# 基于 netfilter 的 Linux 系统防火墙

[基于 netfilter 的 Linux 系统防火墙](#基于-netfilter-的-linux-系统防火墙)  
 [功能完成](#功能完成)  
 [立项意义](#立项意义)  
 [项目概要](#项目概要)  
 [整体架构](#整体架构)  
 [设计思路](#设计思路)  
 [开发环境](#开发环境)  
 [详细设计](#详细设计)  
 [关键数据结构与变量](#关键数据结构与变量)  
 [NETLINK](#netlink)  
 [NETFILTER](#netfilter)  
 [包过滤](#包过滤)  
 [连接状态](#连接状态)  
 [NAT 网络地址转换](#nat-网络地址转换)  
 [web 管理面板](#web-管理面板)  
 [数据库设计](#数据库设计)  
 [前端页面设计](#前端页面设计)  
 [后台接口编写](#后台接口编写)  
 [项目测试](#项目测试)  
 [心得体会](#心得体会)

## 功能完成

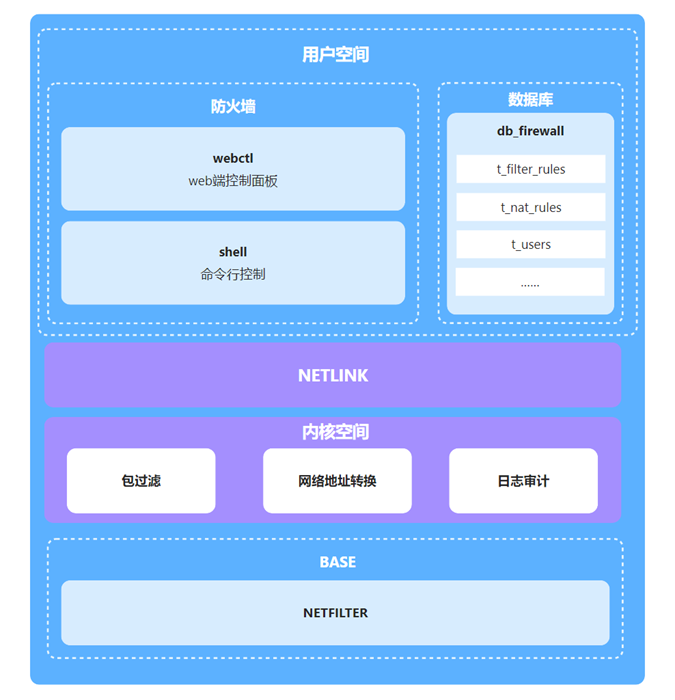
* 用户层
* 内核层
* 通信
* 规则过滤
* 规则过滤 web 面板
* 连接管理
* 连接管理 web 面板
* 状态防火墙
* NAT
* NAT web 面板
* 日志记录
* 日志审计 web 面板
* 基础 DOS 防御
* 远程登录 web 面板

## 立项意义

## 项目概要

### 整体架构

* Web 端控制面板
* 使用数据库存储过滤规则表和 NAT 规则表等数据，使用 B/S 架构直接对防火墙行为进行控制
* 命令行管理工具
* 可以使用提供的用户态的命令行工具进行防火墙的过滤规则和 NAT 规则的配置
* 内核驱动模块
* 在内核基于 NETFILTER 实现包过滤、NAT 等功能



### 设计思路

1、**Node + MySQL**

Web 控制面板使用用户程序提供的命令行管理语法实现便利的的 B/S 架构 UI

2、**用户态程序**

用户程序向下使用内核驱动模块提供的接口，向上为 web 控制面板提供防火墙管理接口

3、**NETLINK 套接字**

使用 NETLINK 套接字和自定义的协议实现用户空间与内核空间的数据交换

4、**NETFILTER 框架**

使用 NETFILTER 提供的底层接口编写相关的内核驱动模块

### 开发环境

* 操作系统：Linux 5.15.0-89-generic #99~20.04.1-Ubuntu SMP Thu Nov 2 15:16:47 UTC 2023 x86\_64 x86\_64 x86\_64 GNU/Linux
* C 编译器：gcc (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0
* 构建工具：GNU Make 4.2.1
* Node 运行环境：node v10.19.0
* 开发工具：visual studio code 1.85.0

