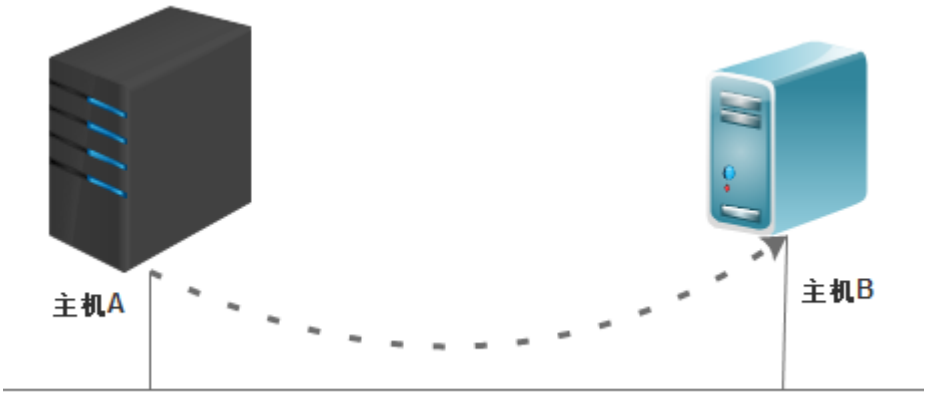
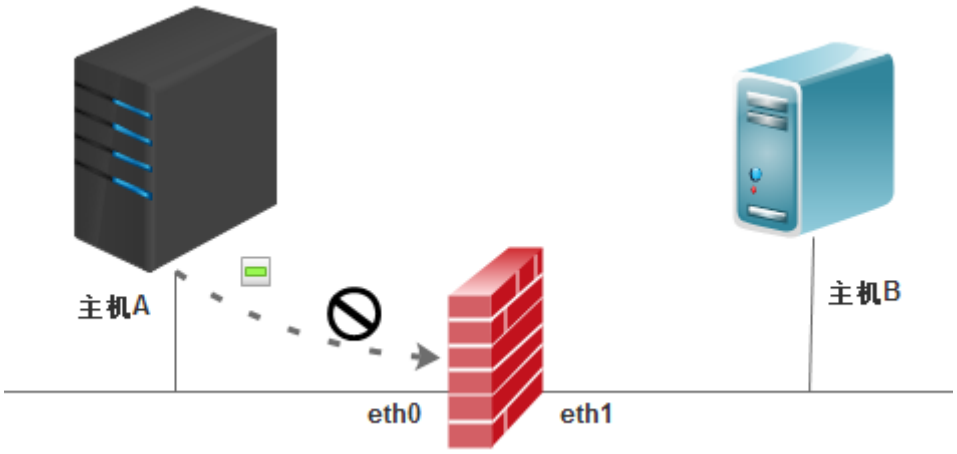
6.1 为什么需要防火墙

对于没有防火墙存在的一条网络路线中，主机 A 发送给主机 B 的任何一个数据包，主机 B 都会照单全收，即使是包含了病毒、木马等的数据也一样会收。虽说害人之心不可有，但是在网络上，你认为是害你的行为在对方眼中是利他的行为。所以防人之心定要有，防火墙就可以提供一定的保障。



