**1、 管道概述及相关API应用**

**1.1 管道相关的关键概念**

管道是Linux支持的最初Unix IPC形式之一，具有以下特点：

* 管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道；
* 只能用于父子进程或者兄弟进程之间（具有亲缘关系的进程）；
* 单独构成一种独立的文件系统：管道对于管道两端的进程而言，就是一个文件，但它不是普通的文件，它不属于某种文件系统，而是自立门户，单独构成一种文件系统，并且只存在与内存中。
* 数据的读出和写入：一个进程向管道中写的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。

**1.2管道的创建：**

|  |
| --- |
| #include <unistd.h>  int pipe(int fd[2]) |

该函数创建的管道的两端处于一个进程中间，在实际应用中没有太大意义，因此，一个进程在由pipe()创建管道后，一般再fork一个子进程，然后通过管道实现父子进程间的通信（因此也不难推出，只要两个进程中存在亲缘关系，这里的亲缘关系指的是具有共同的祖先，都可以采用管道方式来进行通信）。

**1.3管道的读写规则：**

管道两端可分别用描述字fd[0]以及fd[1]来描述，需要注意的是，管道的两端是固定了任务的。即一端只能用于读，由描述字fd[0]表示，称其为管道读端；另一端则只能用于写，由描述字fd[1]来表示，称其为管道写端。如果试图从管道写端读取数据，或者向管道读端写入数据都将导致错误发生。一般文件的I/O函数都可以用于管道，如close、read、write等等。

从管道中读取数据：

* 如果管道的写端不存在，则认为已经读到了数据的末尾，读函数返回的读出字节数为0；
* 当管道的写端存在时，如果请求的字节数目大于PIPE\_BUF，则返回管道中现有的数据字节数，如果请求的字节数目不大于PIPE\_BUF，则返回管道中现有数据字节数（此时，管道中数据量小于请求的数据量）；或者返回请求的字节数（此时，管道中数据量不小于请求的数据量）。注：（PIPE\_BUF在include/linux/limits.h中定义，不同的内核版本可能会有所不同。Posix.1要求PIPE\_BUF至少为512字节，red hat 7.2中为4096）。

关于管道的读规则验证：

|  |
| --- |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* readtest.c \*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #include <unistd.h>  #include <sys/types.h>  #include <errno.h>  main()  {  int pipe\_fd[2];  pid\_t pid;  char r\_buf[100];  char w\_buf[4];  char\* p\_wbuf;  int r\_num;  int cmd;    memset(r\_buf,0,sizeof(r\_buf));  memset(w\_buf,0,sizeof(r\_buf));  p\_wbuf=w\_buf;  if(pipe(pipe\_fd)<0)  {  printf("pipe create error\n");  return -1;  }    if((pid=fork())==0)  {  printf("\n");  close(pipe\_fd[1]);  sleep(3);//确保父进程关闭写端  r\_num=read(pipe\_fd[0],r\_buf,100);  printf( "read num is %d the data read from the pipe is %d\n",r\_num,atoi(r\_buf));    close(pipe\_fd[0]);  exit();  }  else if(pid>0)  {  close(pipe\_fd[0]);//read  strcpy(w\_buf,"111");  if(write(pipe\_fd[1],w\_buf,4)!=-1)  printf("parent write over\n");  close(pipe\_fd[1]);//write  printf("parent close fd[1] over\n");  sleep(10);  }  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 程序输出结果：  \* parent write over  \* parent close fd[1] over  \* read num is 4 the data read from the pipe is 111  \* 附加结论：  \* 管道写端关闭后，写入的数据将一直存在，直到读出为止.  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ |

向管道中写入数据：

* 向管道中写入数据时，linux将不保证写入的原子性，管道缓冲区一有空闲区域，写进程就会试图向管道写入数据。如果读进程不读走管道缓冲区中的数据，那么写操作将一直阻塞。   
  注：只有在管道的读端存在时，向管道中写入数据才有意义。否则，向管道中写入数据的进程将收到内核传来的SIFPIPE信号，应用程序可以处理该信号，也可以忽略（默认动作则是应用程序终止）。

