linux 内核 IOCTL 网络控制框架实现分析

目录

一、概述

从 ioctl 这个名称上看,它是设备驱动程序中对设备的 I/O 通道进行管理的函数。所谓对 I/O 通道进行管理，就是对设备的一些特性进行控制，例如串口的传输波特率、马达的转速等等，但实际上 ioctl 所处理的对象并不限制是真正的 I/O 设备，还可以是其它任何一个内核设备。ioctl 以系统调用的形式提供了一条用户与内核交互的便捷途径。当前一些宽带计费网关、防火墙系统均利用 ioctl 与内核良好的通信互动特点支持用户对基于内核模块的软件系统的控制。本文针对 i386 平台下的 ioctl 内核网络源代码控制框架进行剖析解释，在文章最后列举一个实例，通过编程实践展示如何通过 ioctl 控制函数实现自定义的功能的控制，使读者可以对 ioctl 实现原理有一个全面的认识，本文只对 ioctl 实现流程框架做一定的叙述，并不会深入到具体的控制函数。为了更好的阅读本文，要求读者对 Linux 下的网络编程有一定的了解。

本文约定：

1、以下内容如果没有特殊说明，均参照 linux 内核 2.4.0 版本

2、“->”箭头符表示函数调用关系，如 sys_socket->sock_map_fd 表示 sys_socket 函数调用的 sock_map_fd 函数。

3、第五节的实践是在 redhat9 上实现，基于 2.4.20 内核，但本文所述在 2.4 内核下都适用。

二、用户空间 ioctl 控制函数调用形式

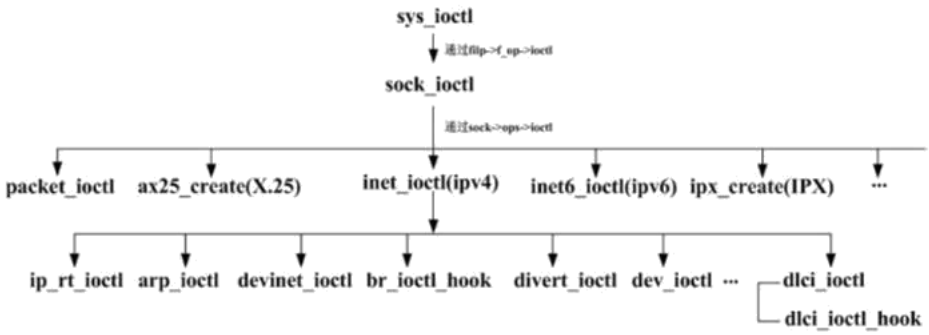
通过 man 2 ioctl 命令查看 ioctl 函数的调用形式类似如下：

```
#include <sys/ioctl.h>
int ioctl(int d, int request, ...);
```

其中 d 就是用户程序打开设备时使用 open 函数返回的文件描述符，request 就是用户程序对设备的控制命令，至于后面的省略号，则是一些补充参数，一般最多一个，有或没有是和 request 的意义相关的，详情请参考 man 2 ioctl_list 以了解更多。ioctl 函数是文件结构中的一个属性分量，就是说如果驱动程序提供了对 ioctl 的支持，用户就可以在用户程序中使用 ioctl 函数控制设备的 I/O 通道或其它一些自己想要控制且设备支持的功能。

三、内核主要函数调用框架

内核实现 ioctl()函数的是 sys_ioctl()，在内核中主要调用框架图如下，它清晰地给我们展示 ioctl 的控制传递框架，我们接下来的内容将根据此图向大家做详细的解释：



四、IOCTL 框架源代码分析

根据前面的图示,我们从入口函数 sys_ioctl 开始分析:

