网络安全课程设计

指导教师：
张云鹤
肖凌
梅松

课程设计任务

——Linux下的状态检测防火墙

一、课程设计的目的

- 计算机网络安全是一门专业核心课，网络安全课程设计是计算机网络安全课程的综合实践环节
- 通过本课程设计，要达到以下目标：
  - 结合理论课程学习，深入理解计算机网络安全的基本原理与协议，巩固计算机网络安全基本理论知识
  - 熟练掌握计算机网络编程方法，拓展学生的应用能力
  - 加强对网络协议栈的理解
  - 提高分析、设计软件系统以及编写文档的能力
  - 培养团队合作能力。

二、要求与评分标准

- 课程设计要求：
  - * 正确理解题意
  - * 具有良好的编程规范和适当的注释
  - * 有详细的文档，文档中应包括设计题目涉及的基础知识、设计思路、程序流程图、程序清单、开发中遇到的问题及解决方法、设计中待解决的问题及改进方向
- 评分标准：
  - * 是否有较好的设计思想
  - * 是否有正确的运行结果
  - * 是否有良好的编程规范和适当的注释
  - * 文档是否完备

三、课程设计的任务

- （1）通过学习和了解Linux Netfilter架构和原理，了解Linux的网络协议栈的处理过程
- （2）利用Linux Netfilter架构，编写一个具有状态检测的linux防火墙，能对收发的报文进行状态分析和过滤
- （3）验证阶段需要综合运用组网技术、协议分析技术
- （4）通过Linux下的开发过程提升学生的内核编程能力和分析问题、解决问题的能力
- （5）设计成果展示讲解
- （6）撰写课程设计报告

课程设计组成

- 课程设计的防火墙系统包括两个部分，一个是防火墙内核模块，一个是应用程序
  - 内核模块用来截获通过防火墙的报文，进行规则过滤，根据规则来决定是禁止还是放行，若放行，是直接路由转发还是NAT之后转发，维护连接表。规则由应用程序通过跟内核的接口写入
  - 应用程序用来设置规则（向内核模块写入），查看日志（从内核模块获得，内核模块不要直接写文件，效率低），查看连接（从内核模块获得）等操作，需要跟内核模块之间通过一定的接口来进行数据交换

课程设计要求

- (1) 系统运行
  - * 系统启动以后插入模块，防火墙以内核模块方式运行
  - * 应用程序读取配置，向内核写入规则，报文到达，按照规则进行处理
- (2) 界面
  - * 采用图形或者命令行方式进行规则配置，界面友好

课程设计任务要求

- (3) 功能要求
  - * 能对TCP、UDP、ICMP协议的报文进行状态分析和过滤
  - * 每一条过滤规则至少包含：报文的源IP(带掩码的网络地址）、目的IP(带掩码的网络地址）、源端口、目的端口、协议、动作（禁止/允许），是否记录日志
  - * 过滤规则可以进行添加、删除、保存，配置的规则能立即生效
  - * 过滤日志可以查看
  - * 具有NAT功能，转换地址分按接口地址转换和指定地址转换（能实现源或者目的地址转换的任一种即可）
  - * 能查看所有连接状态信息

课程设计任务要求

- （4）测试
  - * 测试系统是否符合设计要求
  - * 系统运行稳定
  - * 性能分析

四、课程设计时间与进度安排

- 课程设计总时间为8次课（8天）

序号	阶段内容	时间段
1	布置任务、确定需求	1天
2	系统设计	1-2天
3	详细设计	2天
4	编码调试	3-4天
5	测试	1-2天

- 要求
  - * 充分利用课堂外时间
  - * 课内时间必须到课堂

五、结果检查和成绩评定

- 课设的验收将分为两个部分：
  - 第一部分是成果展示，包括程序演示和讲解
  - 第二部分是提交书面的课设报告
- 本课程采用资格认定和成绩评定双重考核的方式，只有获得资格认定的学生才可进入成绩评定
  - 资格认定
    - 指导教师加强平时考核，对课程设计期间上无关网站、聊天、玩游戏、迟到早退等，每发现一次记一次违规，违规三次算一次未到，三次未到者失去成绩评定的资格，成绩记为0

五、结果检查和成绩评定（续）

* 成绩评定
  - ♦采用百分制评定课程设计成绩
  - ♦成果展示评分（50%，学生互评25%+老师评分25%）
  - ♦课程设计报告（40%）
  - ♦平时考勤（10%）

六、提交资料

- 课程设计报告
  - 包括：需求分析、系统设计、系统测试、结果分析
  - 纸质文档：双面打印
- 电子文档
  - 课程设计报告word版本
  - 所有的源码和实验结果
  - 以班级为单位刻录光盘提交

