用RPM文件编译内核及设置串口 配置文件的几个C函数（备份）

Linux用户空间和内核空间数据交换方式之netlink学习笔记(1)

2012-06-02 11:33:27 | 分类： Linux System Adm | 标签： netlink 订阅 | 字号 | 举报

项目中用到了netlink，为了搞清楚其工作机制，于IBM网站找到燚 杨的一篇博客，里面详细讨论了Linux用户空间和内核空间进行数据交换的各种方式，并且提供了实例代码，这对我正入门学习linux内核来说是一个惊喜，我决定根据其文章做几篇笔记。

原文网址：http://www.ibm.com/developerworks/cn/linux/l-kerns-usrs/

一、netlink简介

Netlink 是一种特殊的 socket,它是 Linux 所特有的,目前在 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)。

今天要讨论的就是通用netlink(NETLINK_CENERIC).

二、用户态使用netlink

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。

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,使用 netlink 的应用必须包含头文件 linux/netlink.h、sys/socket.h。

2.1 创建socket

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， 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。

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。

2.2 绑定socket

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 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。

2.3 发送消息

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;   /* Length of message */
    __u16 nlmsg_type;  /* Message type*/
    __u16 nlmsg_flags; /* Additional flags */
    __u32 nlmsg_seq;   /* Sequence number */
    __u32 nlmsg_pid;   /* Sending process 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。

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();  /* self pid */
 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);
```

2.4 接收消息

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。

2.5 消息处理

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：

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

*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。

数值，它也用于分配消息缓存。

*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。

*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。

*宏NLMSG_OK(nlh,len)用于判断消息是否有len这么长。

*宏NLMSG_PAYLOAD(nlh,len)用于返回payload的长度。

2.6 关闭socket

函数close用于关闭打开的netlink socket。

三、**netlink内核API**

netlink的内核实现在.c文件net/netlink/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。

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。

3.1 创建socket

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) {
  /* process netlink message pointed by skb->data */
  nlh = (struct nlmsghdr *)skb->data;
  data = NLMSG_DATA(nlh);
  /* process netlink message with header pointed by    * nlh
and data pointed by data   */
  }
}
```

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); }
```

3.2 发送消息

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。

3.3 释放socket

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。

四、实例代码

博客中还提供了实例代码下载，基本流程是建立一个内核模块、一个发送消息程序和一个接收消息程序。

4.1 内核模块文件

```
//kernel module: netlink-exam-kern.c
#include <linux/config.h>
#include <linux/module.h>
#include <linux/netlink.h>
#include <linux/sched.h>
#include <net/sock.h>
#include <linux/proc_fs.h>

#define BUF_SIZE 16384
static struct sock *netlink_exam_sock;
static unsigned char buffer[BUF_SIZE];
static unsigned int buffer_tail = 0;
static int exit_flag = 0;
static DECLARE_COMPLETION(exit_completion);

static void recv_handler(struct sock * sk, int length)
{
        wake_up(sk->sk_sleep);
}

static int process_message_thread(void * data)
{
        struct sk_buff * skb = NULL;
        struct nlmsghdr * nlhdr = NULL;
```

```
        int len;
        DEFINE_WAIT(wait);

        daemonize("mynetlink");

        while (exit_flag == 0) {
                prepare_to_wait(netlink_exam_sock->sk_sleep, &wait, TASK_INTERRUPTIBLE);
                schedule();
                finish_wait(netlink_exam_sock->sk_sleep, &wait);

                while ((skb = skb_dequeue(&netlink_exam_sock->sk_receive_queue))
                         != NULL) {
                        nlhdr = (struct nlmsghdr *)skb->data;
                        if (nlhdr->nlmsg_len < sizeof(struct nlmsghdr)) {
                                printk("Corrupt netlink message.\n");
                                continue;
                        }
                        len = nlhdr->nlmsg_len - NLMSG_LENGTH(0);
                        if (len + buffer_tail > BUF_SIZE) {
                                printk("netlink buffer is full.\n");
                        }
                        else {
                                memcpy(buffer + buffer_tail, NLMSG_DATA(nlhdr), len);
                                buffer_tail += len;
```

