目录

[Socket通信机制 1](#_Toc492901072)

[socket套接字： 1](#_Toc492901073)

[套接字描述符： 1](#_Toc492901074)

[文件描述符和文件指针的区别： 2](#_Toc492901075)

[基本的SOCKET接口函数 3](#_Toc492901076)

[socket()函数 4](#_Toc492901077)

[bind()函数 4](#_Toc492901078)

[网络字节序与主机字节序 5](#_Toc492901079)

[listen()、connect()函数 6](#_Toc492901080)

[accept()函数 6](#_Toc492901081)

[read()、write()等函数 7](#_Toc492901082)

[close()函数 8](#_Toc492901083)

[Netlink机制及其关键技术 9](#_Toc492901084)

[3.1 Netlink机制 9](#_Toc492901085)

[3.2 Netlink优点 9](#_Toc492901086)

[建立Netlink会话过程如下： 10](#_Toc492901087)

[其他相关说明 11](#_Toc492901088)

[参考资料 20](#_Toc492901089)

# Socket通信机制

## ****socket****套接字：

     socket起源于Unix，而Unix/[Linux](http://lib.csdn.net/base/linux)基本哲学之一就是“一切皆文件”，都可以用“打开open –> 读写write/read –> 关闭close”模式来操作。Socket就是该模式的一个实现，socket即是一种特殊的文件，一些socket函数就是对其进行的操作（读/写IO、打开、关闭）.  
     说白了Socket是应用层与TCP/IP协议族通信的中间软件抽象层，**它是一组接口**。在设计模式中，Socket其实就是一个门面模式，它把复杂的TCP/IP协议族隐藏在Socket接口后面，对用户来说，一组简单的接口就是全部，让Socket去组织数据，以符合指定的协议。

       注意：其实socket也没有层的概念，它只是一个facade设计模式的应用，让编程变的更简单。是一个软件抽象层。在网络编程中，我们大量用的都是通过socket实现的。

## ****套接字描述符：****

     其实就是一个整数，我们最熟悉的句柄是0、1、2三个，0是标准输入，1是标准输出，2是标准错误输出。0、1、2是整数表示的，对应的FILE \*结构的表示就是stdin、stdout、stderr

套接字API最初是作为UNIX操作系统的一部分而开发的，所以套接字API与系统的其他I/O设备集成在一起。特别是，当应用程序要为因特网通信而创建一个套接字（socket）时，操作系统就返回一个小整数作为描述符（descriptor）来标识这个套接字。然后，应用程序以该描述符作为传递参数，通过调用函数来完成某种操作（例如通过网络传送数据或接收输入的数据）。

在许多操作系统中，套接字描述符和其他I/O描述符是集成在一起的，所以应用程序可以对文件进行套接字I/O或I/O读/写操作。

当应用程序要创建一个套接字时，操作系统就返回一个小整数作为描述符，应用程序则使用这个描述符来引用该套接字需要I/O请求的应用程序请求操作系统打开一个文件。操作系统就创建一个文件描述符提供给应用程序访问文件。从应用程序的角度看，文件描述符是一个整数，应用程序可以用它来读写文件。下图显示，操作系统如何把文件描述符实现为一个指针数组，这些指针指向内部[数据结构](http://lib.csdn.net/base/datastructure)。



     对于每个程序系统都有一张单独的表。精确地讲，系统为每个运行的进程维护一张单独的文件描述符表。当进程打开一个文件时，系统把一个指向此文件内部数据结构的指针写入文件描述符表，并把该表的索引值返回给调用者 。应用程序只需记住这个描述符，并在以后操作该文件时使用它。操作系统把该描述符作为索引访问进程描述符表，通过指针找到保存该文件所有的信息的数据结构。

**针对套接字的系统数据结构：**

   1）、套接字API里有个函数socket，它就是用来创建一个套接字。套接字设计的总体思路是，单个系统调用就可以创建任何套接字，因为套接字是相当笼统的。一旦套接字创建后，应用程序还需要调用其他函数来指定具体细节。例如调用socket将创建一个新的描述符条目：



   2）、虽然套接字的内部数据结构包含很多字段，但是系统创建套接字后，大多数字字段没有填写。应用程序创建套接字后在该套接字可以使用之前，必须调用其他的过程来填充这些字段。

## ****文件描述符和文件指针的区别：****

**文件描述符：**在linux系统中打开文件就会获得文件描述符，它是个很小的正整数。每个进程在PCB（Process Control Block）中保存着一份文件描述符表，文件描述符就是这个表的索引，每个表项都有一个指向已打开文件的指针。

**文件指针：**[C语言](http://lib.csdn.net/base/c)中使用文件指针做为I/O的句柄。文件指针指向进程用户区中的一个被称为FILE结构的数据结构。FILE结构包括一个缓冲区和一个文件描述符。而文件描述符是文件描述符表的一个索引，因此从某种意义上说文件指针就是句柄的句柄（在Windows系统上，文件描述符被称作文件句柄）。

详细内容请看[linux文件系统](http://blog.csdn.net/hguisu/article/details/6122513#t7)：<http://blog.csdn.net/hguisu/article/details/6122513#t7>

## 基本的SOCKET接口函数

在生活中，A要电话给B，A拨号，B听到电话铃声后提起电话，这时A和B就建立起了连接，A和B就可以讲话了。等交流结束，挂断电话结束此次交谈。  打电话很简单解释了这工作原理：“open—write/read—close”模式。



    服务器端先初始化Socket，然后与端口绑定(bind)，对端口进行监听(listen)，调用accept阻塞，等待客户端连接。在这时如果有个客户端初始化一个Socket，然后连接服务器(connect)，如果连接成功，这时客户端与服务器端的连接就建立了。客户端发送数据请求，服务器端接收请求并处理请求，然后把回应数据发送给客户端，客户端读取数据，最后关闭连接，一次交互结束。