## 详细设计

### 关键数据结构与变量

用户请求类型和结构体如下，tp 为请求的类型。

使用了 union关键字来定义结构体 UsrReq成员 msg。可以实现在相同的内存空间中节省存储空间，因为每次请求只需要一种数据，因此只需要存储其中一种类型的数据。

// tp请求类型  
#define REQ\_GETAllFTRULES 1 // 获取所有过滤规则  
#define REQ\_ADDFTRULE 2 // 添加过滤规则  
#define REQ\_DELFTRULES 3 // 删除过滤规则  
#define REQ\_SETACT 4 // 设置行为  
#define REQ\_GETAllLOGS 5 // 获取所有日志  
#define REQ\_GETAllCONNS 6 // 获取所有网络连接  
#define REQ\_ADDNATRULE 7 // 添加网络地址转换规则  
#define REQ\_DELNATRULE 8 // 删除网络地址转换规则  
#define REQ\_GETNATRULES 9 // 获取所有网络地址转换规则  
  
struct UsrReq  
{  
 unsigned int tp;  
 char ruleName[MAXRuleNameLen + 1];  
 // 请求体——过滤规则、NAT规则、默认动作  
 union  
 {  
 struct FTRule FTRule;  
 struct NATRule NATRule;  
 unsigned int defaultAction;  
 unsigned int num;  
 } msg;  
};

内核响应状态码和响应结构体如下，stat 为内核返回第状态码。data 存储返回时数据，header 指针指向 data 中的头部，body 指针指向 data 中的数据载荷。

// stat  
#define RSP\_NULL 10  
#define RSP\_MSG 11  
#define RSP\_FTRULES 12 // body为过滤规则  
#define RSP\_FTLOGS 13 // body为日志  
#define RSP\_NATRULES 14 // body为nat规则  
#define RSP\_CONNLOGS 15 // body为连接  
  
struct KernelResp  
{  
 int stat;  
 void \*data;  
 struct KernelResHdr \*header;  
 void \*body;  
};

过滤规则的结构体如下，包括 rulename、源地址、目的地址、源端口、目的端口、协议、策略、是否记录日志几个参数。

struct FTRule  
{  
 char name[MAXRuleNameLen + 1];  
 unsigned int saddr;  
 unsigned int smask;  
 unsigned int taddr;  
 unsigned int tmask;  
 unsigned int sport;  
 unsigned int tport;  
 u\_int8\_t protocol;  
 unsigned int act;  
 unsigned int islog;  
 struct FTRule \*next;  
};

NAT 规则结构体如下

struct NATRule  
{  
 unsigned int saddr; // 源IP  
 unsigned int smask; // 源IP的掩码  
 unsigned int daddr; // 转换后的IP  
 unsigned short sport; // 原始端口  
 unsigned short dport; // 转换后的端口  
 unsigned short nowPort; // 当前使用的端口  
 struct NATRule \*next;  
};

连接会话的结构体如下

struct connSess  
{  
 struct rb\_node node;  
 conn\_key\_t key; // 连接标识符  
 unsigned long expires; // 超时时间  
 u\_int8\_t syn; // 记录syn数  
 u\_int8\_t rate; // 记录数据包到达的速率  
 u\_int8\_t protocol; // 协议  
 u\_int8\_t needLog; // 是否记录日志  
 struct NATRule nat; // 该连接对应的NAT记录  
 int natType; // NAT 转换类型  
};

过滤规则链表和过滤规则表的读写自旋锁如下。前者维护过滤规则表，后者可以保证规则表可以同时被多个进程读取但不能同时被多个进程写入，用于保证规则表并发安全。

// 规则链表  
static struct FTRule \*FTRuleHd = NULL;  
// 规则链表锁，保证并发安全  
static DEFINE\_RWLOCK(FTRuleLock);