有了简单的防火墙之后，在数据传输的过程中就会接受"入关"检查，能通过的数据包才继续传输，不能通过的数据包则拒绝或者直接丢弃。



从上面的图中可以看出，防火墙至少需要两个网卡，其中一块控制流入数据包，另一块网卡控制流出数据包。即使是软件防火墙，要实现完整的防火墙功能，也需要至少两块网卡。

所谓防火墙就是"防火的墙"，如果过来的是"火"就得挡住，如果过来的不是"火"就放行，但什么是"火"，这由人们自行定制。

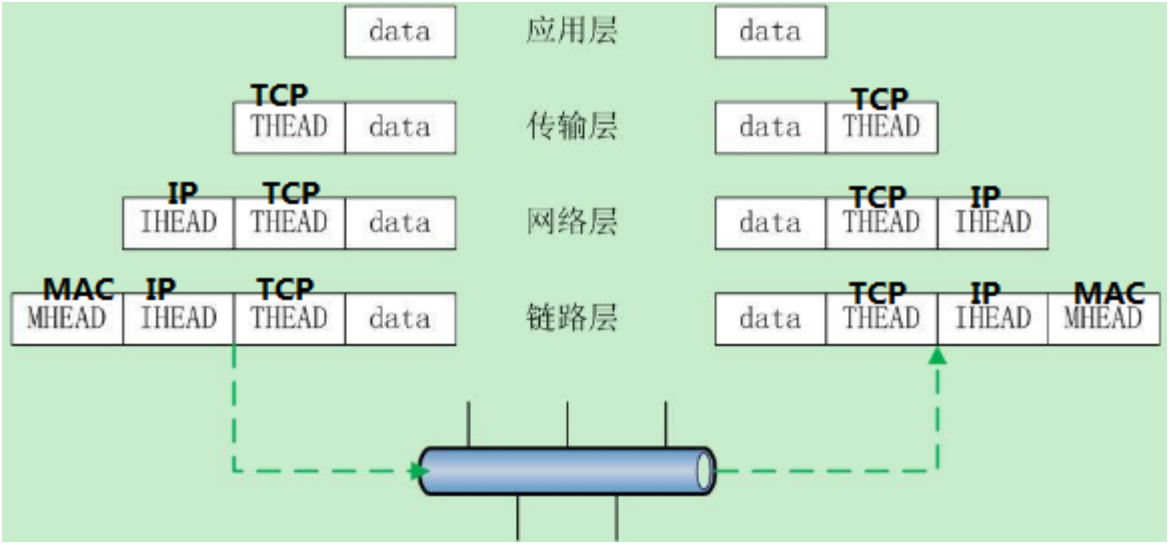
但无论如何，所谓的"火"都是基于 OSI 七层模型的，简单的划分为四层：最高的应用层（如 HTTP/FTP/SMTP），往下一层是传输层（TCP/UDP），再往下一层是网络层，最后是链路层。可以基于整个 7 层模型的每一层来定制防火墙，但是默认防火墙（没有编译内核源码定制七层防火墙）一般认为工作在以上的 4 层中。

在网上和很多书籍上都详细解释了 OSI 七层模型各层的行为和作用，我个人推荐一本《计算机网络原理创新教程》，里面的 OSI 模型是我学习过最通俗易懂且完整详细的教程。

6.2 数据传输流程

6.2.1 网络数据传输过程

首先看看网络数据传输的基本流程。



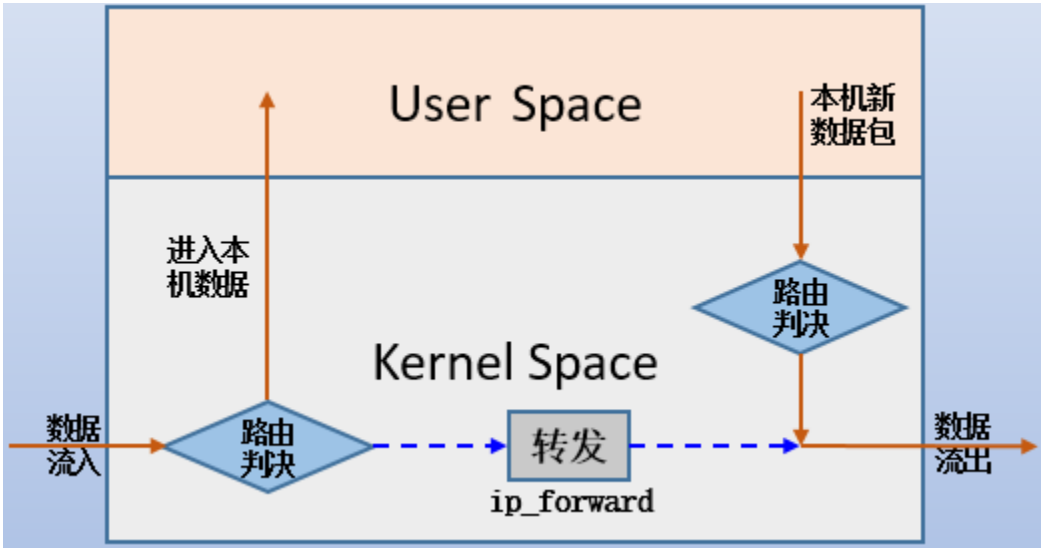
数据从上层进入到传输层，加上源端口和目标端口成为数据段(如果是 UDP 则成为数据报)，再进入网络层加上源 IP 和目标 IP 成为数据包，再进入链路层加上源 MAC 地址和目标 MAC 地址成为数据帧，这段过程是一种"加头"封装数据的过程。数据经过网络传输到达目标主机后，逐层"剃头"解包，最终得到 data 纯数据内容。

6.2.2 本机数据路由决策

其实，进程间数据传输的方式有多种：共享内存、命名管道、套接字、消息队列、信号量等。上面描述的 OSI 通信模型只是数据传输的一种方式，它特指网络数据传输，是基于套接字(ip+port)的，所以既可以是主机间进程通信，也可以是本机服务端和客户端进程间的通信。