**这些接口的实现都是内核来完成。具体如何实现，可以看看linux的内核**

## socket()函数

int **socket**(int protofamily, int type, int protocol);//返回sockfd

  sockfd是描述符。

  socket函数对应于普通文件的打开操作。普通文件的打开操作返回一个文件描述字，而**socket()**用于创建一个socket描述符（socket descriptor），它唯一标识一个socket。这个socket描述字跟文件描述字一样，后续的操作都有用到它，把它作为参数，通过它来进行一些读写操作。

      正如可以给fopen的传入不同参数值，以打开不同的文件。创建socket的时候，也可以指定不同的参数创建不同的socket描述符，socket函数的三个参数分别为：

protofamily：即协议域，又称为协议族（family）。常用的协议族有，AF\_INET(IPV4)、AF\_INET6(IPV6)、AF\_LOCAL（或称AF\_UNIX，Unix域socket）、AF\_ROUTE等等。协议族决定了socket的地址类型，在通信中必须采用对应的地址，如AF\_INET决定了要用ipv4地址（32位的）与端口号（16位的）的组合、AF\_UNIX决定了要用一个绝对路径名作为地址。

type：指定socket类型。常用的socket类型有，SOCK\_STREAM、SOCK\_DGRAM、SOCK\_RAW、SOCK\_PACKET、SOCK\_SEQPACKET等等（socket的类型有哪些？）。

protocol：故名思意，就是指定协议。常用的协议有，IPPROTO\_TCP、IPPTOTO\_UDP、IPPROTO\_SCTP、IPPROTO\_TIPC等，它们分别对应TCP传输协议、UDP传输协议、STCP传输协议、TIPC传输协议（这个协议我将会单独开篇讨论！）。

**注意**：并不是上面的type和protocol可以随意组合的，如SOCK\_STREAM不可以跟IPPROTO\_UDP组合。当protocol为0时，会自动选择type类型对应的默认协议。

当我们调用**socket**创建一个socket时，返回的socket描述字它存在于协议族（address family，AF\_XXX）空间中，但没有一个具体的地址。如果想要给它赋值一个地址，就必须调用bind()函数，否则就当调用connect()、listen()时系统会自动随机分配一个端口。

## bind()函数

正如上面所说bind()函数把一个地址族中的特定地址赋给socket。例如对应AF\_INET、AF\_INET6就是把一个ipv4或ipv6地址和端口号组合赋给socket。

int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

函数的三个参数分别为：

sockfd：即socket描述字，它是通过socket()函数创建了，唯一标识一个socket。bind()函数就是将给这个描述字绑定一个名字。

addr：一个const struct sockaddr \*指针，指向要绑定给sockfd的协议地址。这个地址结构根据地址创建socket时的地址协议族的不同而不同，如ipv4对应的是：

struct sockaddr\_in {

sa\_family\_t sin\_family; /\* address family: AF\_INET \*/

in\_port\_t sin\_port; /\* port in network byte order \*/

struct in\_addr sin\_addr; /\* internet address \*/

};

/\* Internet address. \*/

struct in\_addr {

uint32\_t s\_addr; /\* address in network byte order \*/

};

ipv6对应的是：

struct sockaddr\_in6 {

sa\_family\_t sin6\_family; /\* AF\_INET6 \*/

in\_port\_t sin6\_port; /\* port number \*/

uint32\_t sin6\_flowinfo; /\* IPv6 flow information \*/

struct in6\_addr sin6\_addr; /\* IPv6 address \*/

uint32\_t sin6\_scope\_id; /\* Scope ID (new in 2.4) \*/

};

struct in6\_addr {

unsigned char s6\_addr[16]; /\* IPv6 address \*/

};

Unix域对应的是：

#define UNIX\_PATH\_MAX 108

struct sockaddr\_un {

sa\_family\_t sun\_family; /\* AF\_UNIX \*/

char sun\_path[UNIX\_PATH\_MAX]; /\* pathname \*/

};

addrlen：对应的是地址的长度。

通常服务器在启动的时候都会绑定一个众所周知的地址（如ip地址+端口号），用于提供服务，客户就可以通过它来接连服务器；而客户端就不用指定，有系统自动分配一个端口号和自身的ip地址组合。这就是为什么通常服务器端在listen之前会调用bind()，而客户端就不会调用，而是在connect()时由系统随机生成一个。

## 网络字节序与主机字节序

**主机字节序**就是我们平常说的大端和小端模式：不同的CPU有不同的字节序类型，这些字节序是指整数在内存中保存的顺序，这个叫做主机序。引用标准的Big-Endian和Little-Endian的定义如下：

　　a) Little-Endian就是低位字节排放在内存的低地址端，高位字节排放在内存的高地址端。

　　b) Big-Endian就是高位字节排放在内存的低地址端，低位字节排放在内存的高地址端。

**网络字节序**：4个字节的32 bit值以下面的次序传输：首先是0～7bit，其次8～15bit，然后16～23bit，最后是24~31bit。这种传输次序称作大端字节序。**由于TCP/IP首部中所有的二进制整数在网络中传输时都要求以这种次序，因此它又称作网络字节序。**字节序，顾名思义字节的顺序，就是大于一个字节类型的数据在内存中的存放顺序，一个字节的数据没有顺序的问题了。

所以：在将一个地址绑定到socket的时候，请先将主机字节序转换成为网络字节序，而不要假定主机字节序跟网络字节序一样使用的是Big-Endian。由于这个问题曾引发过血案！公司项目代码中由于存在这个问题，导致了很多莫名其妙的问题，所以请谨记对主机字节序不要做任何假定，务必将其转化为网络字节序再赋给socket。