课程设计原理介绍

课程设计相关

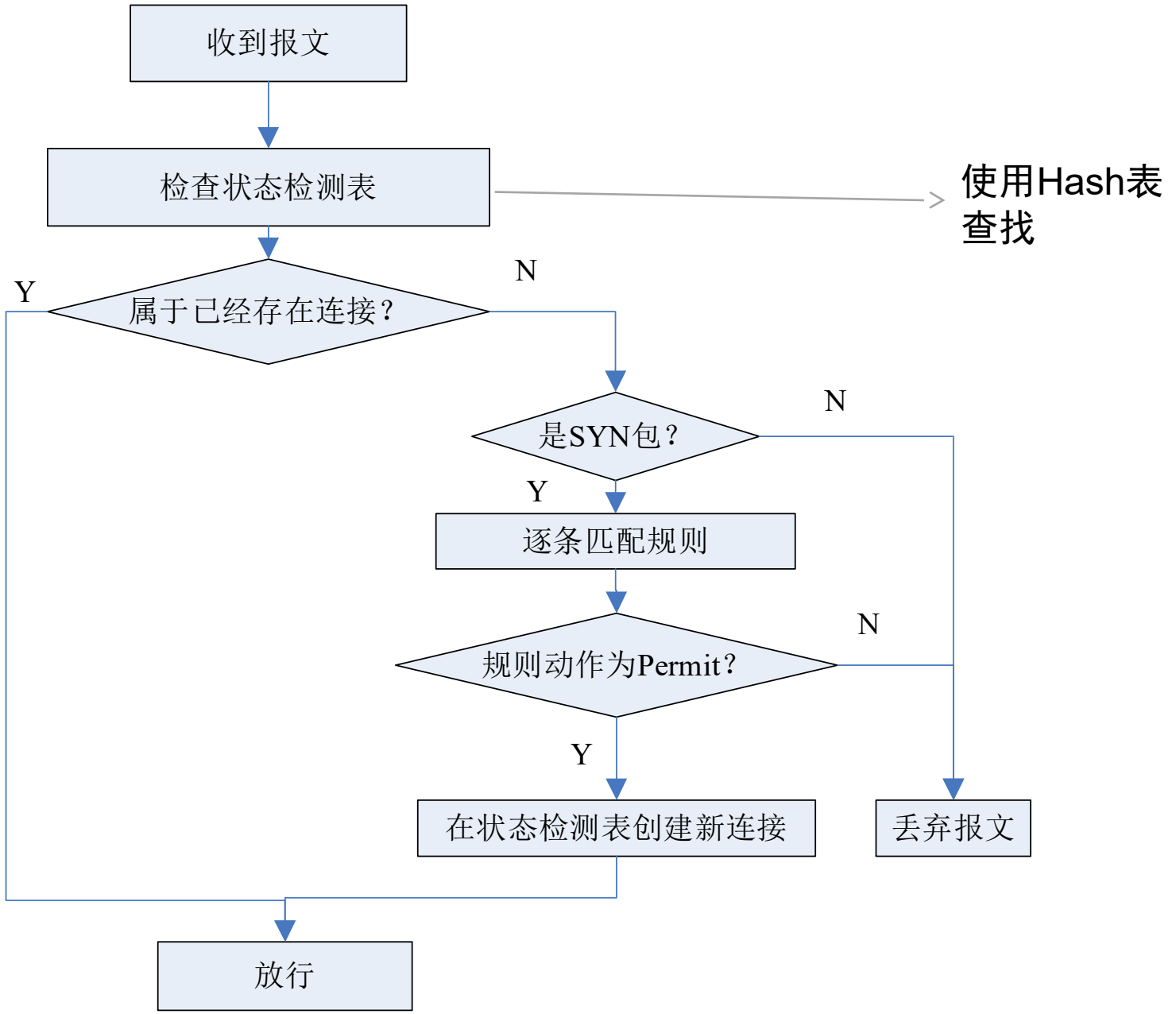
- 1. 状态检测防火墙的工作原理
- 2. Linux Netfilter架构
- 3. Linux内核编程

状态检测技术(Status Detection)

- 思想：
  - * 报文之间存在联系，很多报文属于同一个连接
  - * 可以提高检测效率，仅仅对建立连接的报文进行规则匹配。
  - * 实现连接的跟踪功能，支持动态连接。
    - ♦对于单一连接的协议来说比较简单，根据分组头部的信息进行跟踪；
    - ♦对于复杂协议，除了使用一个公开端口的连接进行通信外，在通信过程中还会动态建立子连接进行数据传输，而子连接的端口信息是在主连接中通过协商得到的随机值，因此，需要对主连接的内容进行分析，无需另加规则。但是需要对不同协议单独处理。(不做要求)

状态检测表（连接表）

- 状态检测表包括所有通过防火墙的连接，并维护所有连接
- 每条连接的信息包括源地址、目的地址、协议类型、协议相关信息（如TCP/UDP协议的端口、ICMP协议的ID号）、连接状态(如TCP连接状态)和超时时间(若进行NAT转换，还需记录转换前后地址端口信息)等，防火墙把这些信息叫做状态。
- 通过状态检测，可实现比简单包过滤防火墙更大的安全性及更高的性能。

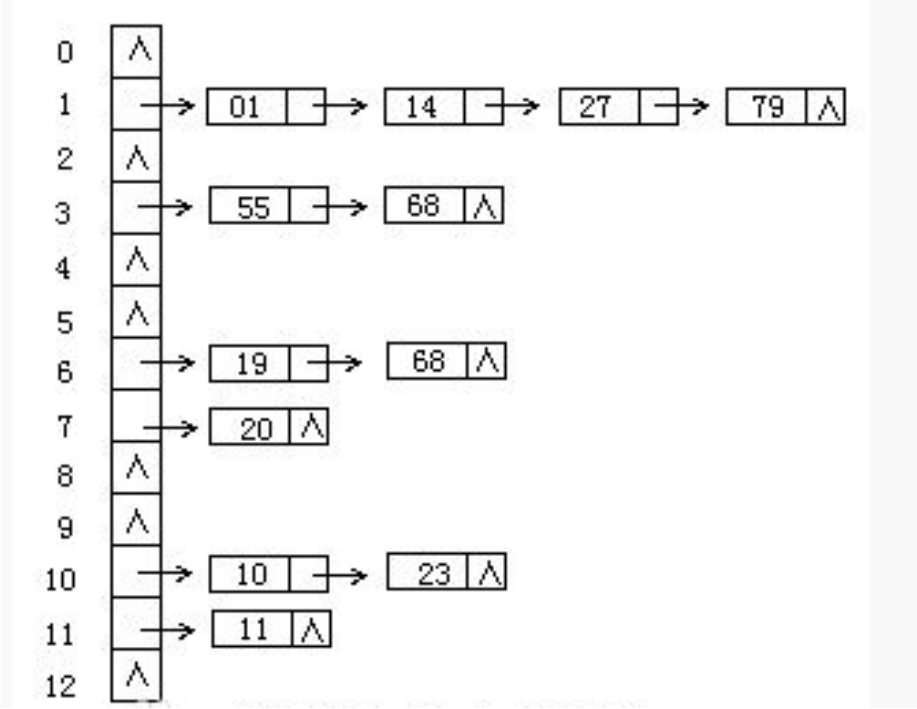

收到报文
检查状态检测表
使用Hash表查找
属于已经存在连接？
Y
N
是SYN包？
N
Y
逐条匹配规则
规则动作为Permit？
N
Y
在状态检测表创建新连接
丢弃报文
放行

非TCP连接

- 防火墙为其建立虚拟连接
- UDP
  - 一方发出UDP包后，如DNS请求，防火墙会将从目的地址和端口返回的、到达源地址源端口的包作为状态相关的包而允许通过。
- ICMP
  - 信息查询报文：如ping(echo)包，其状态相关包就是来自目的地址的echo reply包，而没有与echo包对应的echo reply包则认为是状态非法的。
- 这些连接在防火墙的状态检测表中的超时时间比较短。