NAT 规则表和 NAT 规则表的读写自旋锁。

// NAT规则链表  
static struct NATRule \*NATRuleHd = NULL;  
// NAT规则链表锁，保证并发安全  
static DEFINE\_RWLOCK(NATRuleLock);

利用红黑树存储连接会话。红黑树是一种自平衡的二叉搜索树，它可以高效地查找数据。由于对于连接会话来说，数据包到达的频率远远大于新连接建立的频率，因此使用红黑树存储连接会话，能够大大降低数据包到达时匹配连接会话的时间复杂度。

同理，connLock 为连接会话表的读写自旋锁。

为什么不用红黑树来存储过滤规则呢？这样不是更快吗？

在开发的时候也考虑过使用红黑树来存储过滤规则，但是后来考虑到过滤规则具有先后顺序，应该返回匹配到的第一条规则，那么红黑树就不是最合适的数据结构，如果为了提高效率而增加配置维护的难度和降低安全性，那就是舍本逐末了。因此不使用红黑树来存储过滤规则。

// 红黑树和读写锁  
static struct rb\_root connRoot = RB\_ROOT;  
static DEFINE\_RWLOCK(connLock);

下面的六个钩子点，分别用于

* 本地入站规则检查
* 本地出站规则检查
* 预路由阶段处理数据包，处理 DNAT
* 预路由阶段过滤数据包，建立 NAT 连接
* 后路由阶段处理数据包，处理 SNAT
* 后路由阶段过滤数据包，建立 NAT 连接

static struct nf\_hook\_ops NF\_HKLocalIn;  
static struct nf\_hook\_ops NF\_HKLocalOut;  
static struct nf\_hook\_ops NF\_HKPreRouting;  
static struct nf\_hook\_ops NF\_HKPreRouting2;  
static struct nf\_hook\_ops NF\_HKPostRouting;  
static struct nf\_hook\_ops NF\_HKPostRouting2;

对数据包的默认策略，这个变量定义了对未匹配到规则的数据包的默认策略。

// 设置默认动作  
unsigned int DEFAULT\_ACTION = NF\_ACCEPT;

下面的两个变量用于在内核创建一个 sock 套接字并配置 netlink 内核套接字的参数。创建一个 netlink\_kernel\_cfg结构体变量，配置 netlink 内核套接字的行为和回调函数。

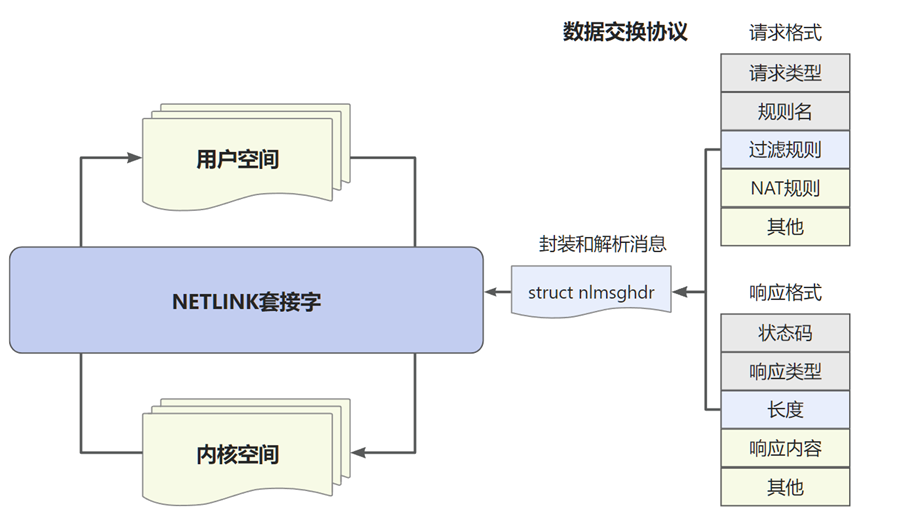
// 创建一个套接字结构  
static struct sock \*nl\_sock = NULL;  
  
struct netlink\_kernel\_cfg nl\_conf = {  
 .groups = 0,  
 .flags = 0,  
 // 指定了一个回调函数NLFWRecv，用于在接收到 netlink 消息时进行处理  
 // void (\*input)(struct sk\_buff \*skb);  
 .input = NLFWRecv,  
 .cb\_mutex = NULL,  
 .bind = NULL,  
 .unbind = NULL,  
 .compare = NULL,  
};