## listen()、connect()函数

如果作为一个服务器，在调用socket()、bind()之后就会调用listen()来监听这个socket，如果客户端这时调用connect()发出连接请求，服务器端就会接收到这个请求。

int listen(int sockfd, int backlog);

int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

listen函数的第一个参数即为要监听的socket描述字，第二个参数为相应socket可以排队的最大连接个数。socket()函数创建的socket默认是一个主动类型的，listen函数将socket变为被动类型的，等待客户的连接请求。

connect函数的第一个参数即为客户端的socket描述字，第二参数为服务器的socket地址，第三个参数为socket地址的长度。客户端通过调用connect函数来建立与TCP服务器的连接。

## accept()函数

TCP服务器端依次调用socket()、bind()、listen()之后，就会监听指定的socket地址了。TCP客户端依次调用socket()、connect()之后就向TCP服务器发送了一个连接请求。TCP服务器监听到这个请求之后，就会调用accept()函数取接收请求，这样连接就建立好了。之后就可以开始网络I/O操作了，即类同于普通文件的读写I/O操作。

int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen); //返回连接connect\_fd

参数sockfd

参数sockfd就是上面解释中的监听套接字，这个套接字用来监听一个端口，当有一个客户与服务器连接时，它使用这个一个端口号，而此时这个端口号正与这个套接字关联。当然客户不知道套接字这些细节，它只知道一个地址和一个端口号。

参数addr

这是一个结果参数，它用来接受一个返回值，这返回值指定客户端的地址，当然这个地址是通过某个地址结构来描述的，用户应该知道这一个什么样的地址结构。如果对客户的地址不感兴趣，那么可以把这个值设置为NULL。

参数len

如同大家所认为的，它也是结果的参数，用来接受上述addr的结构的大小的，它指明addr结构所占有的字节个数。同样的，它也可以被设置为NULL。

如果accept成功返回，则服务器与客户已经正确建立连接了，此时服务器通过accept返回的套接字来完成与客户的通信。

注意：

      accept默认会阻塞进程，直到有一个客户连接建立后返回，它返回的是一个新可用的套接字，这个套接字是连接套接字。

此时我们需要区分两种套接字，

       监听套接字: 监听套接字正如accept的参数sockfd，它是监听套接字，在调用listen函数之后，是服务器开始调用socket()函数生成的，称为监听socket描述字(监听套接字)

       连接套接字：一个套接字会从主动连接的套接字变身为一个监听套接字；而accept函数返回的是已连接socket描述字(一个连接套接字)，它代表着一个网络已经存在的点点连接。

        一个服务器通常通常仅仅只创建一个监听socket描述字，它在该服务器的生命周期内一直存在。内核为每个由服务器进程接受的客户连接创建了一个已连接socket描述字，当服务器完成了对某个客户的服务，相应的已连接socket描述字就被关闭。

        自然要问的是：为什么要有两种套接字？原因很简单，如果使用一个描述字的话，那么它的功能太多，使得使用很不直观，同时在内核确实产生了一个这样的新的描述字。

连接套接字socketfd\_new 并没有占用新的端口与客户端通信，依然使用的是与监听套接字socketfd一样的端口号

## read()、write()等函数

万事具备只欠东风，至此服务器与客户已经建立好连接了。可以调用网络I/O进行读写操作了，即实现了网咯中不同进程之间的通信！网络I/O操作有下面几组：

read()/write()

recv()/send()

readv()/writev()

recvmsg()/sendmsg()

recvfrom()/sendto()

我推荐使用recvmsg()/sendmsg()函数，这两个函数是最通用的I/O函数，实际上可以把上面的其它函数都替换成这两个函数。它们的声明如下：

#include <unistd.h>

ssize\_t read(int fd, void \*buf, size\_t count);

ssize\_t write(int fd, const void \*buf, size\_t count);

#include <sys/types.h>

#include <sys/socket.h>

ssize\_t send(int sockfd, const void \*buf, size\_t len, int flags);

ssize\_t recv(int sockfd, void \*buf, size\_t len, int flags);

ssize\_t sendto(int sockfd, const void \*buf, size\_t len, int flags,

const struct sockaddr \*dest\_addr, socklen\_t addrlen);

ssize\_t recvfrom(int sockfd, void \*buf, size\_t len, int flags,

struct sockaddr \*src\_addr, socklen\_t \*addrlen);

ssize\_t sendmsg(int sockfd, const struct msghdr \*msg, int flags);

ssize\_t recvmsg(int sockfd, struct msghdr \*msg, int flags);

read函数是负责从fd中读取内容.当读成功时，read返回实际所读的字节数，如果返回的值是0表示已经读到文件的结束了，小于0表示出现了错误。如果错误为EINTR说明读是由中断引起的，如果是ECONNREST表示网络连接出了问题。

write函数将buf中的nbytes字节内容写入文件描述符fd.成功时返回写的字节数。失败时返回-1，并设置errno变量。 在网络程序中，当我们向套接字文件描述符写时有俩种可能。1)write的返回值大于0，表示写了部分或者是全部的数据。2)返回的值小于0，此时出现了错误。我们要根据错误类型来处理。如果错误为EINTR表示在写的时候出现了中断错误。如果为EPIPE表示网络连接出现了问题(对方已经关闭了连接)。

其它的我就不一一介绍这几对I/O函数了，具体参见man文档或者baidu、Google，下面的例子中将使用到send/recv。

## close()函数

在服务器与客户端建立连接之后，会进行一些读写操作，完成了读写操作就要关闭相应的socket描述字，好比操作完打开的文件要调用fclose关闭打开的文件。

#include <unistd.h>

int close(int fd);

close一个TCP socket的缺省行为时把该socket标记为以关闭，然后立即返回到调用进程。该描述字不能再由调用进程使用，也就是说不能再作为read或write的第一个参数。