状态检测表

- 非线性数据结构：hash表、树等；
- 直接在内核执行匹配，速度快；
- 采用硬件支持，速度更快。



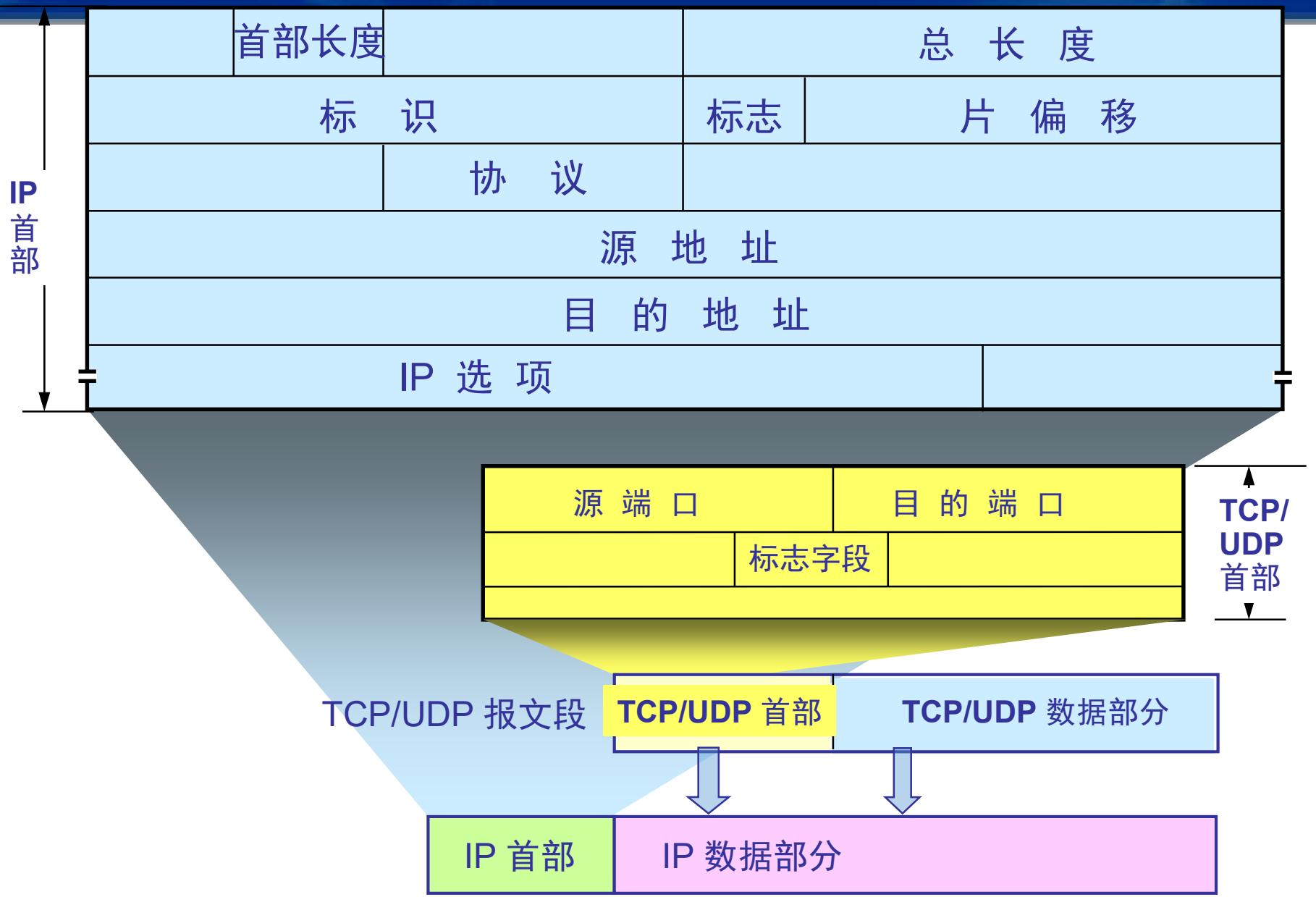
状态检测防火墙的优缺点

- 优点
  - 不需要对每个数据包进行规则检查，而是一个连接的后续数据包（通常是大量的数据包）通过散列（hash）算法，直接进行状态检查，从而使得性能得到了较大提高；
  - 由于状态表是动态的，因而可以有选择地、动态地开通1024号以上的端口，使得安全性得到进一步地提高。
- 缺点
  - 无法彻底的识别数据包中大量的垃圾邮件、广告以及木马程序等等。

默认策略

- 两种默认策略：
  - **默认接受**：是指除非明确地指定禁止某个分组，否则分组是可以通过的。
  - **默认拒绝**：除非明确地指定允许某个分组通过，否则分组是不可以通过的。
  - **从安全的角度讲，默认拒绝应该是更安全的**。