### NETLINK

使用 NETLINK 套接字实现用户空间与内核空间的数据交换。

整体通信的流程如下所示。

用户空间与内核空间的通信通过Netlink Socket实现，在此之上自定义一套数据交换协议。使用sendto()函数向内核发送消息，recvmsg()函数从内核接收消息。在内核与用户空间中对请求与响应进行解析。协议的请求格式和相应格式见先前的关键数据结构与变量部分。

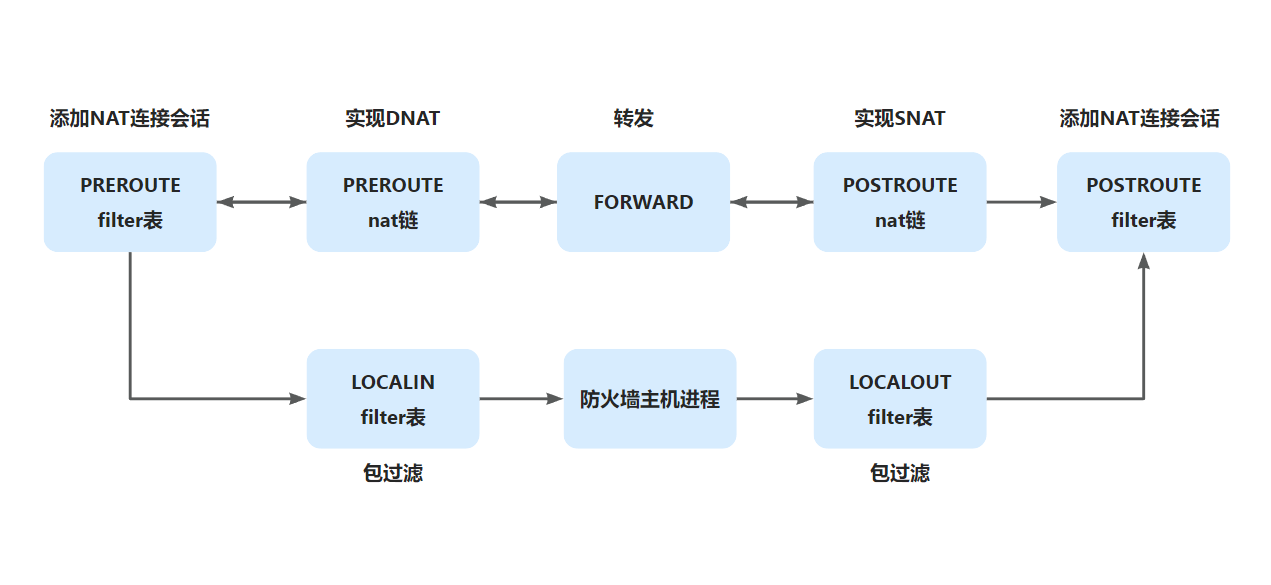


通过netlink完成用户空间与内核空间的通信，实现过滤规则的添加、删除、查看，默认策略的设置，连接会话表的检查，NAT规则的添加、查看与删除等操作，能够在用户空间管理内核驱动程序。

### NETFILTER

本项目拟利用netfilter提供的底层的**hook点**，在此之上实现**内核防火墙驱动模块**和**用户空间的防火墙控制程序**。

定义了多个Netfilter钩子（hook）和对应的回调函数。每个钩子都有一个钩子号（hooknum）、协议族（pf）和优先级（priority）。



**入站（NF\_INET\_LOCAL\_IN）：**

* NF\_HKLocalIn：钩子号为入站（NF\_INET\_LOCAL\_IN），回调函数为 NfHookLocalIn，优先级为最高（NF\_IP\_PRI\_FIRST）。主要实现出站数据包的过滤。