注意：close操作只是使相应socket描述字的引用计数-1，只有当引用计数为0的时候，才会触发TCP客户端向服务器发送终止连接请求。

## Netlink机制及其关键技术

### 3.1 Netlink机制

Linux操作系统中当CPU处于内核状态时，可以分为有用户上下文的状态和执行硬件、软件中断两种。其中当处于有用户上下文时，由于内核态和用户态的内存映射机制不同，不可直接将本地变量传给用户态的内存区；处于硬件、软件中断时，无法直接向用户内存区传递数据，代码执行不可中断。针对传统的进程间通信机制，他们均无法直接在内核态和用户态之间使用，原因如下表：

|  |  |
| --- | --- |
| 通信方法 | 无法介于内核态与用户态的原因 |
| 管道（不包括命名管道） | 局限于父子进程间的通信。 |
| 消息队列 | 在硬、软中断中无法无阻塞地接收数据。 |
| 信号量 | 无法介于内核态和用户态使用。 |
| 内存共享 | 需要信号量辅助，而信号量又无法使用。 |
| 套接字 | 在硬、软中断中无法无阻塞地接收数据。 |

    解决内核态和用户态通信机制可分为两类：

1. 处于有用户上下文时，可以使用Linux提供的copy\_from\_user()和copy\_to\_user()函数完成，但由于这两个函数可能阻塞，因此不能在硬件、软件的中断过程中使用。
2. 处于硬、软件中断时。

2.1   可以通过Linux内核提供的spinlock自旋锁实现内核线程与中断过程的同步，由于内核线程运行在有上下文的进程中，因此可以在内核线程中使用套接字或消息队列来取得用户空间的数据，然后再将数据通过临界区传递给中断过程.

2.2   通过Netlink机制实现。Netlink 套接字的通信依据是一个对应于进程的标识，一般定为该进程的 ID。Netlink通信最大的特点是对对中断过程的支持，它在内核空间接收用户空间数据时不再需要用户自行启动一个内核线程，而是通过另一个软中断调用用户事先指定的接收函数。通过软中断而不是自行启动内核线程保证了数据传输的及时性。

### 3.2 Netlink优点

Netlink相对于其他的通信机制具有以下优点：

1. 使用Netlink通过自定义一种新的协议并加入协议族即可通过socket API使用Netlink协议完成数据交换，而ioctl和proc文件系统均需要通过程序加入相应的设备或文件。[ioctl与proc都需要依赖与文件的创建]
2. Netlink使用socket缓存队列，是一种异步通信机制，而ioctl是同步通信机制，如果传输的数据量较大，会影响系统性能。
3. Netlink支持多播，属于一个Netlink组的模块和进程都能获得该多播消息。
4. Netlink允许内核发起会话，而ioctl和系统调用只能由用户空间进程发起

## 建立Netlink会话过程如下：



内核使用与标准socket API类似的一套API完成通信过程。首先通过netlink\_kernel\_create()创建套接字，该函数的原型如下：

struct sock \*netlink\_kernel\_create(struct net \*net,  
int unit,unsigned int groups,  
void (\*input)(struct sk\_buff \*skb),  
struct mutex \*cb\_mutex,  
struct module \*module);

其中net参数是网络设备命名空间指针，input函数是netlink socket在接受到消息时调用的回调函数指针，module默认为THIS\_MODULE.

然后用户空间进程使用标准Socket API来创建套接字，将进程ID发送至内核空间，用户空间创建使用socket()创建套接字，该函数的原型如下：

int socket(int domain, int type, int protocol);

其中domain值为PF\_NETLINK，即Netlink使用协议族。protocol为Netlink提供的协议或者是用户自定义的协议，Netlink提供的协议包括NETLINK\_ROUTE, NETLINK\_FIREWALL, NETLINK\_ARPD, NETLINK\_ROUTE6和 NETLINK\_IP6\_FW。

接着使用bind函数绑定。Netlink的bind()函数把一个本地socket地址(源socket地址)与一个打开的socket进行关联。完成绑定，内核空间接收到用户进程ID之后便可以进行通讯。

用户空间进程发送数据使用标准socket API中sendmsg()函数完成，使用时需添加struct msghdr消息和nlmsghdr消息头。一个netlink消息体由nlmsghdr和消息的payload部分组成，输入消息后，内核会进入nlmsghdr指向的缓冲区。

内核空间发送数据使用独立创建的sk\_buff缓冲区，Linux定义了如下宏方便对于缓冲区地址的设置，如下所示：

#define NETLINK\_CB(skb) (\*(struct netlink\_skb\_parms\*)&((skb)->cb))

在对缓冲区设置完成消息地址之后，可以使用netlink\_unicast()来发布单播消息，netlink\_unicast()原型如下：

int netlink\_unicast(struct sock \*sk, struct sk\_buff \*skb, u32 pid, int nonblock);

参数sk为函数netlink\_kernel\_create()返回的socket，参数skb存放消息，它的data字段指向要发送的netlink消息结构，而skb的控制块保存了消息的地址信息，前面的宏NETLINK\_CB(skb)就用于方便设置该控制块，参数pid为接收消息进程的pid，参数nonblock表示该函数是否为非阻塞，如果为1，该函数将在没有接收缓存可利用时立即返回，而如果为0，该函数在没有接收缓存可利用时睡眠。

内核模块或子系统也可以使用函数netlink\_broadcast来发送广播消息：

void netlink\_broadcast(struct sock \*sk, struct sk\_buff \*skb, u32 pid, u32 group, int allocation);