数据包结构——IP、TCP/UDP



数据包结构——ICMP

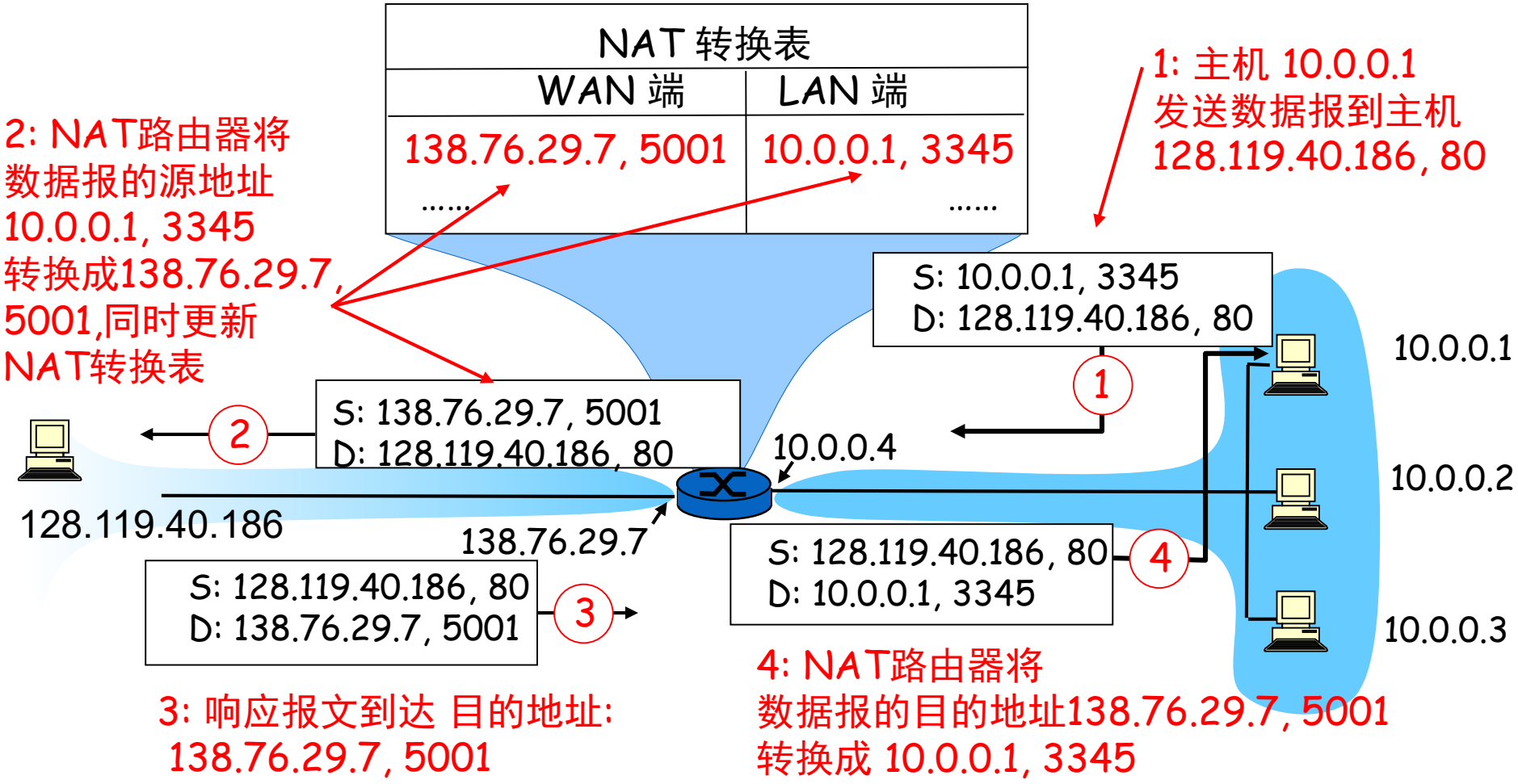

前 4 个字节
都是一样的
0
8
16
31
类型
代码
检验和
（这 4 个字节取决于 ICMP 报文的类型）
ICMP 的数据部分（长度取决于类型）
ICMP 报文

防火墙规则

源地址	源端口	协议	目的地址	目的端口	日志	动作
192.168.1.0/24	ANY	TCP	192.168.3.3	25,110	no	Permit
ANY	ANY	ANY	192.168.1.0/24	ANY	yes	Reject
ANY	ANY	TCP	192.168.3.2	1358	yes	Reject
ANY	ANY	TCP	192.168.3.4	80	no	Permit
192.168.1.0/24	ANY	ANY	ANY	ANY	no	Permit
ANY	ANY	ANY	ANY	ANY	yes	Reject

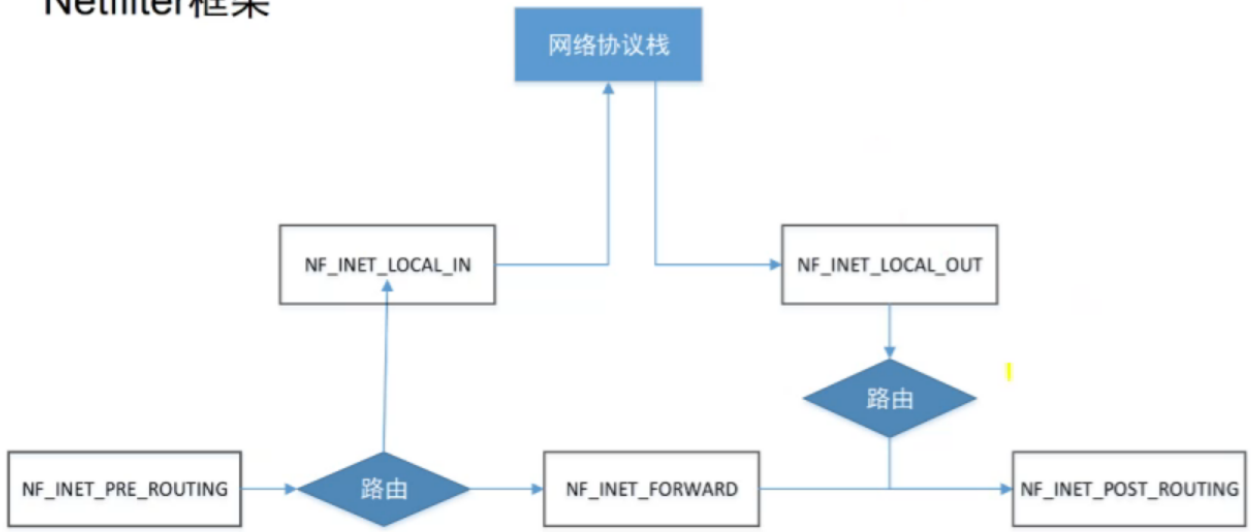
支持NAT的话，规则设置还需要设置是
否进行NAT，并设置NAT地址或接口

NAT转换实例—动态NAT



包过滤模块在linux内核中的位置

Netfilter框架



经路由查找后，送往本机的通过此检查点，INPUT包过滤在此点进行

本机进程发出的包通过此检测点，OUTPUT包过滤在此点进行

刚刚进入网络层的数据包通过此点（刚刚进行完版本号、校验和等检测），目的地址转换在此点进行

NF_INET_LOCAL_IN

要转发的包经过此检测点，FORWARD包过滤在此点进行

所有马上便要通过网络设备出去的包通过此检测点，内置的源地址转换功能（包括地址伪装）在此点进行

NF_INET_PRE_ROUTING

路由

NF_INET_FORWARD

NF_INET_POST_ROUTING

Netfilter hook 点

Hook点	调用的时机
NF_INET_PRE_ROUTING	刚刚进入网络层的数据包通过此点（刚刚进行完版本号、校验和等检测），目的地址转换在此点进行。
NF_INET_LOCAL_IN	经路由查找后，送往本机的通过此检查点，INPUT包过滤在此点进行
NF_INET_FORWARD	要转发的包经过此检测点，FORWARD包过滤在此点进行
NF_INET_LOCAL_OUT	本机进程发出的包通过此检测点，OUTPUT包过滤在此点进行
NF_INET_POST_ROUTING	所有马上便要通过网络设备出去的包通过此检测点，内置的源地址转换功能（包括地址伪装）在此点进行

Hook函数返回值

- 在hook函数完成了对数据包所需的任何的操作之后，它们必须返回下列预定义的Netfilter返回值中的一个：

返回值	含义
NF_DROP	丢弃该数据包
NF_ACCEPT	保留该数据包
NF_STOLEN	告知netfilter忽略该数据包
NF_QUEUE	将该数据包插入到用户空间
NF_REPEAT	请求netfilter再次调用该HOOK函数