对管道的写规则的验证1：写端对读端存在的依赖性

|  |
| --- |
| #include <unistd.h>  #include <sys/types.h>  main()  {  int pipe\_fd[2];  pid\_t pid;  char r\_buf[4];  char\* w\_buf;  int writenum;  int cmd;    memset(r\_buf,0,sizeof(r\_buf));  if(pipe(pipe\_fd)<0)  {  printf("pipe create error\n");  return -1;  }    if((pid=fork())==0)  {  close(pipe\_fd[0]);  close(pipe\_fd[1]);  sleep(10);  exit();  }  else if(pid>0)  {  sleep(1); //等待子进程完成关闭读端的操作  close(pipe\_fd[0]);//write  w\_buf="111";  if((writenum=write(pipe\_fd[1],w\_buf,4))==-1)  printf("write to pipe error\n");  else  printf("the bytes write to pipe is %d \n", writenum);    close(pipe\_fd[1]);  }  } |

则输出结果为： Broken pipe,原因就是该管道以及它的所有fork()产物的读端都已经被关闭。如果在父进程中保留读端，即在写完pipe后，再关闭父进程的读端，也会正常写入pipe，读者可自己验证一下该结论。因此，在向管道写入数据时，至少应该存在某一个进程，其中管道读端没有被关闭，否则就会出现上述错误（管道断裂,进程收到了SIGPIPE信号，默认动作是进程终止）

对管道的写规则的验证2：linux不保证写管道的原子性验证

|  |
| --- |
| #include <unistd.h>  #include <sys/types.h>  #include <errno.h>  main(int argc,char\*\*argv)  {  int pipe\_fd[2];  pid\_t pid;  char r\_buf[4096];  char w\_buf[4096\*2];  int writenum;  int rnum;  memset(r\_buf,0,sizeof(r\_buf));  if(pipe(pipe\_fd)<0)  {  printf("pipe create error\n");  return -1;  }    if((pid=fork())==0)  {  close(pipe\_fd[1]);  while(1)  {  sleep(1);  rnum=read(pipe\_fd[0],r\_buf,1000);  printf("child: readnum is %d\n",rnum);  }  close(pipe\_fd[0]);    exit();  }  else if(pid>0)  {  close(pipe\_fd[0]);//write  memset(r\_buf,0,sizeof(r\_buf));  if((writenum=write(pipe\_fd[1],w\_buf,1024))==-1)  printf("write to pipe error\n");  else  printf("the bytes write to pipe is %d \n", writenum);  writenum=write(pipe\_fd[1],w\_buf,4096);  close(pipe\_fd[1]);  }  }  输出结果：  the bytes write to pipe 1000  the bytes write to pipe 1000 //注意，此行输出说明了写入的非原子性  the bytes write to pipe 1000  the bytes write to pipe 1000  the bytes write to pipe 1000  the bytes write to pipe 120 //注意，此行输出说明了写入的非原子性  the bytes write to pipe 0  the bytes write to pipe 0  ...... |

结论：

写入数目小于4096时写入是非原子的！   
如果把父进程中的两次写入字节数都改为5000，则很容易得出下面结论：   
写入管道的数据量大于4096字节时，缓冲区的空闲空间将被写入数据（补齐），直到写完所有数据为止，如果没有进程读数据，则一直阻塞。

**1.4管道应用实例：**

**实例一：用于shell**

管道可用于输入输出重定向，它将一个命令的输出直接定向到另一个命令的输入。比如，当在某个shell程序（Bourne shell或C shell等）键入who│wc -l后，相应shell程序将创建who以及wc两个进程和这两个进程间的管道。考虑下面的命令行：

$kill -l 运行结果见 [附一](http://www.ibm.com/developerworks/cn/linux/l-ipc/part1/#a)。

$kill -l | grep SIGRTMIN 运行结果如下：

|  |
| --- |
| 30) SIGPWR 31) SIGSYS 32) SIGRTMIN 33) SIGRTMIN+1  34) SIGRTMIN+2 35) SIGRTMIN+3 36) SIGRTMIN+4 37) SIGRTMIN+5  38) SIGRTMIN+6 39) SIGRTMIN+7 40) SIGRTMIN+8 41) SIGRTMIN+9  42) SIGRTMIN+10 43) SIGRTMIN+11 44) SIGRTMIN+12 45) SIGRTMIN+13  46) SIGRTMIN+14 47) SIGRTMIN+15 48) SIGRTMAX-15 49) SIGRTMAX-14 |