前面的三个参数与netlink\_unicast相同，参数group为接收消息的多播组，该参数的每一个代表一个多播组，因此如果发送给多个多播组，就把该参数设置为多个多播组组ID的位或。参数allocation为内核内存分配类型，一般地为GFP\_ATOMIC或GFP\_KERNEL，GFP\_ATOMIC用于原子的上下文（即不可以睡眠），而GFP\_KERNEL用于非原子上下文。

接收数据时程序需要申请足够大的空间来存储netlink消息头和消息的payload部分。然后使用标准函数接口recvmsg()来接收netlink消息

### 其他相关说明

    Netlink 是一种特殊的 socket，它是 Linux 所特有的，类似于 BSD 中的AF\_ROUTE 但又远比它的功能强大，目前在最新的 Linux 内核（2.6.14）中使用netlink 进行应用与内核通信的应用很多，包括：路由 daemon（NETLINK\_ROUTE），1-wire 子系统（NETLINK\_W1），用户态 socket 协议（NETLINK\_USERSOCK），防火墙（NETLINK\_FIREWALL），socket 监视（NETLINK\_INET\_DIAG），netfilter 日志（NETLINK\_NFLOG），ipsec 安全策略（NETLINK\_XFRM），SELinux 事件通知（NETLINK\_SELINUX），iSCSI 子系统（NETLINK\_ISCSI），进程审计（NETLINK\_AUDIT），转发信息表查询（NETLINK\_FIB\_LOOKUP），netlink connector(NETLINK\_CONNECTOR),netfilter 子系统（NETLINK\_NETFILTER），IPv6 防火墙（NETLINK\_IP6\_FW），DECnet 路由信息（NETLINK\_DNRTMSG），内核事件向用户态通知（NETLINK\_KOBJECT\_UEVENT），通用 netlink（NETLINK\_GENERIC）。

    Netlink 是一种在内核与用户应用间进行双向数据传输的非常好的方式，用户态应用使用标准的 socket API 就可以使用 netlink 提供的强大功能，内核态需要使用专门的内核 API 来使用 netlink。

Netlink 相对于系统调用，ioctl 以及 /proc 文件系统而言具有以下优点：

    1，为了使用 netlink，用户仅需要在 include/linux/netlink.h 中增加一个新类型的 netlink 协议定义即可， 如 #define NETLINK\_MYTEST 17 然后，内核和用户态应用就可以立即通过 socket API 使用该 netlink 协议类型进行数据交换。但系统调用需要增加新的系统调用，ioctl 则需要增加设备或文件， 那需要不少代码，proc 文件系统则需要在 /proc 下添加新的文件或目录，那将使本来就混乱的 /proc 更加混乱。

    2. netlink是一种异步通信机制，在内核与用户态应用之间传递的消息保存在socket缓存队列中，发送消息只是把消息保存在接收者的socket的接收队列，而不需要等待接收者收到消息，但系统调用与 ioctl 则是同步通信机制，如果传递的数据太长，将影响调度粒度。

    3．使用 netlink 的内核部分可以采用模块的方式实现，使用 netlink 的应用部分和内核部分没有编译时依赖，但系统调用就有依赖，而且新的系统调用的实现必须静态地连接到内核中，它无法在模块中实现，使用新系统调用的应用在编译时需要依赖内核。

    4．netlink 支持多播，内核模块或应用可以把消息多播给一个netlink组，属于该neilink 组的任何内核模块或应用都能接收到该消息，内核事件向用户态的通知机制就使用了这一特性，任何对内核事件感兴趣的应用都能收到该子系统发送的内核事件，在后面的文章中将介绍这一机制的使用。

    5．内核可以使用 netlink 首先发起会话，但系统调用和 ioctl 只能由用户应用发起调用。

6．netlink 使用标准的 socket API，因此很容易使用，但系统调用和 ioctl则需要专门的培训才能使用。

**用户态使用 netlink**

   用户态应用使用标准的socket APIs， socket(), bind(), sendmsg(), recvmsg() 和 close() 就能很容易地使用 netlink socket，查询手册页可以了解这些函数的使用细节，本文只是讲解使用 netlink 的用户应该如何使用这些函数。注意，使用 netlink 的应用必须包含头文件 linux/netlink.h。当然 socket 需要的头文件也必不可少，sys/socket.h。

   为了创建一个 netlink socket，用户需要使用如下参数调用 socket():

|  |
| --- |
| socket(AF\_NETLINK, SOCK\_RAW, netlink\_type) |

   第一个参数必须是 AF\_NETLINK 或 PF\_NETLINK，在 Linux 中，它们俩实际为一个东西，它表示要使用netlink，第二个参数必须是SOCK\_RAW或SOCK\_DGRAM，第三个参数指定netlink协议类型，如前面讲的用户自定义协议类型NETLINK\_MYTEST， NETLINK\_GENERIC是一个通用的协议类型，它是专门为用户使用的，因此，用户可以直接使用它，而不必再添加新的协议类型。内核预定义的协议类型有：

#define NETLINK\_ROUTE 0  
#define NETLINK\_W1 1  
#define NETLINK\_USERSOCK 2   
#define NETLINK\_FIREWALL 3  
#define NETLINK\_INET\_DIAG 4  
#define NETLINK\_NFLOG 5  
#define NETLINK\_XFRM 6   
#define NETLINK\_SELINUX 7   
#define NETLINK\_ISCSI 8   
#define NETLINK\_AUDIT 9   
#define NETLINK\_FIB\_LOOKUP 10   
#define NETLINK\_CONNECTOR 11   
#define NETLINK\_NETFILTER 12   
#define NETLINK\_IP6\_FW 13   
#define NETLINK\_DNRTMSG 14   
#define NETLINK\_KOBJECT\_UEVENT 15   
#define NETLINK\_GENERIC 16