**出站（NF\_INET\_LOCAL\_OUT）：**

* NF\_HKLocalOut：钩子号为出站（NF\_INET\_LOCAL\_OUT），回调函数为 NfHookLocalOut，优先级为最高（NF\_IP\_PRI\_FIRST）。主要实现入站数据包的过滤。

**预路由（NF\_INET\_PRE\_ROUTING）：**

* nat链：
  + NF\_HKPreRouting：钩子号为预路由（NF\_INET\_PRE\_ROUTING），回调函数为 NfHookPreRouting，优先级为目标地址转换（NF\_IP\_PRI\_NAT\_DST）。主要实现DNAT。
* filter表：
  + NF\_HKPreRouting2：钩子号为预路由（NF\_INET\_PRE\_ROUTING），回调函数为 NfHookLocalIn，优先级为最高（NF\_IP\_PRI\_FIRST）。主要实现对需要转发的数据包的过滤，如果是从内网到互联网的数据包，则根据过滤规则进行放行（放行时根据NAT记录添加正向连接会话和反向连接会话）或者丢弃，如果是从互联网到内网的数据包，则检查连接会话表决定是否转发。

**后路由（NF\_INET\_POST\_ROUTING）：**

* nat链：
  + NF\_HKPostRouting：钩子号为后路由（NF\_INET\_POST\_ROUTING），回调函数为 NfHookPostRouting，优先级为源地址转换（NF\_IP\_PRI\_NAT\_SRC）。主要实现SNAT。
* filter表：
  + NF\_HKPostRouting2：钩子号为后路由（NF\_INET\_POST\_ROUTING），回调函数为 NfHookLocalIn，优先级为最高（NF\_IP\_PRI\_FIRST）。主要实现对需要转发的数据包的过滤，与预路由阶段的filter表同理。