4.1、入口函数：sys_ioctl

以下源码在 fs/ioctl.c 中，其中删除了部分与网络控制关系不大的代码：

```
asmlinkage long sys_ioctl(unsigned int fd, unsigned int cmd, unsigned long arg)
{
    …//根据 fd 获取文件结构(struct file)
```

```
        lock_kernel();
        switch (cmd) {
            case FIOCLEX://对文件设置专用标志，通知内核自动关闭打开的文件
            …
            case FIONCLEX://与 FIOCLEX 标志相反，清除专用标志
            …
            case FIONBIO://将文件操作设置成阻塞/非阻塞
            …
            case FIOASYNC:// 将文件操作设置成同步/异步 IO
            …   //以上省略的代码是关于具体的磁盘文件系统的控制处理，
                //关于 socket 的阻塞或非阻塞等设置很简单，有兴趣的读者直接阅读源码吧
    default: //文件其它部分的处理被放在了 default 部分
                error = -ENOTTY;
                if (S_ISREG(filp->f_dentry->d_inode->i_mode)) //普通文件
                    error = file_ioctl(filp, cmd, arg); //
                else if (filp->f_op && filp->f_op->ioctl) //socket 控制在此处理
                    error = filp->f_op->ioctl(filp->f_dentry->d_inode, filp, cmd, arg);
        }
        unlock_kernel();
        fput(filp);
    out:
        return error;
    }
```

注意上面蓝色字体部分，即为调用网络部分的代码入口。大家注意在 default 情况下，有个 S_ISREG 宏对文件类型作判断，其定义在 include/linux/stat.h 中：

```
#define S_ISLNK(m)     (((m) & S_IFMT) == S_IFLNK) //符号连接文件
#define S_ISREG(m)     (((m) & S_IFMT) == S_IFREG) //普通文件
#define S_ISDIR(m)     (((m) & S_IFMT) == S_IFDIR)   //目录文件
#define S_ISCHR(m)     (((m) & S_IFMT) == S_IFCHR) //字符设备文件
#define S_ISBLK(m)     (((m) & S_IFMT) == S_IFBLK)   //块设备文件
#define S_ISFIFO(m)    (((m) & S_IFMT) == S_IFIFO)   //管道文件
#define S_ISSOCK(m)    (((m) & S_IFMT) == S_IFSOCK)      //socket 套接字文件
```

因为 linux 内核把 socket 套接字当作文件来处理，内核在创建 socket 套接字时，为套接字分配文件 id 以及生成与 id 对应的文件节点，节点的 i_mode 域是代表文件类型的位域标志字段，所以内核定义了上述宏来简化判断操作。由于套接字文件不属于普通文件之列，所以程序直接执行蓝色字体部分。

4.2、入口函数跳转

我们来看一下 filp->f_op->ioctl 函数指针指向了什么函数，可以参考 net/socket.c 文件中的 sys_socket->sock_map_fd 函数中的一行代码(蓝色部分代码)：

```
static int sock_map_fd(struct socket *sock)
{
    …
    sock->file = file;
    file->f_op = sock->inode->i_fop = &socket_file_ops;
    file->f_mode = 3;
    file->f_flags = O_RDWR;
    file->f_pos = 0;
    …
}
```

内核在用户创建 socket 套接字时就将此套接字的文件操作函数指针初始化了。从上面的代码我们可以看到，filp->f_op 以及文件对应的 socket 节点的 i_fop 指针都被赋值为指向 socket_file_ops 结构，所以我们来看看内核是如何实现这个控制过程的转移的。还是在内核的 net/socket.c 文件中，定义了 socket_file_ops 结构如下：

```
static struct file_operations socket_file_ops = {
  llseek:         sock_lseek,
  read:               sock_read,
  write:          sock_write,
  poll:           sock_poll,
  ioctl:          sock_ioctl,
  mmap:           sock_mmap,
  open:           sock_no_open,       /* special open code to disallow open via /proc */
```

```
    release:        sock_close,
    fasync:         sock_fasync,
    readv:          sock_readv,
    writev:         sock_writev
};
```

从上面的代码来看,这个结构定义了 socket 描述字的文件操作函数，如对描述字调用 read 函数读数据时最终将访问 sock_read 函数，对描述字调用 write 函数读数据时最终将访问 sock_write 函数，等等。而对 ioctl 的访问最终将转化为调用 sock_ioctl 函数，看到此处我们明白了，filp->f_op->ioctl(filp->f_dentry->d_inode, filp, cmd, arg)调用实质上转化为对 sock_ioctl 函数的调用。

4.3、sock_ioctl 函数

sock_ioctl 函数依然在 net/socket.c 文件中，列出如下：

```
int sock_ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)
{
    struct socket *sock;
    int err;

    unlock_kernel();
    sock = socki_lookup(inode);
    err = sock->ops->ioctl(sock, cmd, arg);
    lock_kernel();

    return err;
}
```