**实例二：用于具有亲缘关系的进程间通信**

下面例子给出了管道的具体应用，父进程通过管道发送一些命令给子进程，子进程解析命令，并根据命令作相应处理。

|  |
| --- |
| #include <unistd.h>  #include <sys/types.h>  main()  {  int pipe\_fd[2];  pid\_t pid;  char r\_buf[4];  char\*\* w\_buf[256];  int childexit=0;  int i;  int cmd;    memset(r\_buf,0,sizeof(r\_buf));  if(pipe(pipe\_fd)<0)  {  printf("pipe create error\n");  return -1;  }  if((pid=fork())==0)  //子进程：解析从管道中获取的命令，并作相应的处理  {  printf("\n");  close(pipe\_fd[1]);  sleep(2);    while(!childexit)  {  read(pipe\_fd[0],r\_buf,4);  cmd=atoi(r\_buf);  if(cmd==0)  {  printf("child: receive command from parent over\n now child process exit\n");  childexit=1;  }    else if(handle\_cmd(cmd)!=0)  return;  sleep(1);  }  close(pipe\_fd[0]);  exit();  }  else if(pid>0)  //parent: send commands to child  {  close(pipe\_fd[0]);  w\_buf[0]="003";  w\_buf[1]="005";  w\_buf[2]="777";  w\_buf[3]="000";  for(i=0;i<4;i++)  write(pipe\_fd[1],w\_buf[i],4);  close(pipe\_fd[1]);  }  }  //下面是子进程的命令处理函数（特定于应用）：  int handle\_cmd(int cmd)  {  if((cmd<0)||(cmd>256))  //suppose child only support 256 commands  {  printf("child: invalid command \n");  return -1;  }  printf("child: the cmd from parent is %d\n", cmd);  return 0;  } |

**1.5管道的局限性**

管道的主要局限性正体现在它的特点上：

* 只支持单向数据流；
* 只能用于具有亲缘关系的进程之间；
* 没有名字；
* 管道的缓冲区是有限的（管道制存在于内存中，在管道创建时，为缓冲区分配一个页面大小）；
* 管道所传送的是无格式字节流，这就要求管道的读出方和写入方必须事先约定好数据的格式，比如多少字节算作一个消息（或命令、或记录）等等；

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part1/#ibm-pcon)

**2、 有名管道概述及相关API应用**

**2.1 有名管道相关的关键概念**

管道应用的一个重大限制是它没有名字，因此，只能用于具有亲缘关系的进程间通信，在有名管道（named pipe或FIFO）提出后，该限制得到了克服。FIFO不同于管道之处在于它提供一个路径名与之关联，以FIFO的文件形式存在于文件系统中。这样，即使与FIFO的创建进程不存在亲缘关系的进程，只要可以访问该路径，就能够彼此通过FIFO相互通信（能够访问该路径的进程以及FIFO的创建进程之间），因此，通过FIFO不相关的进程也能交换数据。值得注意的是，FIFO严格遵循先进先出（first in first out），对管道及FIFO的读总是从开始处返回数据，对它们的写则把数据添加到末尾。它们不支持诸如lseek()等文件定位操作。

**2.2有名管道的创建**

|  |
| --- |
| #include <sys/types.h>  #include <sys/stat.h>  int mkfifo(const char \* pathname, mode\_t mode) |

该函数的第一个参数是一个普通的路径名，也就是创建后FIFO的名字。第二个参数与打开普通文件的open()函数中的mode 参数相同。 如果mkfifo的第一个参数是一个已经存在的路径名时，会返回EEXIST错误，所以一般典型的调用代码首先会检查是否返回该错误，如果确实返回该错误，那么只要调用打开FIFO的函数就可以了。一般文件的I/O函数都可以用于FIFO，如close、read、write等等。

**2.3有名管道的打开规则**

有名管道比管道多了一个打开操作：open。

FIFO的打开规则：

如果当前打开操作是为读而打开FIFO时，若已经有相应进程为写而打开该FIFO，则当前打开操作将成功返回；否则，可能阻塞直到有相应进程为写而打开该FIFO（当前打开操作设置了阻塞标志）；或者，成功返回（当前打开操作没有设置阻塞标志）。