   对于每一个netlink协议类型，可以有多达 32多播组，每一个多播组用一个位表示，netlink 的多播特性使得发送消息给同一个组仅需要一次系统调用，因而对于需要多拨消息的应用而言，大大地降低了系统调用的次数。

   函数 bind() 用于把一个打开的 netlink socket 与 netlink 源 socket 地址绑定在一起。netlink socket 的地址结构如下：

struct sockaddr\_nl {  
sa\_family\_t nl\_family;  
unsigned short nl\_pad;  
\_\_u32 nl\_pid;  
\_\_u32 nl\_groups;  
};

   字段 nl\_family 必须设置为 AF\_NETLINK 或着 PF\_NETLINK，字段 nl\_pad 当前没有使用，因此要总是设置为 0，字段 nl\_pid 为接收或发送消息的进程的 ID，如果希望内核处理消息或多播消息，就把该字段设置为 0，否则设置为处理消息的进程 ID。字段 nl\_groups 用于指定多播组，bind 函数用于把调用进程加入到该字段指定的多播组，如果设置为 0，表示调用者不加入任何多播组。

   传递给 bind 函数的地址的 nl\_pid 字段应当设置为本进程的进程 ID，这相当于 netlink socket 的本地地址。但是，对于一个进程的多个线程使用 netlink socket 的情况，字段 nl\_pid 则可以设置为其它的值，如：

|  |
| --- |
| pthread\_self() << 16 | getpid(); |

   因此字段 nl\_pid 实际上未必是进程 ID,它只是用于区分不同的接收者或发送者的一个标识，用户可以根据自己需要设置该字段。函数 bind 的调用方式如下：

|  |
| --- |
| bind(fd, (struct sockaddr\*)&nladdr, sizeof(struct sockaddr\_nl)); |

   fd为前面的 socket 调用返回的文件描述符，参数 nladdr 为 struct sockaddr\_nl 类型的地址。为了发送一个 netlink 消息给内核或其他用户态应用，需要填充目标 netlink socket 地址，此时，字段 nl\_pid 和 nl\_groups 分别表示接收消息者的进程 ID 与多播组。如果字段 nl\_pid 设置为 0，表示消息接收者为内核或多播组，如果 nl\_groups为 0，表示该消息为单播消息，否则表示多播消息。使用函数 sendmsg 发送 netlink 消息时还需要引用结构 struct msghdr、struct nlmsghdr 和 struct iovec，结构 struct msghdr 需如下设置：

struct msghdr msg;  
memset(&msg, 0, sizeof(msg));  
msg.msg\_name = (void \*)&(nladdr);  
msg.msg\_namelen = sizeof(nladdr);

其中 nladdr 为消息接收者的 netlink 地址。

   struct nlmsghdr 为 netlink socket 自己的消息头，这用于多路复用和多路分解 netlink 定义的所有协议类型以及其它一些控制，netlink 的内核实现将利用这个消息头来多路复用和多路分解已经其它的一些控制，因此它也被称为netlink 控制块。因此，应用在发送 netlink 消息时必须提供该消息头。

struct nlmsghdr {  
\_\_u32 nlmsg\_len;  
\_\_u16 nlmsg\_type;  
\_\_u16 nlmsg\_flags;  
\_\_u32 nlmsg\_seq;  
\_\_u32 nlmsg\_pid;  
};

字段 nlmsg\_len 指定消息的总长度，包括紧跟该结构的数据部分长度以及该结构的大小，字段 nlmsg\_type 用于应用内部定义消息的类型，它对 netlink 内核实现是透明的，因此大部分情况下设置为 0，字段 nlmsg\_flags 用于设置消息标志，可用的标志包括：

#define NLM\_F\_REQUEST 1   
#define NLM\_F\_MULTI 2   
#define NLM\_F\_ACK 4   
#define NLM\_F\_ECHO 8   
#define NLM\_F\_ROOT 0x100   
#define NLM\_F\_MATCH 0x200   
#define NLM\_F\_ATOMIC 0x400   
#define NLM\_F\_DUMP (NLM\_F\_ROOT|NLM\_F\_MATCH)   
#define NLM\_F\_REPLACE 0x100   
#define NLM\_F\_EXCL 0x200   
#define NLM\_F\_CREATE 0x400   
#define NLM\_F\_APPEND 0x800

标志NLM\_F\_REQUEST用于表示消息是一个请求，所有应用首先发起的消息都应设置该标志。

标志NLM\_F\_MULTI 用于指示该消息是一个多部分消息的一部分，后续的消息可以通过宏NLMSG\_NEXT来获得。

宏NLM\_F\_ACK表示该消息是前一个请求消息的响应，顺序号与进程ID可以把请求与响应关联起来。

标志NLM\_F\_ECHO表示该消息是相关的一个包的回传。

标志NLM\_F\_ROOT 被许多 netlink 协议的各种数据获取操作使用，该标志指示被请求的数据表应当整体返回用户应用，而不是一个条目一个条目地返回。有该标志的请求通常导致响应消息设置NLM\_F\_MULTI标志。注意，当设置了该标志时，请求是协议特定的，因此，需要在字段 nlmsg\_type 中指定协议类型。

标志 NLM\_F\_MATCH 表示该协议特定的请求只需要一个数据子集，数据子集由指定的协议特定的过滤器来匹配。

标志 NLM\_F\_ATOMIC 指示请求返回的数据应当原子地收集，这预防数据在获取期间被修改。

标志 NLM\_F\_DUMP 未实现。

标志 NLM\_F\_REPLACE 用于取代在数据表中的现有条目。

标志 NLM\_F\_EXCL\_ 用于和 CREATE 和 APPEND 配合使用，如果条目已经存在，将失败。

标志 NLM\_F\_CREATE 指示应当在指定的表中创建一个条目。

标志 NLM\_F\_APPEND 指示在表末尾添加新的条目。

内核需要读取和修改这些标志，对于一般的使用，用户把它设置为 0 就可以，只是一些高级应用（如 netfilter 和路由 daemon 需要它进行一些复杂的操作），字段 nlmsg\_seq 和 nlmsg\_pid 用于应用追踪消息，前者表示顺序号，后者为消息来源进程 ID。下面是一个示例：