此处函数引入 inode 参数实质是通过节点找到套接字对应的 socket 结构，通过 socket 的 struct proto_ops 类型的字段 ops 执行具体的控制操作(即 sock->ops->ioctl(sock, cmd, arg))，函数 socki_lookup 也在文件 net/socket.c 中，列出如下：

```
extern __inline__ struct socket *socki_lookup(struct inode *inode)
{
    return &inode->u.socket_i;
}
```

写到这大家可能要问为什么不直接在 filp->f_op->ioctl 函数指针指向的函数里面执行 ioctl 控制操作而要做两次跳转呢？其实这与 linux 良好的设计规范和业务支持的实际情况都有关系，第一次跳转是转入套接字单独处理，因为内核中网络部分是非常重要的，可以与文件系统相提并论，将网络部分独立出来处理在设计思路上更清晰；另外，linux 内核支持不同层次、类型的套接字，如 ipv4、ipv6 套接字以及 sock_raw 原始套接字，对于这些套接字的处理有一定的相似性，又有其不同的地方。所以引入第二次跳转的目的也即在此,以支持对不同的协议类型的套接字进行不同控制,详情见下面小节的介绍。

4.4、二次跳转

闲话少说，步入正题。接下来我们看看 sock->ops->ioctl 函数指针调用了什么函数，首先看看 sock 变量的结构类型 struct socket,大家要多注意这个结构,在后面我们也列出了相关结构相互引用图中涉及到的这个结构的几个字段,以加深大家的印象.结构的源代码在 include/linux/Net.h 文件中：

```
struct socket
{
    socket_state        state;

    unsigned long       flags;
    struct proto_ops    *ops;
    struct inode        *inode;
    struct fasync_struct    *fasync_list;   /* Asynchronous wake up list    */
    struct file     *file;          /* File back pointer for gc     */
    struct sock         *sk;
    wait_queue_head_t wait;
…
};
```

套接字就是通过结构中 ops 指针来执行具体的 ioctl 控制函数的。struct proto_ops 定义在同样的头文件中：

```
struct proto_ops {
    int     family;
```

```
    int     (*release)      (struct socket *sock);
    int     (*bind)         (struct socket *sock, struct sockaddr *umyaddr, int sockaddr_len);
    int     (*connect)      (struct socket *sock, struct sockaddr *uservaddr, int sockaddr_len, int flags);
    int     (*socketpair) (struct socket *sock1, struct socket *sock2);
    int     (*accept) (struct socket *sock, struct socket *newsock,    int flags);
    int     (*getname) (struct socket *sock, struct sockaddr *uaddr, int *usockaddr_len, int peer);
    unsigned int (*poll)    (struct file *file, struct socket *sock, struct poll_table_struct *wait);
     int     (*ioctl)    (struct socket *sock, unsigned int cmd, unsigned long arg);
    int     (*listen)   (struct socket *sock, int len);
    int     (*shutdown)   (struct socket *sock, int flags);
    int     (*setsockopt) (struct socket *sock, int level, int optname, char *optval, int optlen);
    int     (*getsockopt) (struct socket *sock, int level, int optname, char *optval, int *optlen);
    int (*sendmsg) (struct socket *sock, struct msghdr *m, int total_len, struct scm_cookie *scm);
    int (*recvmsg) (struct socket *sock, struct msghdr *m, int total_len, int flags, struct scm_cookie *scm);
    int     (*mmap) (struct file *file, struct socket *sock, struct vm_area_struct * vma);
};
```

补充一下基础知识,一个套接字接口在逻辑上有三个要素:网域,类型和规程(协议).

网域：表明套接字接口用于哪一中网络或这说哪一族网络规程.就是我们通常说的地址族(family),常见的有AF_UNIX/AF_INET/AF_X25/AF_IPX 等待.

类型：表明通讯中所遵循的模式,主要有两种模式:"有连接"和"无连接",对应到以太网就是 SOCK_STREAM 和SOCK_DGRAM 两种.

规程：具体的网络协议.通常,网域和类型基本就能够确定使用的规程了.

这里的 proto_ops 结构就是通过不同的实例来支持具体的网域的不同类型、规程所使用的通信函数,每个网域都有多种类型、多种规程，所以也有多个 proto_ops 实例,给这个实例赋值具体规程的处理函数,如 ipv4 的有连接和无连接实例所指定的控制函数都是 inet_ioctl(如果处理不同也可以指向不同的控制函数),这样可以使具体的控制操作转向具体的处理,细节实现我们下一小节介绍.