如果当前打开操作是为写而打开FIFO时，如果已经有相应进程为读而打开该FIFO，则当前打开操作将成功返回；否则，可能阻塞直到有相应进程为读而打开该FIFO（当前打开操作设置了阻塞标志）；或者，返回ENXIO错误（当前打开操作没有设置阻塞标志）。

对打开规则的验证参见 [附2](http://www.ibm.com/developerworks/cn/linux/l-ipc/part1/#b)。

**2.4有名管道的读写规则**

从FIFO中读取数据：

约定：如果一个进程为了从FIFO中读取数据而阻塞打开FIFO，那么称该进程内的读操作为设置了阻塞标志的读操作。

* 如果有进程写打开FIFO，且当前FIFO内没有数据，则对于设置了阻塞标志的读操作来说，将一直阻塞。对于没有设置阻塞标志读操作来说则返回-1，当前errno值为EAGAIN，提醒以后再试。
* 对于设置了阻塞标志的读操作说，造成阻塞的原因有两种：当前FIFO内有数据，但有其它进程在读这些数据；另外就是FIFO内没有数据。解阻塞的原因则是FIFO中有新的数据写入，不论信写入数据量的大小，也不论读操作请求多少数据量。
* 读打开的阻塞标志只对本进程第一个读操作施加作用，如果本进程内有多个读操作序列，则在第一个读操作被唤醒并完成读操作后，其它将要执行的读操作将不再阻塞，即使在执行读操作时，FIFO中没有数据也一样（此时，读操作返回0）。
* 如果没有进程写打开FIFO，则设置了阻塞标志的读操作会阻塞。

注：如果FIFO中有数据，则设置了阻塞标志的读操作不会因为FIFO中的字节数小于请求读的字节数而阻塞，此时，读操作会返回FIFO中现有的数据量。

向FIFO中写入数据：

约定：如果一个进程为了向FIFO中写入数据而阻塞打开FIFO，那么称该进程内的写操作为设置了阻塞标志的写操作。

对于设置了阻塞标志的写操作：

* 当要写入的数据量不大于PIPE\_BUF时，linux将保证写入的原子性。如果此时管道空闲缓冲区不足以容纳要写入的字节数，则进入睡眠，直到当缓冲区中能够容纳要写入的字节数时，才开始进行一次性写操作。
* 当要写入的数据量大于PIPE\_BUF时，linux将不再保证写入的原子性。FIFO缓冲区一有空闲区域，写进程就会试图向管道写入数据，写操作在写完所有请求写的数据后返回。

对于没有设置阻塞标志的写操作：

* 当要写入的数据量大于PIPE\_BUF时，linux将不再保证写入的原子性。在写满所有FIFO空闲缓冲区后，写操作返回。
* 当要写入的数据量不大于PIPE\_BUF时，linux将保证写入的原子性。如果当前FIFO空闲缓冲区能够容纳请求写入的字节数，写完后成功返回；如果当前FIFO空闲缓冲区不能够容纳请求写入的字节数，则返回EAGAIN错误，提醒以后再写；

对FIFO读写规则的验证：

下面提供了两个对FIFO的读写程序，适当调节程序中的很少地方或者程序的命令行参数就可以对各种FIFO读写规则进行验证。

**程序1：写FIFO的程序**

|  |
| --- |
| #include <sys/types.h>  #include <sys/stat.h>  #include <errno.h>  #include <fcntl.h>  #define FIFO\_SERVER "/tmp/fifoserver"  main(int argc,char\*\* argv)  //参数为即将写入的字节数  {  int fd;  char w\_buf[4096\*2];  int real\_wnum;  memset(w\_buf,0,4096\*2);  if((mkfifo(FIFO\_SERVER,O\_CREAT|O\_EXCL)<0)&&(errno!=EEXIST))  printf("cannot create fifoserver\n");  if(fd==-1)  if(errno==ENXIO)  printf("open error; no reading process\n");    fd=open(FIFO\_SERVER,O\_WRONLY|O\_NONBLOCK,0);  //设置非阻塞标志  //fd=open(FIFO\_SERVER,O\_WRONLY,0);  //设置阻塞标志  real\_wnum=write(fd,w\_buf,2048);  if(real\_wnum==-1)  {  if(errno==EAGAIN)  printf("write to fifo error; try later\n");  }  else  printf("real write num is %d\n",real\_wnum);  real\_wnum=write(fd,w\_buf,5000);  //5000用于测试写入字节大于4096时的非原子性  //real\_wnum=write(fd,w\_buf,4096);  //4096用于测试写入字节不大于4096时的原子性    if(real\_wnum==-1)  if(errno==EAGAIN)  printf("try later\n");  } |

