消息队列（也叫做报文队列）能够克服早期unix通信机制的一些缺点。作为早期unix通信机制之一的信号能够传送的信息量有限，后来虽然POSIX 1003.1b在信号的实时性方面作了拓广，使得信号在传递信息量方面有了相当程度的改进，但是信号这种通信方式更像"即时"的通信方式，它要求接受信号的进程在某个时间范围内对信号做出反应，因此该信号最多在接受信号进程的生命周期内才有意义，信号所传递的信息是接近于随进程持续的概念（process-persistent），见 [附录 1](http://www.ibm.com/developerworks/cn/linux/l-ipc/part3/#listing1)；管道及有名管道及有名管道则是典型的随进程持续IPC，并且，只能传送无格式的字节流无疑会给应用程序开发带来不便，另外，它的缓冲区大小也受到限制。

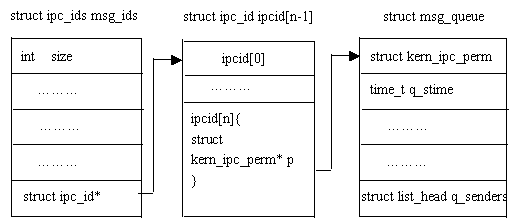
消息队列就是一个消息的链表。可以把消息看作一个记录，具有特定的格式以及特定的优先级。对消息队列有写权限的进程可以向中按照一定的规则添加新消息；对消息队列有读权限的进程则可以从消息队列中读走消息。消息队列是随内核持续的（参见 [附录 1](http://www.ibm.com/developerworks/cn/linux/l-ipc/part3/#listing1)）。

目前主要有两种类型的消息队列：POSIX消息队列以及系统V消息队列，系统V消息队列目前被大量使用。考虑到程序的可移植性，新开发的应用程序应尽量使用POSIX消息队列。

在本系列专题的序（深刻理解Linux进程间通信（IPC））中，提到对于消息队列、信号灯、以及共享内存区来说，有两个实现版本：POSIX的以及系统V的。Linux内核（内核2.4.18）支持POSIX信号灯、POSIX共享内存区以及POSIX消息队列，但对于主流Linux发行版本之一redhad8.0（内核2.4.18），还没有提供对POSIX进程间通信API的支持，不过应该只是时间上的事。

因此，本文将主要介绍系统V消息队列及其相应API。 **在没有声明的情况下，以下讨论中指的都是系统V消息队列。**

**一、消息队列基本概念**

1. 系统V消息队列是随内核持续的，只有在内核重起或者显示删除一个消息队列时，该消息队列才会真正被删除。因此系统中记录消息队列的数据结构（struct ipc\_ids msg\_ids）位于内核中，系统中的所有消息队列都可以在结构msg\_ids中找到访问入口。
2. 消息队列就是一个消息的链表。每个消息队列都有一个队列头，用结构struct msg\_queue来描述（参见 [附录 2](http://www.ibm.com/developerworks/cn/linux/l-ipc/part3/#listing2)）。队列头中包含了该消息队列的大量信息，包括消息队列键值、用户ID、组ID、消息队列中消息数目等等，甚至记录了最近对消息队列读写进程的ID。读者可以访问这些信息，也可以设置其中的某些信息。
3. 下图说明了内核与消息队列是怎样建立起联系的：   
   其中：struct ipc\_ids msg\_ids是内核中记录消息队列的全局数据结构；struct msg\_queue是每个消息队列的队列头。   
   

从上图可以看出，全局数据结构 struct ipc\_ids msg\_ids 可以访问到每个消息队列头的第一个成员：struct kern\_ipc\_perm；而每个struct kern\_ipc\_perm能够与具体的消息队列对应起来是因为在该结构中，有一个key\_t类型成员key，而key则唯一确定一个消息队列。kern\_ipc\_perm结构如下：

|  |
| --- |
| struct kern\_ipc\_perm{ //内核中记录消息队列的全局数据结构msg\_ids能够访问到该结构；  key\_t key; //该键值则唯一对应一个消息队列  uid\_t uid;  gid\_t gid;  uid\_t cuid;  gid\_t cgid;  mode\_t mode;  unsigned long seq;  } |

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part3/#ibm-pcon)

**二、操作消息队列**

**对消息队列的操作无非有下面三种类型：**

1、 打开或创建消息队列   
消息队列的内核持续性要求每个消息队列都在系统范围内对应唯一的键值，所以，要获得一个消息队列的描述字，只需提供该消息队列的键值即可；