内核模块插入、卸载函数

```
static void kexec_test_exit(void)
{
    printk("kexec test exit ...\n");
    nf_unregister_hook(&nfho);
}

module_init(kexec_test_init);
module_exit(kexec_test_exit);
```

内核模块编译

- Makefile编写

```
KERN_DIR = /lib/modules/$(shell uname -r)/build
mymodule-objs := file1.o file2.o
obj-m += mymodule.o

all:
    make -C $(KERN_DIR) M=$(shell pwd) modules
clean:
    make -C $(KERN_DIR) M=$(shell pwd) modules clean
    rm -rf modules.order
```

- 编译
- 插入模块：insmod mymodule.ko
- 查看模块：lsmod
- 卸载模块：rmmod mymodule

（1）注册和注销Netfilter hook

```
struct nf_hook_ops {
        struct list_head list;  //用于维护netfilter的hook列表

        /* User fills in from here down. */
        nf_hookfn *hook;  //指向nf_hookfn类型的函数的指针，该函数是这个hook被调用时执行的函数
        struct net_device       *dev;
        void                    *priv;
        u_int8_t pf;       //指定协议族,对于IPv4协议族PF_INET
        unsigned int hooknum;  //指定安装的这个函数对应的具体的hook类型
        /* Hooks are ordered in ascending priority. */
        int priority;  //指定执行的顺序，统一hook可能hook多个函数
};
```

注册一个Netfilter hook需要调用nf_register_hook()函数

示例代码1 ： Netfilter hook的注册

```
/*
* 安装一个丢弃所有到达的数据包的Netfilter hook函数的示例代码
*/
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/skbuff.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("xsc");

static struct nf_hook_ops nfho;
```

```
unsigned int hook_func(void *priv,
                       struct sk_buff *skb,
                       const struct nf_hook_state *state)
{
    return NF_DROP;
}

static int kexec_test_init(void)
{
    printk("kexec test start ...\n");
    nfho.hook = hook_func;
    nfho.pf = PF_INET;
    nfho.hooknum = NF_INET_LOCAL_OUT;  //新版本内核红色字体部分由IP改成了INET
    nfho.priority = NF_IP_PRI_FIRST;

    nf_register_hook(&nfho);                    // 注册一个钩子函数
    return 0;
}
```

（2） Netfilter 基本的数据报过滤技术

```
unsigned int hook_func(void *priv,
                       struct sk_buff *skb,
                       const struct nf_hook_state *state)
```

1）priv私有指针

2）skb指针指向sk_buff数据结构，网络堆栈用sk_buff数据结构来描述数据包。这个数据结构在linux/skbuff.h中定义。

sk_buff数据结构中最有用的部分就是那三个描述传输层包头、网络层包头以及链路层包头的联合(union)了。这三个联合的名字分别是h、nh以及mac

3）state指针记录包的一些状态，包括接收接口、转发接口等信息；

基于接口进行过滤

```
/* 我们丢弃的数据包来自的接口的名字 */
static char *drop_if = "lo";
unsigned int hook_func(void *priv,
                       struct sk_buff *skb,
                       const struct nf_hook_state *state)
{
    /* state->in为报文接收接口，state->out为报文转发接口*/
    if ( (state->out != NULL) && (strcmp(state->out->name, drop_if)==0) ){
        printk("Dropped packet on %s..\n",drop_if);
        return NF_ACCEPT;
    } else{
        return NF_DROP;
    }
}
```

基于地址进行过滤

```
static char *parg = "220.181.111.147";
unsigned int hook_func(void *priv,
                       struct sk_buff *skb,
                       const struct nf_hook_state *state)
 {
        struct iphdr *ip;    //ip头结构指针
        if (!skb)
                return NF_ACCEPT;
        ip = ip_hdr(skb);
        if(ip->saddr == inet_addr(parg))      /// 目标IP地址
                return NF_DROP;
        else
                return NF_ACCEPT;
}
```

基于TCP端口过滤

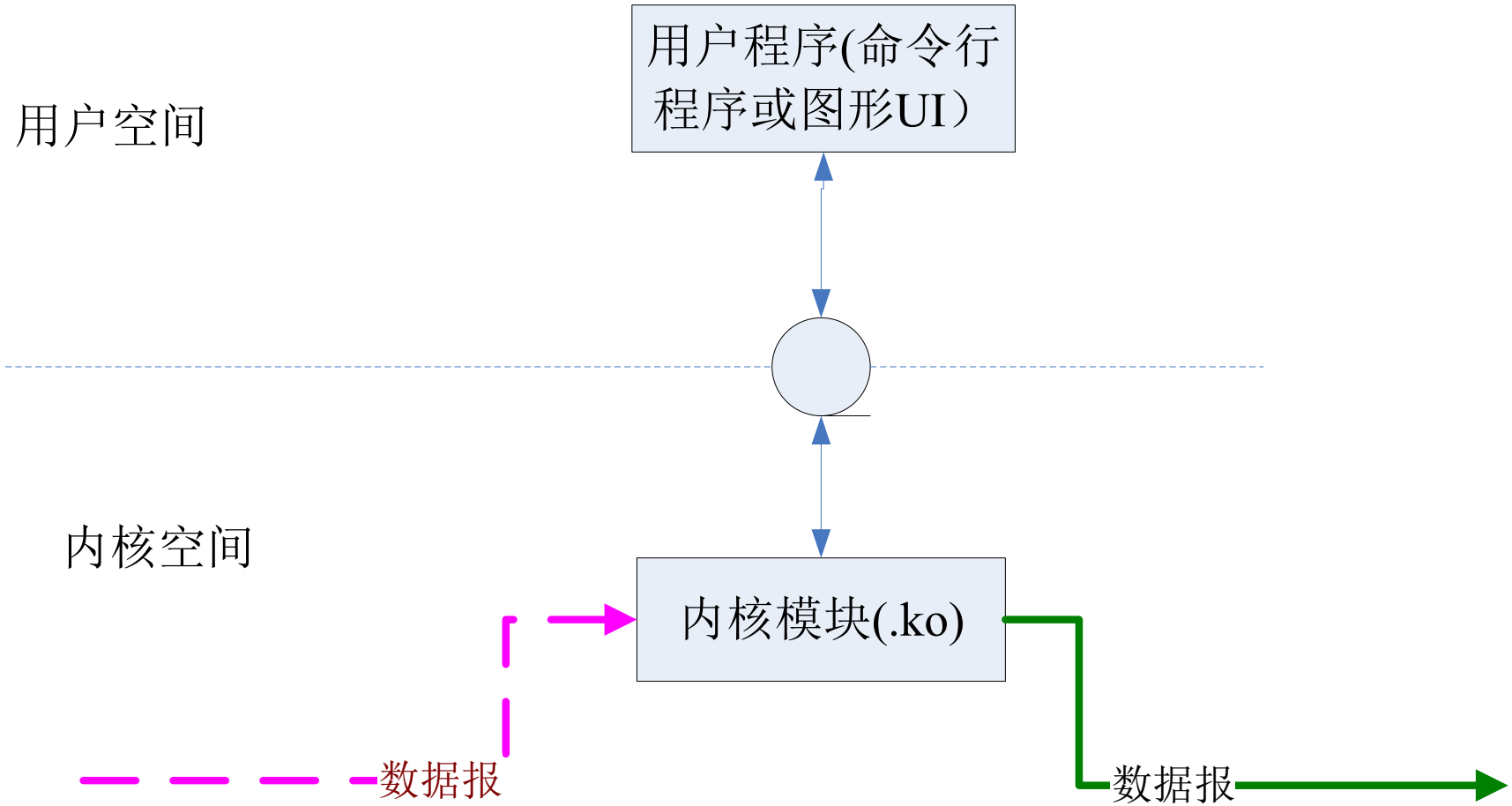
```
static int check_tcp_packet(struct sk_buff *skb)
{
        struct tcphdr *tcph = NULL;
         const struct iphdr *iph = NULL;
         struct iphdr *ip;
         __be16 dport;

         if (!skb)
                return NF_ACCEPT;
         iph = ip_hdr(skb);
         if(iph->protocol == IPPROTO_TCP){                              /// TCP 协议
             tcph = (void *) iph + iph->ihl * 4;                       /// TCP 包头
             dport = tcph->dest;                                       /// 目标端口
             if(ntohs(dport) == 25 ){
                     return NF_DROP;
             }else
                     return NF_ACCEPT;
         }
         return NF_ACCEPT;
}
```

