底数部分,指数部分则存于exp指针所指的地址。 |
| 附加说明 | 使用GCC编译时请加入-lm。 |
| 范例 |  |
| #include <math.h>  main(){  int exp;  double fraction;  fraction = frexp(1024,&exp);  printf(“exp = %d\n”,exp);  printf(“fraction = %f\n”,fraction);  } | |
| 执行 |  |
| exp = 11  fraction = 0.500000 /\*0.5\*(2^11)=1024\*/ | |

## ldexp(计算2的次方值)

|  |  |
| --- | --- |
| **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 |  |
| 3\*2^(2) = 12.000000 | |

## log(计算以e为底的对数值)

|  |  |
| --- | --- |
| **log(计算以e为底的对数值)** | |
| 相关函数 | exp,log10,pow |
| 表头文件 | #include <math.h> |
| 定义函数 | double log(double x); |
| 函数说明 | log()用来计算以e为底的x对数值,然后将结果返回。 |
| 返回值 | 返回参数x的自然对数值。 |
| 错误代码 | EDOM 参数x为负数  ERANGE 参数x为零值,零的对数值无定义。 |
| 附加说明 | 使用GCC编译时请加入-lm。 |
| 范例 |  |
| #include <math.h>  main(){  double answer;  answer = log(100);  printf(“log(100) = %f\n”,answer);  } | |
| 执行 |  |
| log(100) = 4.605170 | |

## log10(计算以10为底的对数值)

|  |  |
| --- | --- |
| **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 |  |
| log10(100) = 2.000000 | |

## pow(计算次方值)

|  |  |
| --- | --- |
| **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 |  |
| 2^10 = 1024.000000 | |

## sqrt(计算平方根值)

|  |  |
| --- | --- |
| **sqrt(计算平方根值)** | |
| 相关函数 | hypotq |
| 表头文件 | #include <math.h> |
| 定义函数 | double sqrt(double x); |
| 函数说明 | sqrt()用来计算参数x的平方根,然后将结果返回。参数x必须为正数。 |
| 返回值 | 返回参数x的平方根值。 |
| 错误代码 | EDOM 参数x为负数。 |
| 附加说明 | 使用GCC编译时请加入-lm。 |
| 范例 |  |
| /\*计算200的平方根值\*/  #include <math.h>  main(){  double root;  root = sqrt(200);  printf(“answer is %f\n”,root);  } | |
| 执行 |  |
| answer is 14.142136 | |

# 环境变量

## setenv(改变或增加环境变量)

|  |  |
| --- | --- |
| **setenv(改变或增加环境变量)** | |
| 相关函数 | getenv,putenv,unsetenv |
| 表头文件 | #include <stdlib.h> |
| 定义函数 | int setenv(const char \*name,const char \*value,int overwrite); |
| 函数说明 | setenv()用来改变或增加环境变量的内容。参数name为环境变量名称字符串,参数value则为变量内容,参数overwrite用来决定是否要改变已存在的环境变量。如果overwrite不为0,而该环境变量原已有内容,则原内容会被改为参数value所指的变量内容：如果overwrite为0,且该环境变量已有内容,则参数value会被忽略。 |
| 返回值 | 执行成功则返回0,有错误发生时返回-1。 |
| 错误代码 | ENOMEM 内存不足,无法配置新的环境变量空间 |
| 范例 |  |
| #include <stdlib.h>  main(){  char \*p;  if((p=getenv(“USER”)))  printf(“USER =%s\n”,p);  setenv(“USER”,”test”,1);  printf(“USER=%s\n”,getenv(“USEr”));  unsetenv(“USER”);  printf(“USER=%s\n”,getenv(“USER”));  } | |
| 执行 |  |
| USER = root  USER = test  USER =(null) | |

## getenv(取得环境变量内容)

|  |  |
| --- | --- |
| **getenv(取得环境变量内容)** | |
| 相关函数 | putenv,setenv,unsetenv |
| 表头文件 | #include <stdlib.h> |
| 定义函数 | char \*getenv(const char \*name); |
| 函数说明 | getenv()用来取得参数name环境变量的内容。参数name为环境变量的名称,如果该变量存在则会返回指向该内容的指针。  环境变量的格式为name＝value。 |
| 返回值 | 执行成功则返回指向该内容的指针,找不到符合的环境变量名称则返回NULL。 |
| 范例 |  |
| #include <stdlib.h>  mian(){  char \*p;  if((p = getenv(“USER”)))  printf(“USER=%s\n”,p);  } | |
| 执行 |  |
| USER = root | |