```
                        }
                        nlhdr->nlmsg_pid = 0;
                        nlhdr->nlmsg_flags = 0;
                        NETLINK_CB(skb).pid = 0;
                        NETLINK_CB(skb).dst_pid = 0;
                        NETLINK_CB(skb).dst_group = 1;
                        netlink_broadcast(netlink_exam_sock, skb, 0, 1, GFP_KERNEL);
                }
        }
        complete(&exit_completion);
        return 0;
}

static int netlink_exam_readproc(char *page, char **start, off_t off,
                        int count, int *eof, void *data)
{
        int len;

        if (off >= buffer_tail) {
                * eof = 1;
                return 0;
        }
        else {
                len = count;
```

```
                if (count > PAGE_SIZE) {
                        len = PAGE_SIZE;
                }
                if (len > buffer_tail - off) {
                        len = buffer_tail - off;
                }
                memcpy(page, buffer + off, len);
                *start = page;
                return len;
        }

}

static int __init netlink_exam_init(void)
{
        netlink_exam_sock = netlink_kernel_create(NETLINK_GENERIC, 0, recv_handler,
THIS_MODULE);
        if (!netlink_exam_sock) {
                printk("Fail to create netlink socket.\n");
                return 1;
        }
        kernel_thread(process_message_thread, NULL, CLONE_KERNEL);
        create_proc_read_entry("netlink_exam_buffer", 0444, NULL, netlink_exam_readproc,
0);
```

```
        return 0;
}

static void __exit netlink_exam_exit(void)
{
        exit_flag = 1;
        wake_up(netlink_exam_sock->sk_sleep);
        wait_for_completion(&exit_completion);
        sock_release(netlink_exam_sock->sk_socket);
}

module_init(netlink_exam_init);
module_exit(netlink_exam_exit);
MODULE_LICENSE("GPL");
```

4.2 发送消息文件

```
//application sender: netlink-exam-user-send.c
#include <stdio.h>
#include <sys/types.h>
```

```c
#include <sys/socket.h>
#include <linux/netlink.h>

#define MAX_MSGSIZE 1024


int main(int argc, char * argv[])
{
        FILE * fp;
        struct sockaddr_nl saddr, daddr;
        struct nlmsghdr *nlhdr = NULL;
        struct msghdr msg;
        struct iovec iov;
        int sd;
        char text_line[MAX_MSGSIZE];
        int ret = -1;

        if (argc < 2) {
                printf("Usage: %s atextfilename\n", argv[0]);
                exit(1);
        }

        if ((fp = fopen(argv[1], "r")) == NULL) {
                printf("File %s dosen't exist.\n");
```

```
        exit(1);
    }

    sd = socket(AF_NETLINK, SOCK_RAW,NETLINK_GENERIC);
    memset(&saddr, 0, sizeof(saddr));
    memset(&daddr, 0, sizeof(daddr));

    saddr.nl_family = AF_NETLINK;
    saddr.nl_pid = getpid();
    saddr.nl_groups = 0;
    bind(sd, (struct sockaddr*)&saddr, sizeof(saddr));

    daddr.nl_family = AF_NETLINK;
    daddr.nl_pid = 0;
    daddr.nl_groups = 0;

    nlhdr = (struct nlmsghdr *)malloc(NLMSG_SPACE(MAX_MSGSIZE));

    while (fgets(text_line, MAX_MSGSIZE, fp)) {
        memcpy(NLMSG_DATA(nlhdr), text_line, strlen(text_line));
        memset(&msg, 0 ,sizeof(struct msghdr));

        nlhdr->nlmsg_len = NLMSG_LENGTH(strlen(text_line));
        nlhdr->nlmsg_pid = getpid();  /* self pid */
```