注：消息队列描述字是由在系统范围内唯一的键值生成的，而键值可以看作对应系统内的一条路经。

2、 读写操作

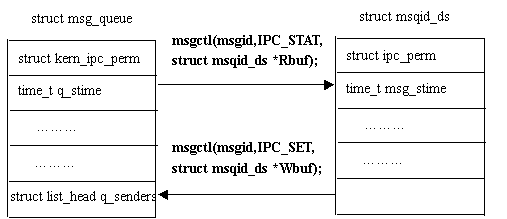
消息读写操作非常简单，对开发人员来说，每个消息都类似如下的数据结构：

|  |
| --- |
| struct msgbuf{  long mtype;  char mtext[1];  }; |

mtype成员代表消息类型，从消息队列中读取消息的一个重要依据就是消息的类型；mtext是消息内容，当然长度不一定为1。因此，对于发送消息来说，首先预置一个msgbuf缓冲区并写入消息类型和内容，调用相应的发送函数即可；对读取消息来说，首先分配这样一个msgbuf缓冲区，然后把消息读入该缓冲区即可。

3、 获得或设置消息队列属性：

消息队列的信息基本上都保存在消息队列头中，因此，可以分配一个类似于消息队列头的结构(struct msqid\_ds，见 [附录 2](http://www.ibm.com/developerworks/cn/linux/l-ipc/part3/#listing2))，来返回消息队列的属性；同样可以设置该数据结构。



**消息队列API**

**1、文件名到键值**

|  |
| --- |
| #include <sys/types.h>  #include <sys/ipc.h>  key\_t ftok (char\*pathname, char proj)； |

它返回与路径pathname相对应的一个键值。该函数不直接对消息队列操作，但在调用ipc(MSGGET,…)或msgget()来获得消息队列描述字前，往往要调用该函数。典型的调用代码是：

|  |
| --- |
| key=ftok(path\_ptr, 'a');  ipc\_id=ipc(MSGGET, (int)key, flags,0,NULL,0);  … |

**2、linux为操作系统V进程间通信的三种方式（消息队列、信号灯、共享内存区）提供了一个统一的用户界面：**   
**int ipc**(unsigned int **call**, int **first**, int **second**, int **third**, void \* **ptr**, long **fifth**);

第一个参数指明对IPC对象的操作方式，对消息队列而言共有四种操作：MSGSND、MSGRCV、MSGGET以及MSGCTL，分别代表向消息队列发送消息、从消息队列读取消息、打开或创建消息队列、控制消息队列；first参数代表唯一的IPC对象；下面将介绍四种操作。

* **int ipc**( **MSGGET, int**first, **int**second, **int**third, **void**\*ptr, **long**fifth);   
  与该操作对应的系统V调用为：int msgget( (key\_t)first，second)。
* **int ipc**( **MSGCTL, int**first, **int**second, **int**third, **void**\*ptr, **long**fifth)   
  与该操作对应的系统V调用为：int msgctl( first，second, (struct msqid\_ds\*) ptr)。
* **int ipc**( **MSGSND, int**first, **int**second, **int**third, **void**\*ptr, **long**fifth);   
  与该操作对应的系统V调用为：int msgsnd( first, (struct msgbuf\*)ptr, second, third)。
* **int ipc**( **MSGRCV, int**first, **int**second, **int**third, **void**\*ptr, **long**fifth);   
  与该操作对应的系统V调用为：int msgrcv( first，(struct msgbuf\*)ptr, second, fifth,third)，

注：本人不主张采用系统调用ipc()，而更倾向于采用系统V或者POSIX进程间通信API。原因如下：

* 虽然该系统调用提供了统一的用户界面，但正是由于这个特性，它的参数几乎不能给出特定的实际意义（如以first、second来命名参数），在一定程度上造成开发不便。
* 正如ipc手册所说的：ipc()是linux所特有的，编写程序时应注意程序的移植性问题；
* 该系统调用的实现不过是把系统V IPC函数进行了封装，没有任何效率上的优势；
* 系统V在IPC方面的API数量不多，形式也较简洁。