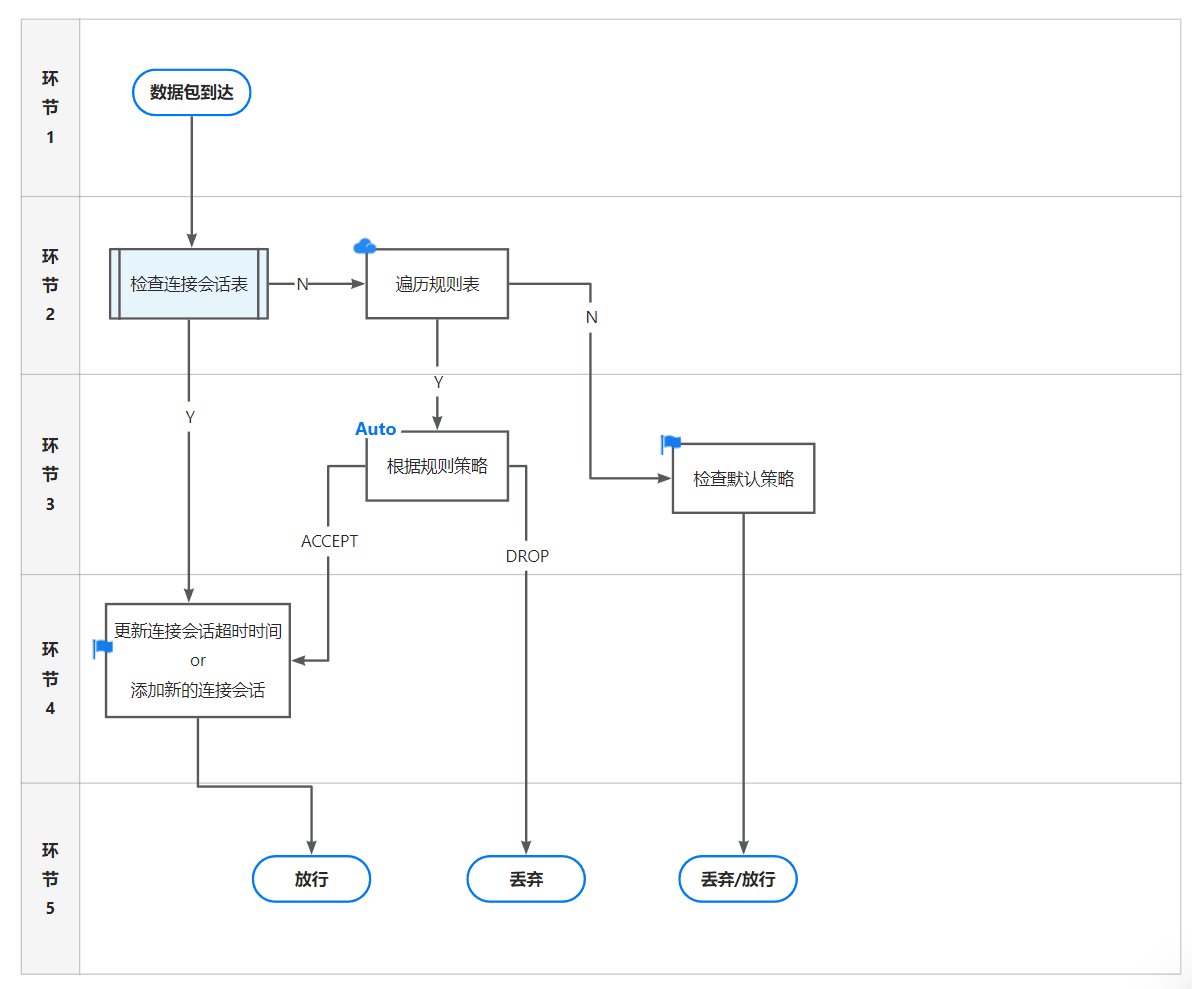
### 包过滤

根据特定的过滤规则对网络数据包进行筛选和处理。通过包过滤，可以控制网络流量，允许或阻止特定类型的数据包通过网络设备。

防火墙程序在内核进程中维护一个包过滤的规则链表。每个元素包含五元组和其他信息——源IP、目的IP、源端口、目的端口、协议类型以及对报文的操作。用户空间程序可以通过之前提到的netlink套接字对这个规则链表进行维护。

在数据包出站与入站时首先在连接会话表中查询是否存在对应的连接会话，如果存在则直接放行。如果没有找到连接会话，则遍历规则链表，并与数据包进行匹配，然后对出入站数据包执行放行与丢弃的操作，如果策略为放行，则在连接会话表中插入一条新的连接。（关于连接状态的实现在后文描述）