构造内核时，内核会初始化网络地址族,即初始化 net_families[NRPORO]全局量,这是一个静态指针数组。每个网域地址族的初始化函数都由其中一个元素来表征，例如，"INET"和它的初始程序地址分别是 PF_INET（等同于 AF_INET）和 inet_create。当套接口启动时被初始化时，要调用每一网域初始化程序,为具体的类型指定处理函数,内核初始化网域地址族后net_families[NRPORO]变量的相关字段取值状态示意图如下:

对 IPV4 地址族来说,这个初始化函数就是 inet_create,其代码在 net/ipv4/af_inet.c 中:

```
static int inet_create(struct socket *sock, int protocol)
{
    …
    switch (sock->type) {
    case SOCK_STREAM:
        if (protocol && protocol != IPPROTO_TCP) //类型与规程检测
            goto free_and_noproto;
        protocol = IPPROTO_TCP;
        prot = &tcp_prot;
        sock->ops = &inet_stream_ops; //此处指定函数跳转表
        break;
    case SOCK_SEQPACKET:
        goto free_and_badtype;
    case SOCK_DGRAM:
        if (protocol && protocol != IPPROTO_UDP)
            goto free_and_noproto;
        protocol = IPPROTO_UDP;
        sk->no_check = UDP_CSUM_DEFAULT;
        prot=&udp_prot;
        sock->ops = &inet_dgram_ops; //此处指定函数跳转表
        break;
    case SOCK_RAW:
        if (!capable(CAP_NET_RAW)) //检验是否有创建原始套接字的权限
            …
        sock->ops = &inet_dgram_ops;//
        if (protocol == IPPROTO_RAW)
            sk->protinfo.af_inet.hdrincl = 1;
        break;
```

```
        default:
            goto free_and_badtype;
        }
    …
    }
```

从上面的代码可以看出：已注册的网域的类型所对应的操作被存在 socket 结构的 ops 指针中，它就是指向具体的 proto_ops 数据结构实例,如 inet_stream_ops、inet_dgram_ops 等。proto_ops 结构由地址族类型和一系列指向与特定地址族对应的 socket 操作函数的指针组成。ops 字段通过地址族标识符来索引，接下来我们看看 proto_ops 结构。

4.5、struct proto_ops 结构实例

前面说过，具体的 ioctl 执行过程时通过两次跳转而来，其中第二次就是针对各个不同层次、类型的套接字。我们来看看内核中所定义的各个具体的 proto_ops 结构实例以分析不同的控制执行流程.　　内核中为每个规程定义了一个 proto_ops 结构实例，常见的如下:

1、在 net/ipv4/Af_inet.c 文件中:

```
struct proto_ops inet_stream_ops = {
    …
    poll:       tcp_poll,
    ioctl:          inet_ioctl,
     listen:          inet_listen,
…
};

struct proto_ops inet_dgram_ops = {
    …
    poll:       datagram_poll,
    ioctl:          inet_ioctl,
    listen:         sock_no_listen,
    …
};
```

可见这两个实例有相当多的处理函数都是一样的,并且最终调用相同的控制函数 inet_ioctl.

2、在 net/ipv6/Af_inet6.c 文件中提供了 inet6_stream_ops 和 inet6_dgram_ops,其地址族及 ioctl 处理函数分别为 PF_INET6 和 inet6_ioctl：

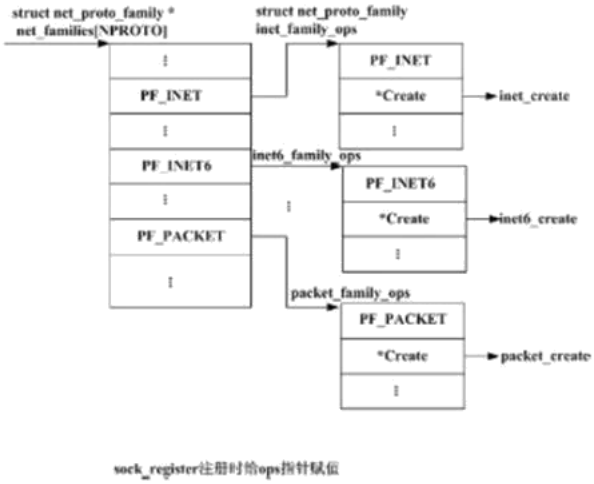
```
struct proto_ops inet6_stream_ops = {
    family:         PF_INET6,
…
    ioctl:          inet6_ioctl,            /* must change */
    …
};

struct proto_ops inet6_dgram_ops = {
    family:         PF_INET6,
…
    ioctl:          inet6_ioctl,            /* must change */
    …
};
```