**3.系统V消息队列API**   
系统V消息队列API共有四个，使用时需要包括几个头文件：

|  |
| --- |
| #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h> |

**1）int msgget(key\_t key, int msgflg)**

参数key是一个键值，由ftok获得；msgflg参数是一些标志位。该调用返回与健值key相对应的消息队列描述字。

在以下两种情况下，该调用将创建一个新的消息队列：

* 如果没有消息队列与健值key相对应，并且msgflg中包含了IPC\_CREAT标志位；
* key参数为IPC\_PRIVATE；

参数msgflg可以为以下：IPC\_CREAT、IPC\_EXCL、IPC\_NOWAIT或三者的或结果。

**调用返回：**成功返回消息队列描述字，否则返回-1。

注：参数key设置成常数IPC\_PRIVATE并不意味着其他进程不能访问该消息队列，只意味着即将创建新的消息队列。

**2）int msgrcv(int msqid, struct msgbuf \*msgp, int msgsz, long msgtyp, int msgflg);**   
该系统调用从msgid代表的消息队列中读取一个消息，并把消息存储在msgp指向的msgbuf结构中。

msqid为消息队列描述字；消息返回后存储在msgp指向的地址，msgsz指定msgbuf的mtext成员的长度（即消息内容的长度），msgtyp为请求读取的消息类型；读消息标志msgflg可以为以下几个常值的或：

* IPC\_NOWAIT 如果没有满足条件的消息，调用立即返回，此时，errno=ENOMSG
* IPC\_EXCEPT 与msgtyp>0配合使用，返回队列中第一个类型不为msgtyp的消息
* IPC\_NOERROR 如果队列中满足条件的消息内容大于所请求的msgsz字节，则把该消息截断，截断部分将丢失。

msgrcv手册中详细给出了消息类型取不同值时(>0; <0; =0)，调用将返回消息队列中的哪个消息。

msgrcv()解除阻塞的条件有三个：

1. 消息队列中有了满足条件的消息；
2. msqid代表的消息队列被删除；
3. 调用msgrcv（）的进程被信号中断；

**调用返回：**成功返回读出消息的实际字节数，否则返回-1。

**3）int msgsnd(int msqid, struct msgbuf \*msgp, int msgsz, int msgflg);**   
向msgid代表的消息队列发送一个消息，即将发送的消息存储在msgp指向的msgbuf结构中，消息的大小由msgze指定。

对发送消息来说，有意义的msgflg标志为IPC\_NOWAIT，指明在消息队列没有足够空间容纳要发送的消息时，msgsnd是否等待。造成msgsnd()等待的条件有两种：

* 当前消息的大小与当前消息队列中的字节数之和超过了消息队列的总容量；
* 当前消息队列的消息数（单位"个"）不小于消息队列的总容量（单位"字节数"），此时，虽然消息队列中的消息数目很多，但基本上都只有一个字节。

msgsnd()解除阻塞的条件有三个：

1. 不满足上述两个条件，即消息队列中有容纳该消息的空间；
2. msqid代表的消息队列被删除；
3. 调用msgsnd（）的进程被信号中断；

**调用返回：**成功返回0，否则返回-1。

**4）int msgctl(int msqid, int cmd, struct msqid\_ds \*buf);**   
该系统调用对由msqid标识的消息队列执行cmd操作，共有三种cmd操作：IPC\_STAT、IPC\_SET 、IPC\_RMID。

1. IPC\_STAT：该命令用来获取消息队列信息，返回的信息存贮在buf指向的msqid结构中；
2. IPC\_SET：该命令用来设置消息队列的属性，要设置的属性存储在buf指向的msqid结构中；可设置属性包括：msg\_perm.uid、msg\_perm.gid、msg\_perm.mode以及msg\_qbytes，同时，也影响msg\_ctime成员。
3. IPC\_RMID：删除msqid标识的消息队列；

**调用返回：**成功返回0，否则返回-1。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part3/#ibm-pcon)

**三、消息队列的限制**

每个消息队列的容量（所能容纳的字节数）都有限制，该值因系统不同而不同。在后面的应用实例中，输出了redhat 8.0的限制，结果参见 [附录 3](http://www.ibm.com/developerworks/cn/linux/l-ipc/part3/#listing3)。

另一个限制是每个消息队列所能容纳的最大消息数：在redhad 8.0中，该限制是受消息队列容量制约的：消息个数要小于消息队列的容量（字节数）。

注：上述两个限制是针对每个消息队列而言的，系统对消息队列的限制还有系统范围内的最大消息队列个数，以及整个系统范围内的最大消息数。一般来说，实际开发过程中不会超过这个限制。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part3/#ibm-pcon)