3. 用户空间跟内核空间交互

- Linux 操作系统和驱动程序运行在内核空间（比如:报文过滤的内核模块），应用程序（比如：配置规则的程序）运行在用户空间，两者不能简单地使用指针传递数据，因为Linux使用的虚拟内存机制，用户空间的数据可能被换出，当内核空间使用用户空间指针时，对应的数据可能不在内存中。

用户空间和内核空间



内核空间和用户空间信息交互方式

- 1）编写自己的系统调用
- **2）编写设备驱动程序**
  - 驱动程序运行于内核空间，用户空间的应用程序通过文件系统中/dev/目录下的一个文件（字符设备文件）来和它交互。这就是我们熟悉的那个文件操作流程：open() —— read() —— write() —— ioctl() —— close()。
  - 驱动程序也是用户空间和内核信息交互的重要方式之一。其实ioctl，read，write本质上讲也是通过系统调用去完成的，只是这些调用已被内核进行了标准封装，统一定义。
- **3）使用proc文件系统**
  - proc是Linux提供的一种特殊的文件系统，推出它的目的就是提供一种便捷的用户和内核间的交互方式。
- **4）使用netlink**
- 5）使用内存映像
  - 内存影射方式通常也正是应用在那些内核和用户空间需要快速大量交互数据的情况下，特别是那些对实时性要求较强的应用。