## putenv(改变或增加环境变量)

|  |  |
| --- | --- |
| **putenv(改变或增加环境变量)** | |
| 相关函数 | getenv,setenv,unsetenv |
| 表头文件 | #include 4<stdlib.h> |
| 定义函数 | int putenv(const char \*string); |
| 函数说明 | putenv()用来改变或增加环境变量的内容。参数string的格式为name＝value,如果该环境变量原先存在,则变量内容会依参数string改变,否则此参数内容会成为新的环境变量。 |
| 返回值 | 执行成功则返回0,有错误发生则返回-1。 |
| 错误代码 | ENOMEM 内存不足,无法配置新的环境变量空间 |
| 范例 |  |
| #include <stdlib.h>  main(){  char \*p;  if((p = getenv(“USER”)))  printf(“USER =%s\n”,p);  putenv(“USER=test”);  printf(“USER+5s\n”,getenv(“USER”));  } | |
| 执行 |  |
| USER=root  USER=test | |

# 终端控制

## getopt(分析命令行参数)

|  |  |
| --- | --- |
| **getopt(分析命令行参数)** | |
| 相关函数 | getoptlong() |
| 表头文件 | #include <unistd.h> |
| 定义函数 | int getopt(int argc,char \*const argv[ ],const char \*optstring); |
| 函数说明 | getopt()用来分析命令行参数。参数argc和argv是由main()传递的参数个数和内容。参数optstring则代表欲处理的选项字符串。此函数会返回在argv中下一个的选项字母,此字母会对应参数optstring中的字母。如果选项字符串里的字母后接着冒号“：”,则表示还有相关的参数,全域变量optarg即会指向此额外参数。  如果getopt()找不到符合的参数则会印出错信息,并将全域变量optopt设为“？”字符,如果不希望getopt()印出错信息,则只要将全域变量opterr设为0即可。 |
| 返回值 | 如果找到符合的参数则返回此参数字母,如果参数不包含在参数optstring的选项字母则返回“？”字符,分析结束则返回-1。 |
| 范例 |  |
| #include <stdio.h>  #include <unistd.h>  int main(int argc,char \*\*argv){  int ch;  opterr = 0;  while((ch = getopt(argc,argv,”a:bcde”))!= -1)  switch(ch){  case ‘a’:  printf(“option a:’%s’\n”,optarg);  break;  case ‘b’:  printf(“option b :b\n”);  break;  default:  printf(“other option :%c\n”,ch);  }  printf(“optopt +%c\n”,optopt);  } | |
| 执行 |  |
| $./getopt –b  option b:b  $./getopt –c  other option:c  $./getopt –a  other option :?  $./getopt –a12345  option a:’12345’ | |

## select(I/O多工机制)

|  |  |
| --- | --- |
| **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()用来等待文件描述符状态的改变。参数n代表最大的文件描述符加1,参数readfds、writefds和exceptfds称为描述词**组**,是用来回传该描述词的读,写或例外的状况。底下的宏提供了处理这三种描述词组的方式：  FD\_CLR(inr fd,fd\_set\*set)；用来清除描述词组set中相关fd的位  FD\_ISSET(int fd,fd\_set \*set)；用来测试描述词组set中相关fd的位是否为真  FD\_SET(int fd,fd\_set\*set)；用来设置描述词组set中相关fd的位  FD\_ZERO(fd\_set \*set)； 用来清除描述词组set的全部位  参数timeout为结构timeval,用来设置select()的等待时间,其结构定义如下：  struct timeval{  time\_t tv\_sec;  time\_t tv\_usec;  }; |
| 返回值 | 如果参数timeout设为NULL则表示select()没有timeout。 |
| 错误代码 | 执行成功则返回文件描述符状态已改变的个数,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 文件描述符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 |  |
| 下面是常见的程序片段  fs\_set readset；  FD\_ZERO(&readset);  FD\_SET(fd,&readset);  select(fd+1,&readset,NULL,NULL,NULL);  if(FD\_ISSET(fd,readset){……} | |

## isatty(判断文件描述符是否是为终端机)

|  |  |
| --- | --- |
| **isatty(判断文件描述符是否是为终端机)** | |
| 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所代表的文件描述符为一终端机则返回1,否则返回0。 |
| 返回值 | 如果文件为终端机则返回1,否则返回0。 |
| 范例 |  |
| 参考ttyname()。 | |

## ttyname(返回一终端机名称)

|  |  |
| --- | --- |
| **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所代表的文件描述符为一终端机,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(fiel,O\_RDONLY);  printf(“%s”,file);  if(isatty(fd)){  printf(“is a tty.\n”);  printf(“ttyname = %s \n”,ttyname(fd));  } else  printf(“ is not a tty\n”);  close(fd);  } | |
| 执行 |  |
| /dev/tty is a tty  ttyname = /dev/tty | |