#define MAX\_MSGSIZE 1024  
char buffer[] = "An example message";  
struct nlmsghdr nlhdr;  
nlhdr = (struct nlmsghdr \*)malloc(NLMSG\_SPACE(MAX\_MSGSIZE));  
strcpy(NLMSG\_DATA(nlhdr),buffer);  
nlhdr->nlmsg\_len = NLMSG\_LENGTH(strlen(buffer));  
nlhdr->nlmsg\_pid = getpid();  
nlhdr->nlmsg\_flags = 0;

结构 struct iovec 用于把多个消息通过一次系统调用来发送，下面是该结构使用示例：

struct iovec iov;  
iov.iov\_base = (void \*)nlhdr;  
iov.iov\_len = nlh->nlmsg\_len;  
msg.msg\_iov = &iov;  
msg.msg\_iovlen = 1;

在完成以上步骤后，消息就可以通过下面语句直接发送：

|  |
| --- |
| sendmsg(fd, &msg, 0); |

应用接收消息时需要首先分配一个足够大的缓存来保存消息头以及消息的数据部分，然后填充消息头，添完后就可以直接调用函数 recvmsg() 来接收。

#define MAX\_NL\_MSG\_LEN 1024   
struct sockaddr\_nl nladdr;  
struct msghdr msg;  
struct iovec iov;  
struct nlmsghdr \* nlhdr;  
nlhdr = (struct nlmsghdr \*)malloc(MAX\_NL\_MSG\_LEN);  
iov.iov\_base = (void \*)nlhdr;  
iov.iov\_len = MAX\_NL\_MSG\_LEN;  
msg.msg\_name = (void \*)&(nladdr);  
msg.msg\_namelen = sizeof(nladdr);  
msg.msg\_iov = &iov;  
msg.msg\_iovlen = 1;  
recvmsg(fd, &msg, 0);

注意：fd为socket调用打开的netlink socket描述符。

在消息接收后，nlhdr指向接收到的消息的消息头，nladdr保存了接收到的消息的目标地址，宏NLMSG\_DATA(nlhdr)返回指向消息的数据部分的指针。

在linux/netlink.h中定义了一些方便对消息进行处理的宏，这些宏包括：

|  |
| --- |
| #define NLMSG\_ALIGNTO 4  #define NLMSG\_ALIGN(len) ( ((len)+NLMSG\_ALIGNTO-1) & ~(NLMSG\_ALIGNTO-1) ) |

宏NLMSG\_ALIGN(len)用于得到不小于len且字节对齐的最小数值。

|  |
| --- |
| #define NLMSG\_LENGTH(len) ((len)+NLMSG\_ALIGN(sizeof(struct nlmsghdr))) |

宏NLMSG\_LENGTH(len)用于计算数据部分长度为len时实际的消息长度。它一般用于分配消息缓存。

|  |
| --- |
| #define NLMSG\_SPACE(len) NLMSG\_ALIGN(NLMSG\_LENGTH(len)) |

宏NLMSG\_SPACE(len)返回不小于NLMSG\_LENGTH(len)且字节对齐的最小数值，它也用于分配消息缓存。

|  |
| --- |
| #define NLMSG\_DATA(nlh) ((void\*)(((char\*)nlh) + NLMSG\_LENGTH(0))) |

宏NLMSG\_DATA(nlh)用于取得消息的数据部分的首地址，设置和读取消息数据部分时需要使用该宏。

|  |
| --- |
| #define NLMSG\_NEXT(nlh,len) ((len) -= NLMSG\_ALIGN((nlh)->nlmsg\_len), \  (struct nlmsghdr\*)(((char\*)(nlh)) + NLMSG\_ALIGN((nlh)->nlmsg\_len))) |

宏NLMSG\_NEXT(nlh,len)用于得到下一个消息的首地址，同时len也减少为剩余消息的总长度，该宏一般在一个消息被分成几个部分发送或接收时使用。

|  |
| --- |
| #define NLMSG\_OK(nlh,len) ((len) >= (int)sizeof(struct nlmsghdr) && \  (nlh)->nlmsg\_len >= sizeof(struct nlmsghdr) && \  (nlh)->nlmsg\_len <= (len)) |

宏NLMSG\_OK(nlh,len)用于判断消息是否有len这么长。

|  |
| --- |
| #define NLMSG\_PAYLOAD(nlh,len) ((nlh)->nlmsg\_len - NLMSG\_SPACE((len))) |

宏NLMSG\_PAYLOAD(nlh,len)用于返回payload的长度。

函数close用于关闭打开的netlink socket。

**netlink内核API**

netlink的内核实现在.c文件net/core/af\_netlink.c中，内核模块要想使用netlink，也必须包含头文件linux/netlink.h。内核使用netlink需要专门的API，这完全不同于用户态应用对netlink的使用。如果用户需要增加新的netlink协议类型，必须通过修改linux/netlink.h来实现，当然，目前的netlink实现已经包含了一个通用的协议类型NETLINK\_GENERIC以方便用户使用，用户可以直接使用它而不必增加新的协议类型。前面讲到，为了增加新的netlink协议类型，用户仅需增加如下定义到linux/netlink.h就可以：

|  |
| --- |
| #define NETLINK\_MYTEST 17 |

只要增加这个定义之后，用户就可以在内核的任何地方引用该协议。

在内核中，为了创建一个netlink socket用户需要调用如下函数：

|  |
| --- |
| struct sock \* netlink\_kernel\_create(int unit, void (\*input)(struct sock \*sk, int len)); |