整体的流程图如下所示。

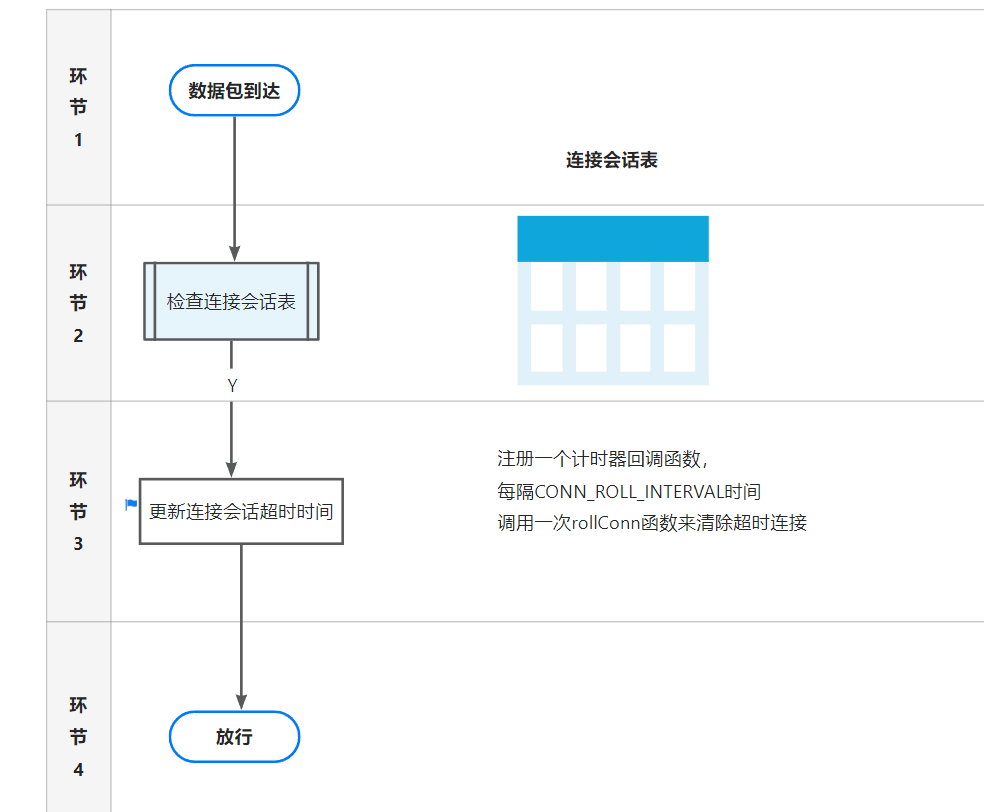


### 连接状态

在内核维护一个连接会话表，为每一个第一次通过的数据包插入一条连接会话项，这样当数据包到达时，就能够先检查会话表，而不需要每次都遍历规则表，大大节省了时间。

由于对于连接会话来说，数据包到达的频率远远大于新连接建立的频率，因此使用红黑树存储连接会话，能够大大降低数据包到达时匹配连接会话的时间复杂度。

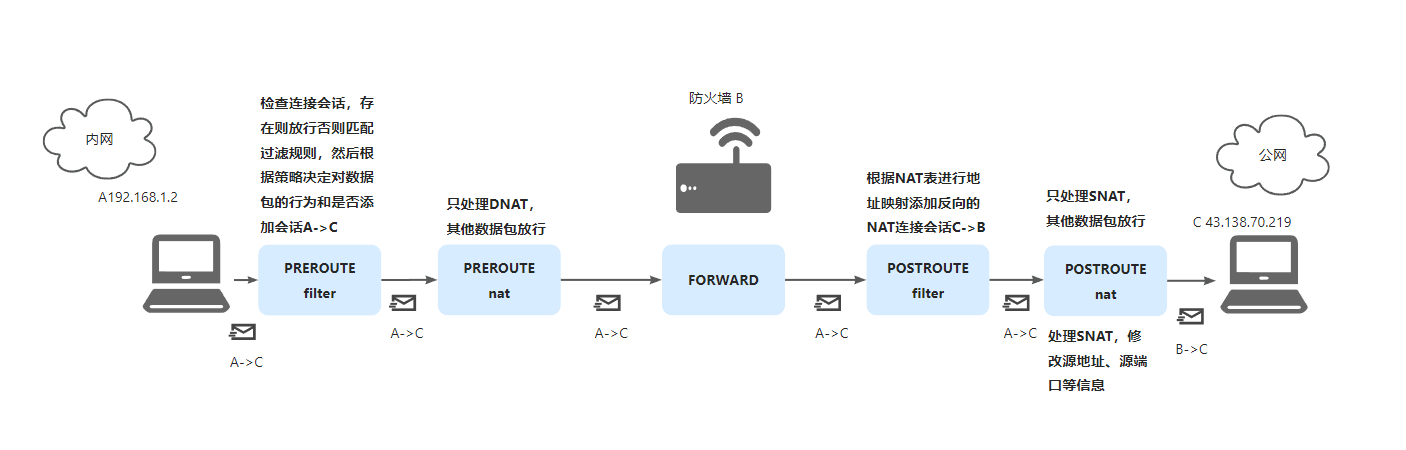
连接表的检查流程如下所示。如果数据包到达，则更新这条连接的超时时间。对于连接会话表，首先初始化一个定时器，定时器注册一个回调函数，实现一段时间后清理超时的连接会话。为了让这个过程能一直持续下去，可以在计时器回调函数中重新激活定时器，这样就实现了每隔CONN\_ROLL\_INTERVAL时间调用一次rollConn函数。