无论如何，网络数据总是会流入、流出的，即使是本机的客户端和服务端进程间通信，也需要从一个套接字流出、另一个套接字流入，只不过这些数据无需路由、无需经过物理网卡(走的是 LoopBack)。**当接收外界发送的数据时，在数据从网卡流入后需要对它做路由决策，根据其目标决定是流入本机数据还是转发给其他主机，如果是流入本机的数据，则数据会从内核空间进入用户空间(被应用程序接收、处理)。当用户空间响应(应用程序生成新的数据包)时，响应数据包是本机产生的新数据，在响应包流出之前，需要做路由决策，根据目标决定从哪个网卡流出。如果不是流入本机的，而是要转发给其他主机的，则必然涉及到另一个流出网卡，此时数据包必须从流入网卡完整地转发给流出网卡，这要求 Linux 主机能够完成这样的转发。但 Linux 主机默认未开启 ip_forward 功能，这使得数据包无法转发而被丢弃。Linux 主机和路由器不同，路由器本身就是为了转发数据包，所以路由器内部默认就能在不同网卡间转发数据包，而 Linux 主机默认则不能转发。**

下图可以很好地解释上面的过程：



本文是为了介绍防火墙的，充当防火墙的主机需要至少两块网卡，所以有必要解释下数据流入和流出时，Linux 主机是如何处理的。

首先要说明的是，**IP 地址是属于内核的(不仅如此，整个 tcp/ip 协议栈都属于内核，包括端口号)**，只要能和其中一个地址通信，就能和另一个地址通信(这么说不严谨，准确地说是能路由这两个地址)，而不管是否开启了数据包转发功能。例如某 Linux 主机有两网卡 eth0:172.16.10.5 和 eth1:192.168.100.20，某 192.168.100.22 主机网关指向 192.168.100.20，它能 ping 通 192.168.100.20，但也一样能 ping 通 172.16.10.5，因为地址属于内核，从 eth1 进来的数据包被内核分析时，发现目标地址为本机地址，直接就产生新数据包回应 192.168.100.22，根据路由决策，该响应包应从 eth1 出去，于是 192.168.100.22 能收到回复完成整个 ping 过程。

在此过程中，没有进行数据包转发过程，因为流出的响应包是新产生的，而非原来流入的数据包。如果流入和流出的包是一样的(或者稍作修改)，则数据流入后不能进入用户空间，而是直接通过内核转发给另一个网卡。数据包从网卡 1 交给网卡 2，这个过程就是转发，在 Linux 主机上由 ip_forward 进行控制。例如，网卡 1 所在网段主机 ping 网卡 2 所在主机时，数据包流入网卡 1 后就需要转交给网卡 2，然后从网卡 2 流出。

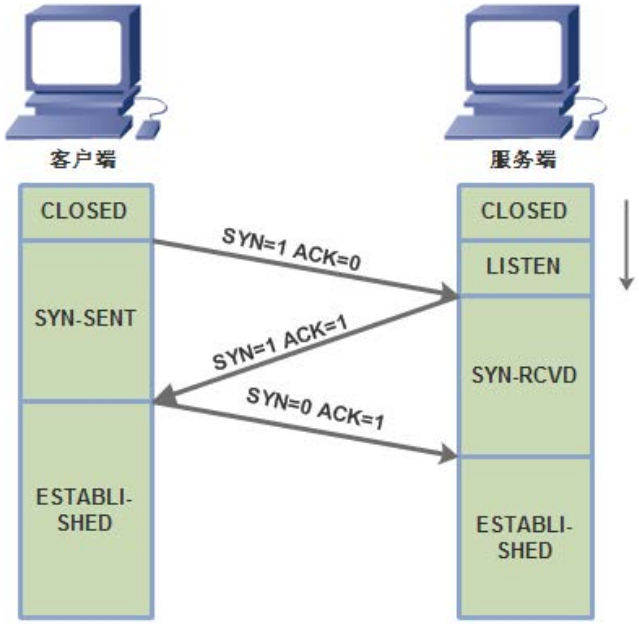
在后文中有实验专门测试和说明上面的过程。

6.3 TCP 三次握手、四次挥手以及 syn 攻击

每次 TCP 会话的建立都需要经过三次握手，断开时都需要四次挥手。

6.3.1 三次握手建立 TCP 连接

如图。



(1).客户端和服务端都处于 CLOSED 状态。(发起 TCP 请求的称为客户端，接受请求的称为服务端)

(2).服务端打开服务端口，处于 listen 状态。

(3).客户端发起连接请求。首先发送 SYN(synchronous)报文给服务端，等待服务端给出 ACK 报文回应。发送的 SYN=1，ACK=0，表示只发送了 SYN 信号。此时客户端处于 SYN-SENT 状态（SYN 信号已发送）。