参数unit表示netlink协议类型，如NETLINK\_MYTEST，参数input则为内核模块定义的netlink消息处理函数，当有消息到达这个netlink socket时，该input函数指针就会被引用。函数指针input的参数sk实际上就是函数netlink\_kernel\_create返回的struct sock指针，sock实际是socket的一个内核表示数据结构，用户态应用创建的socket在内核中也会有一个struct sock结构来表示。下面是一个input函数的示例：

|  |
| --- |
| void input (struct sock \*sk, int len)  {  struct sk\_buff \*skb;  struct nlmsghdr \*nlh = NULL;  u8 \*data = NULL;  while ((skb = skb\_dequeue(&sk->receive\_queue)) != NULL)  {  nlh = (struct nlmsghdr \*)skb->data;  data = NLMSG\_DATA(nlh);  }  } |

函数input()会在发送进程执行sendmsg()时被调用，这样处理消息比较及时，但是，如果消息特别长时，这样处理将增加系统调用sendmsg()的执行时间，对于这种情况，可以定义一个内核线程专门负责消息接收，而函数input的工作只是唤醒该内核线程，这样sendmsg将很快返回。

函数skb = skb\_dequeue(&sk->receive\_queue)用于取得socket sk的接收队列上的消息，返回为一个struct sk\_buff的结构，skb->data指向实际的netlink消息。

函数skb\_recv\_datagram(nl\_sk)也用于在netlink socket nl\_sk上接收消息，与skb\_dequeue的不同指出是，如果socket的接收队列上没有消息，它将导致调用进程睡眠在等待队列nl\_sk->sk\_sleep，因此它必须在进程上下文使用，刚才讲的内核线程就可以采用这种方式来接收消息。

下面的函数input就是这种使用的示例：

|  |
| --- |
| void input (struct sock \*sk, int len) { wake\_up\_interruptible(sk->sk\_sleep); } |

当内核中发送netlink消息时，也需要设置目标地址与源地址，而且内核中消息是通过struct sk\_buff来管理的， linux/netlink.h中定义了一个宏：

|  |
| --- |
| #define NETLINK\_CB(skb) (\*(struct netlink\_skb\_parms\*)&((skb)->cb)) |

来方便消息的地址设置。下面是一个消息地址设置的例子：

|  |
| --- |
| NETLINK\_CB(skb).pid = 0; NETLINK\_CB(skb).dst\_pid = 0; NETLINK\_CB(skb).dst\_group = 1; |

字段pid表示消息发送者进程ID，也即源地址，对于内核，它为 0， dst\_pid 表示消息接收者进程 ID，也即目标地址，如果目标为组或内核，它设置为 0，否则 dst\_group 表示目标组地址，如果它目标为某一进程或内核，dst\_group 应当设置为 0。

在内核中，模块调用函数 netlink\_unicast 来发送单播消息：

|  |
| --- |
| int netlink\_unicast(struct sock \*sk, struct sk\_buff \*skb, u32 pid, int nonblock); |

参数sk为函数netlink\_kernel\_create()返回的socket，参数skb存放消息，它的data字段指向要发送的netlink消息结构，而skb的控制块保存了消息的地址信息，前面的宏NETLINK\_CB(skb)就用于方便设置该控制块，参数pid为接收消息进程的pid，参数nonblock表示该函数是否为非阻塞，如果为1，该函数将在没有接收缓存可利用时立即返回，而如果为0，该函数在没有接收缓存可利用时睡眠。

内核模块或子系统也可以使用函数netlink\_broadcast来发送广播消息：

|  |
| --- |
| void netlink\_broadcast(struct sock \*sk, struct sk\_buff \*skb, u32 pid, u32 group, int allocation); |

前面的三个参数与netlink\_unicast相同，参数group为接收消息的多播组，该参数的每一个代表一个多播组，因此如果发送给多个多播组，就把该参数设置为多个多播组组ID的位或。参数allocation为内核内存分配类型，一般地为GFP\_ATOMIC或GFP\_KERNEL，GFP\_ATOMIC用于原子的上下文（即不可以睡眠），而GFP\_KERNEL用于非原子上下文。

在内核中使用函数sock\_release来释放函数netlink\_kernel\_create()创建的netlink socket：

|  |
| --- |
| void sock\_release(struct socket \* sock); |

注意函数netlink\_kernel\_create()返回的类型为struct sock，因此函数sock\_release应该这种调用：

|  |
| --- |
| sock\_release(sk->sk\_socket); |

sk为函数netlink\_kernel\_create()的返回值。

sk为函数netlink\_kernel\_create()的返回值。在源代码包中给出了一个使用 netlink 的示例，它包括一个内核模块 netlink-exam-kern.c 和两个应用程序 netlink-exam-user-recv.c, netlink-exam-user-send.c。内核模块必须先插入到内核，然后在一个终端上运行用户态接收程序，在另一个终端上运行用户态发送程序，发送程序读取参数指定的文本文件并把它作为 netlink 消息的内容发送给内核模块，内核模块接受该消息保存到内核缓存中，它也通过proc接口出口到 procfs，因此用户也能够通过 /proc/netlink\_exam\_buffer 看到全部的内容，同时内核也把该消息发送给用户态接收程序，用户态接收程序将把接收到的内容输出到屏幕上。

## 参考资料

<http://blog.chinaunix.net/uid-23069658-id-3405954.html>

<http://blog.sina.com.cn/s/blog_62ec291601010qgh.html>

<http://blog.chinaunix.net/uid-20788470-id-1841640.html>

<http://blog.csdn.net/hguisu/article/details/7445768/>