```
                nlhdr->nlmsg_flags = 0;

                iov.iov_base = (void *)nlhdr;
                iov.iov_len = nlhdr->nlmsg_len;
                msg.msg_name = (void *)&daddr;
                msg.msg_namelen = sizeof(daddr);
                msg.msg_iov = &iov;
                msg.msg_iovlen = 1;
                ret = sendmsg(sd, &msg, 0);
                if (ret == -1) {
                        perror("sendmsg error:");
                }
        }

        close(sd);
}
```

4.3 接收消息文件

```
//application receiver: netlink-exam-user-recv.c
#include <stdio.h>
```

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <linux/netlink.h>

#define MAX_MSGSIZE 1024


int main(void)
{
        struct sockaddr_nl saddr, daddr;
        struct nlmsghdr *nlhdr = NULL;
        struct msghdr msg;
        struct iovec iov;
        int sd;
        int ret = 1;

        sd = socket(AF_NETLINK, SOCK_RAW,NETLINK_GENERIC);
        memset(&saddr, 0, sizeof(saddr));
        memset(&daddr, 0, sizeof(daddr));

        saddr.nl_family = AF_NETLINK;
        saddr.nl_pid = getpid();
        saddr.nl_groups = 1;
        bind(sd, (struct sockaddr*)&saddr, sizeof(saddr));
```

```
        nlhdr = (struct nlmsghdr *)malloc(NLMSG_SPACE(MAX_MSGSIZE));

        while (1) {
                memset(nlhdr, 0, NLMSG_SPACE(MAX_MSGSIZE));

                iov.iov_base = (void *)nlhdr;
                iov.iov_len = NLMSG_SPACE(MAX_MSGSIZE);
                msg.msg_name = (void *)&daddr;
                msg.msg_namelen = sizeof(daddr);
                msg.msg_iov = &iov;
                msg.msg_iovlen = 1;

                ret = recvmsg(sd, &msg, 0);
                if (ret == 0) {
                        printf("Exit.\n");
                        exit(0);
                }
                else if (ret == -1) {
                        perror("recvmsg:");
                        exit(1);
                }
                printf("%s", NLMSG_DATA(nlhdr));
        }
```

```
        close(sd);
}
```

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。

五、实践

将上面代码用于linux-2.6.13内核编译时发生一些错误，比如netlink.h中：

#define NETLINK_GENERIC 16

缺少这行代码。

另外，dst_group改为dst_groups。

您可能也喜欢：

配置文件的几个C函数（备份） 2012.06.04
配置文件的几个C函数（备份）

地址族AF_INET和协议族PF_INET 2012.01.05
地址族AF_INET和协议族PF_INET在Linux

获取CPU、网卡序列号并进行产生MD5哈 2012.06.08
获取CPU、网卡序列号并进行产生MD5哈希值

C语言中——嵌套结构体初始化模式—— 2012.01.08
C语言中——嵌套结构体初始化模式——学习笔记

DES加密解密举例 2012.06.08
DES加密解密举例

AWK语言学习笔记——文本处理利器 2012.07.01
AWK语言学习笔记——文本处理利器

getopt和getopt_long使用举例 2012.06.04
getopt和getopt_long使用举例

编译Linux内核、启动新内核和编译自定 2012.05.12
编译Linux内核、启动新内核和编译自定义模块

linux内核源码编译，制作可启动内核镜像 2012.02.04
linux内核源码编译，制作可启动内核镜像

Linux下文本编码格式转换 2011.12.17
Linux下文本编码格式转换

ASDF打包Common Lisp程序——学习 2012.04.22
ASDF打包Common Lisp程序——学习笔记

Linux搭建C开发环境 2012.01.08
Linux搭建C开发环境

LeslieChu

被关注 3

+关注

""

1 什么是STL STL就是C++ Standard

http://t.163.com/lesliechu

用RPM文件编译内核及设置串口 配置文件的几个C函数（备份）

最近读者

登录后，您可以在此留下足迹。 pczb sqhxhg mou0…

评论

点击登录 | 昵称：

发表

minminjunzhu 2012-06-03 21:48

这么多，你得写多久啊。。。挺你。加油。。。

回复

zhubuntu 回复 minminjunzhu 2012-06-03 21:53

复制

回复