3、在 net/packet/Af_ packet 6.c 文件中提供了 packet_ops_spkt 和 packet_ops,其地址族及 ioctl 处理函数分别为 PF_PACKET 和 packet_ioctl：

```
struct proto_ops packet_ops = {
    family:         PF_PACKET,
…
    ioctl:          packet_ioctl,
    …
};
```

还有 x25 和 ipx、netlink、unix 域等等地址族所对应的文件提供了各自的协议规程操作函数指针以支持不同的 ioctl 处理函数，大家有兴趣可以参考内核相关源码.

可见，通过二次跳转表，内核可以支持不同协议规程做不同的操作，包括控制处理。本文把重点放在 ipv4 的 ioctl 控制函数,引导大家深入到其处理源码.



4.6、inet_ioctl 函数

由于 inet_ioctl 函数内容分支很多,但功能、处理不难理解,所以我把一些不常见的内容都省去,挑简单重要的说,完全在于抛砖引玉:

```
static int inet_ioctl(struct socket *sock, unsigned int cmd, unsigned long arg)
{
    …
    switch(cmd)
    {
        case FIOSETOWN://设置属主
        case SIOCSPGRP://设置进程组
            err = get_user(pid, (int *) arg);
            if (err)
                return err;
            if (current->pid != pid && current->pgrp != -pid &&
                !capable(CAP_NET_ADMIN))
                return -EPERM;
            sk->proc = pid;
            return(0);
        case FIOGETOWN://获取属主
        case SIOCGPGRP://获取进程组
            return put_user(sk->proc, (int *)arg);
        case SIOCGSTAMP://
            if(sk->stamp.tv_sec==0)
                return -ENOENT;
            err = copy_to_user((void *)arg,&sk->stamp,sizeof(struct timeval));
            if (err)
                err = -EFAULT;
            return err;
        case SIOCADDRT://增加路由
        case SIOCDELRT://删除路由
        case SIOCRTMSG:
            return(ip_rt_ioctl(cmd,(void *) arg));//IP 路由配置
        case SIOCDARP://删除 arp 项
        case SIOCGARP://获取 arp 项
        case SIOCSARP://创建/修改 arp 项
            return(arp_ioctl(cmd,(void *) arg));//arp 配置
        case SIOCGIFADDR://获取接口地址
        case SIOCSIFADDR://设置接口地址
        case SIOCGIFBRDADDR://获取广播地址
        case SIOCSIFBRDADDR://设置广播地址
        case SIOCGIFNETMASK://获取网络掩码
        case SIOCSIFNETMASK://设置网络掩码
        case SIOCGIFDSTADDR://获取 p2p 地址
        case SIOCSIFDSTADDR://设置 p2p 地址
```

```
            case SIOCSIFPFLAGS: //
            case SIOCGIFPFLAGS:
            case SIOCSIFFLAGS://设置接口标志
                return(devinet_ioctl(cmd,(void *) arg));//网络接口相关配置,linux 内核自带的 ifconfig
    //的很多处理都是通过这里实现的
            case SIOCGIFBR:
            case SIOCSIFBR://网桥设置,稍后的实例就是介绍如何截获网桥控制钩子
    #if defined(CONFIG_BRIDGE) || defined(CONFIG_BRIDGE_MODULE) //如果内核支持网桥功能
    #ifdef CONFIG_KMOD//若支持内核模块动态加载
                if (br_ioctl_hook == NULL)//网桥钩子为空则动态请求模块
                    request_module("bridge");//加载网桥模块
    #endif
                if (br_ioctl_hook != NULL)
                    return br_ioctl_hook(arg);//通过钩子函数处理命令参数
    #endif
            case SIOCGIFDIVERT://
            case SIOCSIFDIVERT:
    #ifdef CONFIG_NET_DIVERT
                return(divert_ioctl(cmd, (struct divert_cf *) arg));
    #else
                return -ENOPKG;
    #endif     /* CONFIG_NET_DIVERT */
                return -ENOPKG;

            case SIOCADDDLCI://
            case SIOCDELDLCI://  数据链路连接标识控制
    #ifdef CONFIG_DLCI
                lock_kernel();
                err = dlci_ioctl(cmd, (void *) arg);//控制函数
                unlock_kernel();
                return err;
    #endif

    #ifdef CONFIG_DLCI_MODULE

    #ifdef CONFIG_KMOD
                if (dlci_ioctl_hook == NULL)//如果钩子函数为空,则加载模块
                    request_module("dlci");
    #endif

                if (dlci_ioctl_hook) {//钩子函数指针不空
                    lock_kernel();
                    err = (*dlci_ioctl_hook)(cmd, (void *) arg);//调用钩子函数
                    unlock_kernel();
                    return err;
                }
    #endif
                return -ENOPKG;

            default:
                …
                return err;
        }
        /*NOTREACHED*/
        return(0);
    }
```