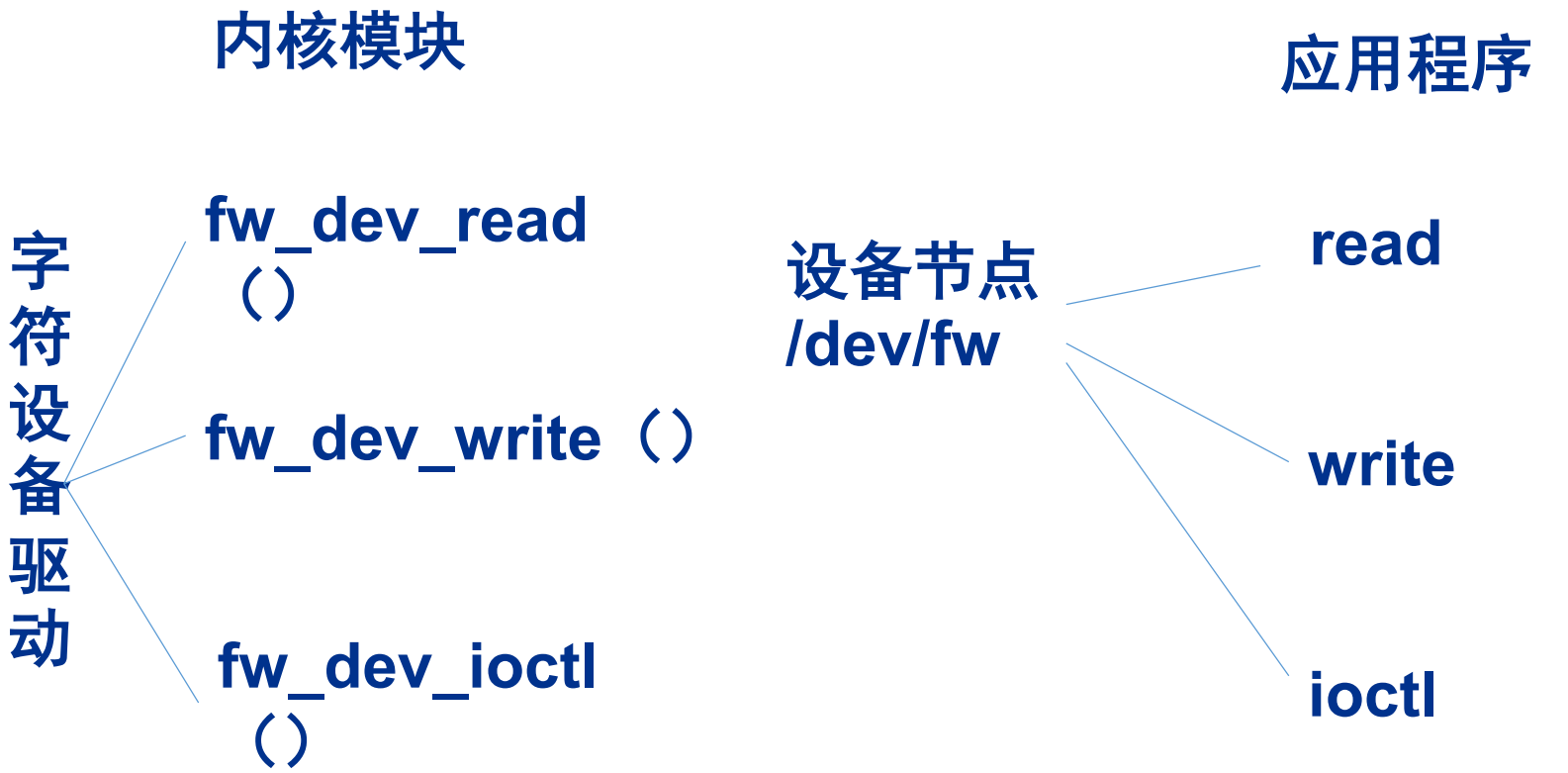
- **推荐采用2）、3）、4）方案**，例子代码采用的第2方案

- 参考资料中采用的第二种方式：通过**字符设备**来进行交互
- 1）定义file_operations结构

```
struct file_operations  lwfw_fops = {
        .owner = THIS_MODULE,
        .unlocked_ioctl = lwfw_ioctl,
        .open = lwfw_open,
        .write = lwfw_write, //可以指定相应的函数
        .release = lwfw_release,
};
```

- 2）在/dev下用**mknod命令创建设备节点**
- 3）字符设备创建与注册：
  https://blog.csdn.net/weixin_42314225/article/details/81112217

用户程序与内核通过设备节点交互的过程



在应用程序从设备文件/dev/fw读取数据时，触发了内核的fw_dev_read函数；依次类推，应用程序向/dev/fw写数据时，内核对应的处理函数是fw_dev_write，它们之间通过函数参数中的缓冲区来交换数据

简单字符设备示例

```
/*
 *  devDrv.c
 *  threeProcess
 *  Created on: 2012-2-24
 *      Author: zhushengben
 */

#include "linux/kernel.h"
#include "linux/module.h"
#include "linux/fs.h"
#include "linux/init.h"
#include "linux/types.h"
#include "linux/errno.h"
#include "linux/uaccess.h"
#include "linux/kdev_t.h"
#define MAX_SIZE 1024
```

```
static int my_open(struct inode *inode, struct file *file);
static int my_release(struct inode *inode, struct file *file);
static ssize_t my_read(struct file *file, char __user *user, size_t t, loff_t *f)
;
static ssize_t my_write(struct file *file, const char __user *user, size_t t, lof
f_t *f);

static char message[MAX_SIZE] = "-------congratulations--------!";
static int device_num = 0;//设备号
static int counter = 0;//计数用
static int mutex = 0;//互斥用
static char* devName = "myDevice";//设备名

struct file_operations pStruct =
{ open:my_open, release:my_release, read:my_read, write:my_write, };
```

```
/* 注册模块，内核模块的入口函数，相当于应用程序的main函数 */
int init_module()
{
    int ret;
    /* 函数中第一个参数是告诉系统，新注册的设备的主设备号由系统分配，
     * 第二个参数是新设备注册时的设备名字，
     * 第三个参数是指向file_operations的指针，
     * 当用设备号为0创建时，系统一个可以用的设备号创建模块 */
    ret = register_chrdev(0, devName, &pStruct);
    if (ret < 0)   {
        printk("regist failure!\n");
        return -1;
    }
    else   {
        printk("the device has been registered!\n");
        device_num = ret;
        printk("<1>the virtual device's major number %d.\n", device_num);
        printk("<1>Or you can see it by using\n");
        printk("<1>------more /proc/devices-------\n");
        printk("<1>To talk to the driver,create a dev file with\n");
        printk("<1>------'mknod /dev/myDevice c %d 0'-------\n", device_num);
        printk("<1>Use \"rmmode\" to remove the module\n");
        return 0;
    }
}
```

```
/* 注销模块，函数名很特殊 */
void cleanup_module()
{
    unregister_chrdev(device_num, devName);
    printk("unregister it success!\n");
}

static int my_open(struct inode *inode, struct file *file)
{
    if (mutex)
        return -EBUSY;
    mutex = 1;//上锁
    printk("<1>main  device : %d\n", MAJOR(inode->i_rdev));
    printk("<1>slave device : %d\n", MINOR(inode->i_rdev));
    printk("<1>%d times to call the device\n", ++counter);
    try_module_get(THIS_MODULE);
    return 0;
}
```

```
/* 每次使用完后会release */
static int my_release(struct inode *inode, struct file *file)
{
    printk("Device released!\n");
    module_put(THIS_MODULE);
    mutex = 0;//开锁
    return 0;
}
static ssize_t my_read(struct file *file, char __user *user, size_t t, loff_t *f)
{
    if(copy_to_user(user,message,sizeof(message)))    //copy_to_user从内核向用户空间拷贝数据
        return -EFAULT;
    return sizeof(message);
}
static ssize_t my_write(struct file *file, const char __user *user, size_t t, loff_t *f)
{
    if(copy_from_user(message,user,sizeof(message)))  //copy_from_user从用户空间向内核拷贝数据
        return -EFAULT;
    return sizeof(message);
}
```

Netlink用户空间与内核空间通信

- Netlink套接字是用以实现用户进程与内核进程通信的一种特殊的进程间通信(IPC)
- 网络应用程序与内核通信的最常用的接口之一
- 用户态应用使用标准的 socket API 就能使用 Netlink 提供的强大功能
  - socket(),bind(),sendmsg(),recvmsg(),close()
- 内核态需要使用专门的内核 API 来使用 Netlink

netlink内核部分

- 模块初始化时，创建 netlink socket

  struct sock *netlink_kernel_create(struct net *net, int unit, struct netlink_kernel_cfg *cfg)

  - (1)net：网络名字空间namespace，默认使用 init_net
  - (2)unit：表示netlink协议类型，如自定义的NETLINK_TEST
  - (3)cfg：是netlink的配置信息，结构体定义如下：

```
struct netlink_kernel_cfg {
        unsigned int groups;
        unsigned int flags;
        void (*input)(struct sk_buff *skb);
        struct mutex *cb_mutex;
        int (*bind)(struct net *net, int group);
        void (*unbind)(struct net *net, int group);
        bool (*compare)(struct net *net, struct sock *sk);
};
```