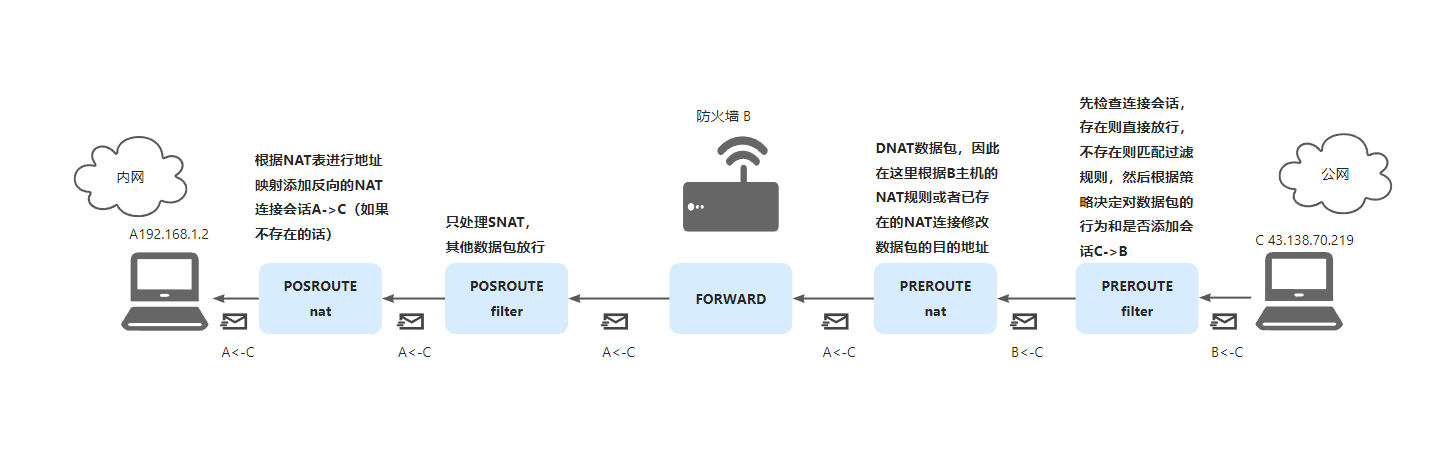
### NAT 网络地址转换

当内网主机访问互联网时，会建立两个连接会话 A->C 和 C->B，这两条会话都有对应的 NAT 记录，进入互联网的数据包首先进行过滤，放行则添加连接会话 A->C。在后路由阶段匹配连接会话 A->C 的 NAT 规则，修改源地址（如果是初次进入则会添加反向的连接会话 C->B 并创建对应的 NAT 规则）然后发送到互联网。从互联网进入内网的数据包先在预路由阶段检查是否存在连接会话，如果存在则根据连接会话 C->B 的 NAT 记录修改目的地址，并进行转发。

数据包从内网发送到互联网的网络地址转换流程如下所示。首先在预路由阶段匹配过滤规则，如果可以通过或者存在连接会话则放行，然后在预路由nat阶段不进行处理，经过转发进入后路由阶段。在filter链添加反向的NAT连接（如果不存在的话），之后在后路由nat阶段对数据包进行处理，修改源IP和源端口号等数据，实现SNAT。



数据包从互联网发送到内网的网络地址转换流程如下所示。从互联网进入的数据包首先经过预路由的filter链，检查连接会话表，如果存在连接则放行，不存在则检查规则表，如果允许通过则添加一条连接会话。放行之后进入预路由nat阶段，在这里根据会话表或者存在的关于目的地的NAT规则修改数据包的目的地址和目的端口，实现DNAT，然后经过转发和后路由filter链阶段，在后路由nat阶段检查连接会话表查看是否存在反向连接，不存在则添加一条反向连接，实现DNAT。



### web 管理面板

#### 数据库设计

#### 前端页面设计

#### 后台接口编写

## 项目测试

## 心得体会