**程序2：与程序1一起测试写FIFO的规则，第一个命令行参数是请求从FIFO读出的字节数**

|  |
| --- |
| #include <sys/types.h>  #include <sys/stat.h>  #include <errno.h>  #include <fcntl.h>  #define FIFO\_SERVER "/tmp/fifoserver"  main(int argc,char\*\* argv)  {  char r\_buf[4096\*2];  int fd;  int r\_size;  int ret\_size;  r\_size=atoi(argv[1]);  printf("requred real read bytes %d\n",r\_size);  memset(r\_buf,0,sizeof(r\_buf));  fd=open(FIFO\_SERVER,O\_RDONLY|O\_NONBLOCK,0);  //fd=open(FIFO\_SERVER,O\_RDONLY,0);  //在此处可以把读程序编译成两个不同版本：阻塞版本及非阻塞版本  if(fd==-1)  {  printf("open %s for read error\n");  exit();  }  while(1)  {    memset(r\_buf,0,sizeof(r\_buf));  ret\_size=read(fd,r\_buf,r\_size);  if(ret\_size==-1)  if(errno==EAGAIN)  printf("no data avlaible\n");  printf("real read bytes %d\n",ret\_size);  sleep(1);  }  pause();  unlink(FIFO\_SERVER);  } |

程序应用说明：

把读程序编译成两个不同版本：

* 阻塞读版本:br
* 以及非阻塞读版本nbr

把写程序编译成两个四个版本：

* 非阻塞且请求写的字节数大于PIPE\_BUF版本：nbwg
* 非阻塞且请求写的字节数不大于PIPE\_BUF版本：版本nbw
* 阻塞且请求写的字节数大于PIPE\_BUF版本：bwg
* 阻塞且请求写的字节数不大于PIPE\_BUF版本：版本bw

下面将使用br、nbr、w代替相应程序中的阻塞读、非阻塞读

验证阻塞写操作：

1. 当请求写入的数据量大于PIPE\_BUF时的非原子性：
   * nbr 1000
   * bwg
2. 当请求写入的数据量不大于PIPE\_BUF时的原子性：
   * nbr 1000
   * bw

验证非阻塞写操作：

1. 当请求写入的数据量大于PIPE\_BUF时的非原子性：
   * nbr 1000
   * nbwg
2. 请求写入的数据量不大于PIPE\_BUF时的原子性：
   * nbr 1000
   * nbw

不管写打开的阻塞标志是否设置，在请求写入的字节数大于4096时，都不保证写入的原子性。但二者有本质区别：

对于阻塞写来说，写操作在写满FIFO的空闲区域后，会一直等待，直到写完所有数据为止，请求写入的数据最终都会写入FIFO；

而非阻塞写则在写满FIFO的空闲区域后，就返回(实际写入的字节数)，所以有些数据最终不能够写入。

对于读操作的验证则比较简单，不再讨论。

**2.5有名管道应用实例**

在验证了相应的读写规则后，应用实例似乎就没有必要了。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part1/#ibm-pcon)

**小结：**

管道常用于两个方面：（1）在shell中时常会用到管道（作为输入输入的重定向），在这种应用方式下，管道的创建对于用户来说是透明的；（2）用于具有亲缘关系的进程间通信，用户自己创建管道，并完成读写操作。