  - 其中最重要的是input域，该函数用于处理用户空间发送到内核方向的数据，内核接收到数据包后，会调用input函数。

netlink内核部分

- 消息接收，input函数被调用
  - netlink消息头结构指针

    ```
    struct nlmsghdr *nlh;
    ```
  - 提取netlink消息

    ```
    nlh = nlmsg_hdr(skb);
    ```
  - 消息长度

    ```
    nlh->nlmsg_len
    ```
  - 用户进程id

    ```
    nlh->nlmsg_pid
    ```
  - 提取消息数据部分

    ```
    (char *) NLMSG_DATA(nlh)
    ```

netlink内核部分

- 发送应答消息
  - 创建netlink消息报文

```
skb = nlmsg_new(rlen, GFP_ATOMIC);
```

  - 设置netlink消息

```
nlh = nlmsg_put(skb, 0, 0, 0, NLMSG_SPACE(rlen) -
NLMSG_HDRLEN, 0);
```

  - 设置消息数据
  - 发送消息

```
netlink_unicast(nlsk, skb, pid, MSG_DONTWAIT);
```

    - pid：之前接收到的消息的用户进程id

netlink应用程序

- 创建套接字

```
skfd = socket(PF_NETLINK, SOCK_RAW,
NETLINK_TEST);
```

- 绑定本地地址

```
local.nl_family = AF_NETLINK;
local.nl_pid = getpid();
local.nl_groups = 0;
bind(skfd, (struct sockaddr *) &local,
sizeof(local))
```

netlink应用程序

- 设置netlink消息

```
message->nlmsg_len = NLMSG_SPACE(dlen);
message->nlmsg_pid = local.nl_pid;
```

- 发送消息至内核

```
sendto(skfd, message, message->nlmsg_len, 0,
(struct sockaddr *) &kpeer, sizeof(kpeer));
```

- 接收内核应答消息

```
recvfrom(skfd, &info, sizeof(struct
u_packet_info), 0, (struct sockaddr *) &kpeer,
&kpeerlen);
```

课程设计任务分解

- 任务1：测试模块编译完成，插入模块、卸载模块（1小时）
- 任务2：注册netfilter hook函数，注册字符设备节点（如果应用程序和内核用字符设备通信的话），在hook函数中打印出报文信息（1天）
- 任务3：设计规则表结构和日志格式，在hook函数中对报文进行过滤，规则可以在内核模块中用数组指定（1天）
- 任务4：实现用户程序，以及用户程序跟内核模块之间的交互，规则能够写入内核，内核维护规则链表，用户程序可以从内核读取已经生效的规则信息和日志信息（3天）
- 任务5：修改内核模块的过滤流程，实现状态检测，维护状态表，状态表预留NAT的信息（1天）
- 任务6：实现NAT功能（2天）
- 功能完善与稳定（1-2天）
- 程序展示（另行安排）

内核调试方法

- printk 打印调试语句，如果没有在屏幕输出，可以用dmesg 命令查看内核打印信息
- 分段调试，排除问题
- 内核指针出错，很容易造成系统崩溃，需要重启系统才能恢复，所以需要及时进行代码保存与备份

一些说明

- 1. 《深入linux网络核心堆栈.docx》中的例子代码适用于linux 3.x的内核版本，如果内核是4.x的版本，需要做部分修改（任务书中netfilter的接口已经修改为4.x下的）；
- 2. 例子代码仅仅是演示了地址如何过滤，端口如何过滤，并不等同于课设要求的规则，课设要求的每一条规则至少包括五元组、还有动作、日志选项等等；单项控制认为是例子代码本身具有的功能，不予认可；
- 3. 内核模块调试过程中很容易系统崩溃，崩溃以后需要重新启动系统，所以务必做好备份工作；

一些说明

- 4.my_firewall.tgz中是一个最简单的防火墙内核模块，以及Makefile
  - 运行tar xzvf my_firewall.tgz将my_firewall.tgz解压缩到linux下，直接make，会生成myfw.ko，即为生成的内核模块文件；
  - insmod myfw.ko 插入模块，会丢掉本机外出的目的端口为tcp 23端口的报文
  - lsmod 列出所有内核模块，可以看到自己的模块
  - rmmod myfw 卸载模块（注意卸载时不带.ko后缀），网络通信恢复正常

一些说明

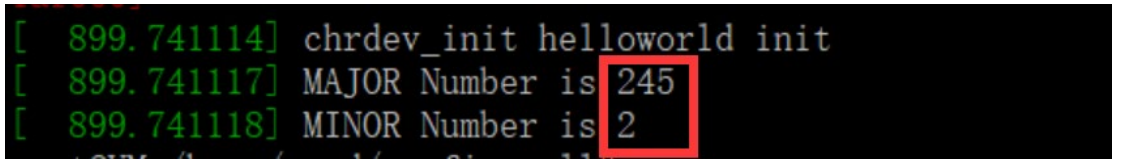
- 5.my_dev.c是另一个测试代码
  - ＊修改Makefile，将Makefile中的

  myfw-objs := my_firewall.o #my_dev.o

  改成

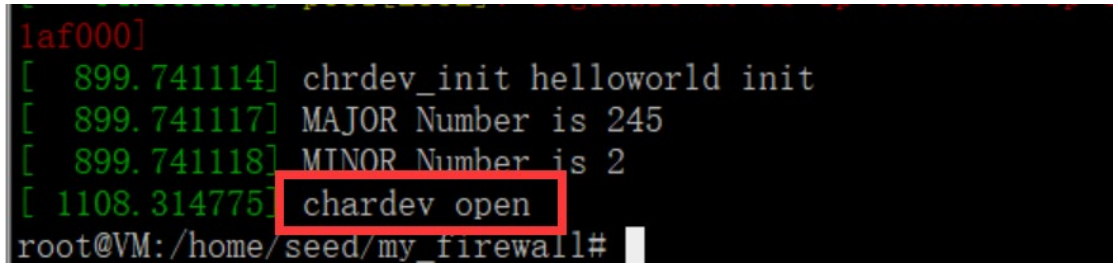
  myfw-objs := my_dev.o #my_firewall.o

  就可以编译该模块了
  - ＊这个模块的功能是注册了一个字符设备，插入模块以后，可以通过/proc/devices查看到该字符设备，名字为chardev_test_byHc
  - ＊运行dmesg的最后两行可以查看到该设备的主设备号和次设备号



一些说明

* mknod /dev/your_name c 245 2 就可以对该设备节点进行操作了，设备节点文件名任意，主设备号和次设备号不能错误。
* cat /dev/your_name后，dmesg可以查看到打印了my_dev.c中字符设备的open函数中的调试信息



一些说明