**四、消息队列应用实例**

消息队列应用相对较简单，下面实例基本上覆盖了对消息队列的所有操作，同时，程序输出结果有助于加深对前面所讲的某些规则及消息队列限制的理解。

|  |
| --- |
| #include <sys/types.h>  #include <sys/msg.h>  #include <unistd.h>  void msg\_stat(int,struct msqid\_ds );  main()  {  int gflags,sflags,rflags;  key\_t key;  int msgid;  int reval;  struct msgsbuf{  int mtype;  char mtext[1];  }msg\_sbuf;  struct msgmbuf  {  int mtype;  char mtext[10];  }msg\_rbuf;  struct msqid\_ds msg\_ginfo,msg\_sinfo;  char\* msgpath="/unix/msgqueue";  key=ftok(msgpath,'a');  gflags=IPC\_CREAT|IPC\_EXCL;  msgid=msgget(key,gflags|00666);  if(msgid==-1)  {  printf("msg create error\n");  return;  }  //创建一个消息队列后，输出消息队列缺省属性  msg\_stat(msgid,msg\_ginfo);  sflags=IPC\_NOWAIT;  msg\_sbuf.mtype=10;  msg\_sbuf.mtext[0]='a';  reval=msgsnd(msgid,&msg\_sbuf,sizeof(msg\_sbuf.mtext),sflags);  if(reval==-1)  {  printf("message send error\n");  }  //发送一个消息后，输出消息队列属性  msg\_stat(msgid,msg\_ginfo);  rflags=IPC\_NOWAIT|MSG\_NOERROR;  reval=msgrcv(msgid,&msg\_rbuf,4,10,rflags);  if(reval==-1)  printf("read msg error\n");  else  printf("read from msg queue %d bytes\n",reval);  //从消息队列中读出消息后，输出消息队列属性  msg\_stat(msgid,msg\_ginfo);  msg\_sinfo.msg\_perm.uid=8;//just a try  msg\_sinfo.msg\_perm.gid=8;//  msg\_sinfo.msg\_qbytes=16388;  //此处验证超级用户可以更改消息队列的缺省msg\_qbytes  //注意这里设置的值大于缺省值  reval=msgctl(msgid,IPC\_SET,&msg\_sinfo);  if(reval==-1)  {  printf("msg set info error\n");  return;  }  msg\_stat(msgid,msg\_ginfo);  //验证设置消息队列属性  reval=msgctl(msgid,IPC\_RMID,NULL);//删除消息队列  if(reval==-1)  {  printf("unlink msg queue error\n");  return;  }  }  void msg\_stat(int msgid,struct msqid\_ds msg\_info)  {  int reval;  sleep(1);//只是为了后面输出时间的方便  reval=msgctl(msgid,IPC\_STAT,&msg\_info);  if(reval==-1)  {  printf("get msg info error\n");  return;  }  printf("\n");  printf("current number of bytes on queue is %d\n",msg\_info.msg\_cbytes);  printf("number of messages in queue is %d\n",msg\_info.msg\_qnum);  printf("max number of bytes on queue is %d\n",msg\_info.msg\_qbytes);  //每个消息队列的容量（字节数）都有限制MSGMNB，值的大小因系统而异。在创建新的消息队列时，//msg\_qbytes的缺省值就是MSGMNB  printf("pid of last msgsnd is %d\n",msg\_info.msg\_lspid);  printf("pid of last msgrcv is %d\n",msg\_info.msg\_lrpid);  printf("last msgsnd time is %s", ctime(&(msg\_info.msg\_stime)));  printf("last msgrcv time is %s", ctime(&(msg\_info.msg\_rtime)));  printf("last change time is %s", ctime(&(msg\_info.msg\_ctime)));  printf("msg uid is %d\n",msg\_info.msg\_perm.uid);  printf("msg gid is %d\n",msg\_info.msg\_perm.gid);  } |

程序输出结果见 [附录 3](http://www.ibm.com/developerworks/cn/linux/l-ipc/part3/#listing3)。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part3/#ibm-pcon)

**小结：**

消息队列与管道以及有名管道相比，具有更大的灵活性，首先，它提供有格式字节流，有利于减少开发人员的工作量；其次，消息具有类型，在实际应用中，可作为优先级使用。这两点是管道以及有名管道所不能比的。同样，消息队列可以在几个进程间复用，而不管这几个进程是否具有亲缘关系，这一点与有名管道很相似；但消息队列是随内核持续的，与有名管道（随进程持续）相比，生命力更强，应用空间更大。

**附录 1：**在参考文献[1]中，给出了IPC随进程持续、随内核持续以及随文件系统持续的定义：

1. 随进程持续：IPC一直存在到打开IPC对象的最后一个进程关闭该对象为止。如管道和有名管道；
2. 随内核持续：IPC一直持续到内核重新自举或者显示删除该对象为止。如消息队列、信号灯以及共享内存等；
3. 随文件系统持续：IPC一直持续到显示删除该对象为止。