FIFO可以说是管道的推广，克服了管道无名字的限制，使得无亲缘关系的进程同样可以采用先进先出的通信机制进行通信。

管道和FIFO的数据是字节流，应用程序之间必须事先确定特定的传输"协议"，采用传播具有特定意义的消息。

要灵活应用管道及FIFO，理解它们的读写规则是关键。

附1：kill -l 的运行结果，显示了当前系统支持的所有信号：

|  |
| --- |
| 1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL  5) SIGTRAP 6) SIGABRT 7) SIGBUS 8) SIGFPE  9) SIGKILL 10) SIGUSR1 11) SIGSEGV 12) SIGUSR2  13) SIGPIPE 14) SIGALRM 15) SIGTERM 17) SIGCHLD  18) SIGCONT 19) SIGSTOP 20) SIGTSTP 21) SIGTTIN  22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ  26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO  30) SIGPWR 31) SIGSYS 32) SIGRTMIN 33) SIGRTMIN+1  34) SIGRTMIN+2 35) SIGRTMIN+3 36) SIGRTMIN+4 37) SIGRTMIN+5  38) SIGRTMIN+6 39) SIGRTMIN+7 40) SIGRTMIN+8 41) SIGRTMIN+9  42) SIGRTMIN+10 43) SIGRTMIN+11 44) SIGRTMIN+12 45) SIGRTMIN+13  46) SIGRTMIN+14 47) SIGRTMIN+15 48) SIGRTMAX-15 49) SIGRTMAX-14  50) SIGRTMAX-13 51) SIGRTMAX-12 52) SIGRTMAX-11 53) SIGRTMAX-10  54) SIGRTMAX-9 55) SIGRTMAX-8 56) SIGRTMAX-7 57) SIGRTMAX-6  58) SIGRTMAX-5 59) SIGRTMAX-4 60) SIGRTMAX-3 61) SIGRTMAX-2  62) SIGRTMAX-1 63) SIGRTMAX |

除了在此处用来说明管道应用外，接下来的专题还要对这些信号分类讨论。

附2：对FIFO打开规则的验证（主要验证写打开对读打开的依赖性）

|  |
| --- |
| #include <sys/types.h>  #include <sys/stat.h>  #include <errno.h>  #include <fcntl.h>  #define FIFO\_SERVER "/tmp/fifoserver"  int handle\_client(char\*);  main(int argc,char\*\* argv)  {  int r\_rd;  int w\_fd;  pid\_t pid;  if((mkfifo(FIFO\_SERVER,O\_CREAT|O\_EXCL)<0)&&(errno!=EEXIST))  printf("cannot create fifoserver\n");  handle\_client(FIFO\_SERVER);    }  int handle\_client(char\* arg)  {  int ret;  ret=w\_open(arg);  switch(ret)  {  case 0:  {  printf("open %s error\n",arg);  printf("no process has the fifo open for reading\n");  return -1;  }  case -1:  {  printf("something wrong with open the fifo except for ENXIO");  return -1;  }  case 1:  {  printf("open server ok\n");  return 1;  }  default:  {  printf("w\_no\_r return ----\n");  return 0;  }  }  unlink(FIFO\_SERVER);  }  int w\_open(char\*arg)  //0 open error for no reading  //-1 open error for other reasons  //1 open ok  {  if(open(arg,O\_WRONLY|O\_NONBLOCK,0)==-1)  { if(errno==ENXIO)  {  return 0;  }  else  return -1;  }  return 1;    } |

**参考资料**

* UNIX网络编程第二卷：进程间通信，作者：W.Richard Stevens，译者：杨继张，清华大学出版社。丰富的UNIX进程间通信实例及分析，对Linux环境下的程序开发有极大的启发意义。
* linux内核源代码情景分析（上、下），毛德操、胡希明著，浙江大学出版社，当要验证某个结论、想法时，最好的参考资料；
* UNIX环境高级编程，作者：W.Richard Stevens，译者：尤晋元等，机械工业出版社。具有丰富的编程实例，以及关键函数伴随Unix的发展历程。
* <http://www.linux.org.tw/CLDP/gb/Secure-Programs-HOWTO/x346.html> 点明linux下sigaction的实现基础，linux源码../kernel/signal.c更说明了问题；
* pipe手册，最直接而可靠的参考资料
* fifo手册，最直接而可靠的参考资料

**关于作者**

郑彦兴，男，现攻读国防科大计算机学院网络方向博士学位。您可以通过电子邮件 [mlinux@163.com](mailto:mlinux@163.com)和他联系。