从上面的函数代码来看，同套接字有关的控制请求主要有如下几类：

1、文件操作

2、套接字操作

3、路由选项操作

4、接口操作

5、ARP 高速缓存操作

6、网桥控制

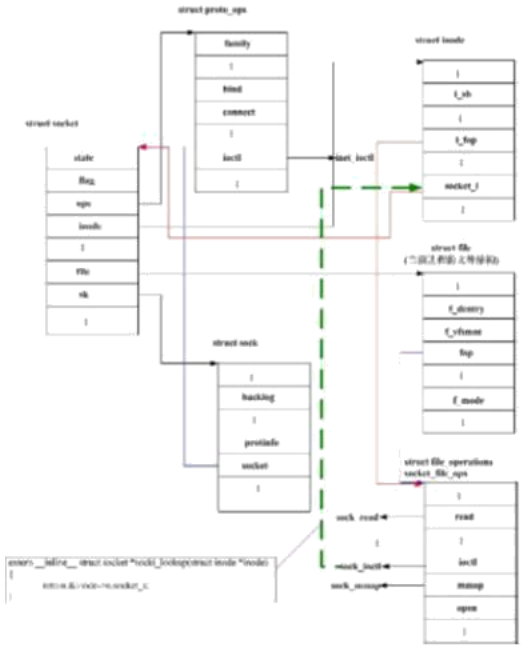
7、数据链路连接标识控制

结合代码中的注释，读者不难理解具体的控制分支。具体的控制处理就转到具体的函数里面去处理了，例如关于内核自带的命令工具 ifconfig 对 ip 地址的配置处理，基本都在 devinet_ioctl 函数中；关于 arp 命令的处理都在 arp_ioctl 中处理；关于路由配置都在 ip_rt_ioctl 中处理。其中参数 arg 是用户空间传来的自定义的数据，可以是结构，可以是联合或其它一些更复杂的类型，由具体的业务模块来解释处理。在随后的实践中，我们就是通过 arg 的不同解释来做不同的处理。

4.7、网络主要结构相关字段相互引用图

通过上面的分析，大家应该大致明白了 linux 内核网络 ioctl 控制框架的实现了。下面是在内核网络组件初始化后,ipv4 相关的结构字段之间相互引用图，供大家阅读是参考:

结合前面主要函数调用关系图与源码分析,读者可以很清晰的顺着上图所示的箭头，从 ioctl 入口函数开始，方便地找到具体的处理模块.其中,文件操作对象 socket_file_ops 调用 sock_ioctl()时,通过 inode 节点的 socket_i 字段最终找到 inet_ioctl()函数.