**附录 2：**  
结构msg\_queue用来描述消息队列头，存在于系统空间：

|  |
| --- |
| struct msg\_queue {  struct kern\_ipc\_perm q\_perm;  time\_t q\_stime; /\* last msgsnd time \*/  time\_t q\_rtime; /\* last msgrcv time \*/  time\_t q\_ctime; /\* last change time \*/  unsigned long q\_cbytes; /\* current number of bytes on queue \*/  unsigned long q\_qnum; /\* number of messages in queue \*/  unsigned long q\_qbytes; /\* max number of bytes on queue \*/  pid\_t q\_lspid; /\* pid of last msgsnd \*/  pid\_t q\_lrpid; /\* last receive pid \*/  struct list\_head q\_messages;  struct list\_head q\_receivers;  struct list\_head q\_senders;  }; |

结构msqid\_ds用来设置或返回消息队列的信息，存在于用户空间；

|  |
| --- |
| struct msqid\_ds {  struct ipc\_perm msg\_perm;  struct msg \*msg\_first; /\* first message on queue,unused \*/  struct msg \*msg\_last; /\* last message in queue,unused \*/  \_\_kernel\_time\_t msg\_stime; /\* last msgsnd time \*/  \_\_kernel\_time\_t msg\_rtime; /\* last msgrcv time \*/  \_\_kernel\_time\_t msg\_ctime; /\* last change time \*/  unsigned long msg\_lcbytes; /\* Reuse junk fields for 32 bit \*/  unsigned long msg\_lqbytes; /\* ditto \*/  unsigned short msg\_cbytes; /\* current number of bytes on queue \*/  unsigned short msg\_qnum; /\* number of messages in queue \*/  unsigned short msg\_qbytes; /\* max number of bytes on queue \*/  \_\_kernel\_ipc\_pid\_t msg\_lspid; /\* pid of last msgsnd \*/  \_\_kernel\_ipc\_pid\_t msg\_lrpid; /\* last receive pid \*/  }; |

//可以看出上述两个结构很相似。

**附录 3：**消息队列实例输出结果：

|  |
| --- |
| current number of bytes on queue is 0  number of messages in queue is 0  max number of bytes on queue is 16384  pid of last msgsnd is 0  pid of last msgrcv is 0  last msgsnd time is Thu Jan 1 08:00:00 1970  last msgrcv time is Thu Jan 1 08:00:00 1970  last change time is Sun Dec 29 18:28:20 2002  msg uid is 0  msg gid is 0  //上面刚刚创建一个新消息队列时的输出  current number of bytes on queue is 1  number of messages in queue is 1  max number of bytes on queue is 16384  pid of last msgsnd is 2510  pid of last msgrcv is 0  last msgsnd time is Sun Dec 29 18:28:21 2002  last msgrcv time is Thu Jan 1 08:00:00 1970  last change time is Sun Dec 29 18:28:20 2002  msg uid is 0  msg gid is 0  read from msg queue 1 bytes  //实际读出的字节数  current number of bytes on queue is 0  number of messages in queue is 0  max number of bytes on queue is 16384 //每个消息队列最大容量（字节数）  pid of last msgsnd is 2510  pid of last msgrcv is 2510  last msgsnd time is Sun Dec 29 18:28:21 2002  last msgrcv time is Sun Dec 29 18:28:22 2002  last change time is Sun Dec 29 18:28:20 2002  msg uid is 0  msg gid is 0  current number of bytes on queue is 0  number of messages in queue is 0  max number of bytes on queue is 16388 //可看出超级用户可修改消息队列最大容量  pid of last msgsnd is 2510  pid of last msgrcv is 2510 //对操作消息队列进程的跟踪  last msgsnd time is Sun Dec 29 18:28:21 2002  last msgrcv time is Sun Dec 29 18:28:22 2002  last change time is Sun Dec 29 18:28:23 2002 //msgctl()调用对msg\_ctime有影响  msg uid is 8  msg gid is 8 |

**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。对POSIX以及系统V消息队列都有阐述，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），毛德操、胡希明著，浙江大学出版社，给出了系统V消息队列相关的源代码分析。
* <http://www.fanqiang.com/a4/b2/20010508/113315.html>，主要阐述linux下对文件的操作，详细介绍了对文件的存取权限位，对IPC对象的存取权限同样具有很好的借鉴意义。
* msgget、msgsnd、msgrcv、msgctl手册