五、调用实践

此处介绍通过自己编写控制程序,在用户空间调用 ioctl 函数控制内核显示一行信息的例子供大家参考:

5.1、编写运行于用户空间的控制程序

(1)一般先定义自己的结构参数类型,如下:

```
typedef struct stMyIoctlArg {
    unsigned int cmd;//其实就是第一个参数,当作自己的命令参数
    unsigned int arg1; //用于提供给具体的命令参数
    unsigned int arg2;
    …//如果有更多参数直接加在后面
} IOCTL_ARG,P IOCTL_ARG;
```

(2)然后在 main 中赋值并调用 ioctl 函数:

```
#define   FIOCSMYSHOW 0x1234
int main( int argc, char **argv )
{
      int fd;
IOCTL_ARG   arg; //需要组织传递到内核的参数
arg.cmd= FIOCSMYSHOW //自定义命令
…//其它参数赋值
int fd = socket( AF_INET, SOCK_STREAM, 0 );//创建控制 socket
    if ( fd < 0 )
```

```
    {
        perror( "socket failed" );
        return 0;
    }

    if ( ioctl( fd, SIOCSIFBR, &arg) < 0 ) //通过网桥请求参数来控制内核作相关操作
    {
        perror( "ioctl( SIOCSIFBR ) failed" );
        close( fd );
        return 0;
}
…
close(fd);
…
}
```

例子源代码：

5.2、内核功能支持

2.1、修改内核相关代码：

(1)在内核 include/linux/sockios.h 的尾部加入前面定义的公共的结构与常量:

```
typedef struct stMyIoctlArg {
    unsigned int cmd;//其实就是第一个参数,当作自己的命令参数
    unsigned int arg1;
    unsigned int arg2;
    …//如果有更多参数直接加在后面
} IOCTL_ARG,P IOCTL_ARG;
#define FIOCSMYSHOW 0x1234
```

(2)在 inet_ioctl 函数网桥处理分支处增加如下蓝色字体内容:

```
IOCTL_ARG myarg;//在 inet_ioctl 函数开始时加入此变量定义
…
#if defined(CONFIG_BRIDGE) || defined(CONFIG_BRIDGE_MODULE)
if ( copy_from_user( & myarg, (void *) arg, sizeof(IOCTL_ARG ) ) ) //拷贝用户空间参数
      return -EFAULT;
switch (myarg.cmd ){
      case FIOCSMYSHOW ://解析自己的命令
           printk(KERN_INFO “get ioctl hook./n”); //可以增加对 arg1/arg2 等参数的解析处理
           return 0; //直接返回
            break;
      …
      default:
            break;
}
#ifdef CONFIG_KMOD
                if (br_ioctl_hook == NULL)
                     request_module("bridge");
#endif
                if (br_ioctl_hook != NULL)
                     return br_ioctl_hook(arg);
#endif
```

内核修改文件： 。注意在修改内核代码后，用 README 中的命令编译一下修改的文件，没有错误才编译内核，避免走弯路重新编译。

2.2、编译内核

具体编译过程请参照网络上的文章，我所用到的重要的命令有：

```
make mrproper
make oldconfig
make xconfig //在 network options 中选择 802.1 ethernet bridge 选项支持网桥功能
make dep
make bzImage
make modules
make modules_install
depmod -a
cp System.map /boot/System.map-2.4.20-8custom
```

cp arch/i386/boot/bzImage /boot/vmlinuz-2.4.20-8custom
new-kernel-pkg --install --mkinitrd --depmod 2.4.20-8custom

5.3、运行控制程序

内核编译前运行显示：

```
[root@localhost ioctl_clt]# dmesg |grep hook
[root@localhost ioctl_clt]# ./myioctl
ioctl( SIOCSIFBR ) failed: Operation not supported
[root@localhost ioctl_clt]#
```

内核编译后运行显示：

```
[root@zhouys ioctl_clt]# ./myioctl
handle end.[root@zhouys ioctl_clt]#
```

5.4、查看结果

可以通过 dmesg | grep hook 命令查看结果，显示：

```
[root@zhouys ioctl_clt]# dmesg |grep hook
get ioctl hook.
```

这正是我们在内核中要打印的字符，说明我们的控制命令已经通知给内核了。

六、结束语

ioctl 系统调用是最常用的用户与内核空间交互的手段之一，linux 系统自带的相当多的命令工具尤其是网络控制工具都是采用 ioctl 控制框架实现了用户和内核通信的桥梁，在当前一些基于内核模块技术的软件系统中也有重要的用途，如某些宽带计费网关、防火墙软件、网络交换机等。了解 ioctl 控制框架，无疑会提高我们对 linux 内核通信机制的认识，也可以指导我们的实践工作。

七、参考资料

1 linux 内核源代码情景分析

2 linux 内核 2.4.0 源码

3 ioctl man 手册

4 ifconfig 工具源码