(4).服务端收到 SYN 信号后，发出 ACK 报文回应，并同时发出自己的 SYN 信号请求连接。此时服务端处于 SYN-RECV 状态(syn recieved，在图中显示的是 SYN-RCVD)。发送的 SYN=1 ACK=1，表示发送了 SYN+ACK。

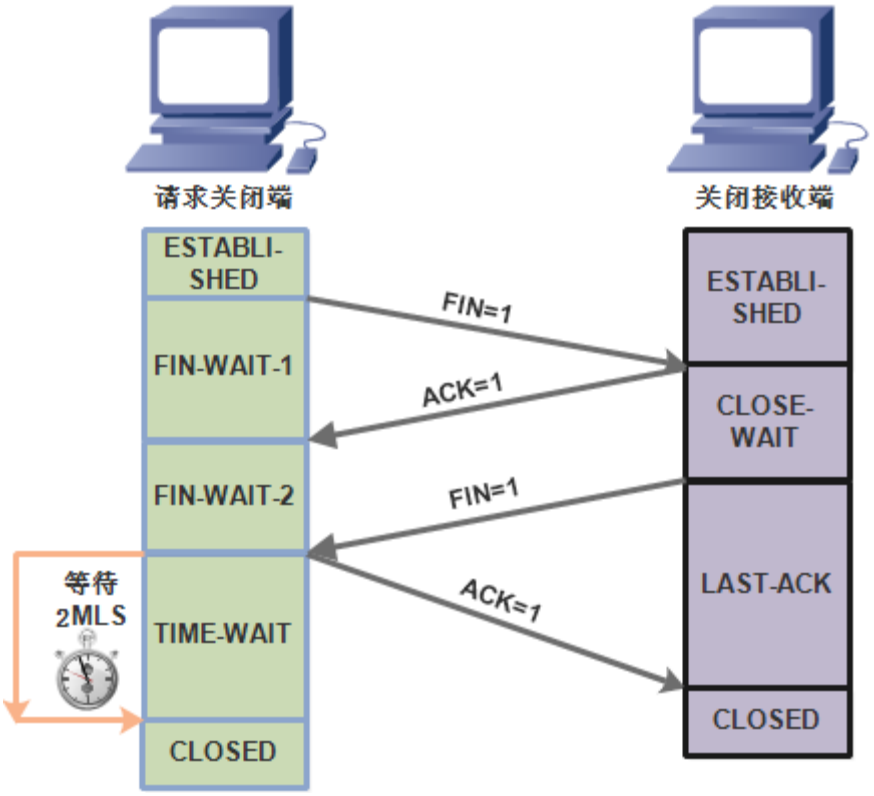
(5).客户端收到服务端的确认信号 ACK 后，再次发送 ACK 信号给服务端以回复服务端发送的 syn。此时客户端进入 ESTABLISHED 状态，发送的 SYN=0，ACK=1 表示只发送了 ACK。

(6).服务端收到 ACK 信号后，也进入 ESTABLISHED 状态。

此后进行数据的传输都通过此连接进行。其中第 3、4、5 步是三次握手的过程。这个过程通俗地说就是双方请求并回应的过程：①A 发送 syn 请求 B 并等待 B 回应；②B 回应 A，并同时请求 A；③A 回应 B。

6.3.2 四次挥手断开 TCP 连接

断开之前，双方都处于 ESTABLISHED 状态。假设是客户端请求断开连接。



(1).客户端发送 FIN(finally)报文信号，请求断开。此后客户端进入 FIN-WAIT-1 状态。

(2).服务端收到 FIN 信号，给出确认信号 ACK，表示同意断开。此时服务端进入 CLOSE-WAIT 状态。此过程结束后表示从客户端到服务端方向的 TCP 连接已经关闭了，也就是说整个 TCP 连接处于半关闭状态。

(3).客户端收到服务端的 ACK 后进入 FIN-WAIT-2 状态，以等待服务端发出断开信号 FIN。在客户端的 FIN-WAIT-2 状态的等待过程中，服务端再发出自己的 FIN 信号给客户端。此时服务端进入 LAST-ACK 状态。

(4).客户端收到服务端的 FIN 信号，给出回应信号 ACK，表示接受服务端的断开请求，此时客户端进入 TIME-WAIT 状态，此时客户端已脱离整个 TCP，只需再等待一段时间(2*MSL)就自动进入 CLOSED 状态。

(5).服务端收到客户端的回应 ACK 信号，知道客户端同意了服务端到客户端方向的 TCP 断开，直接进入 CLOSED 状态。

以上的 1、2、3、4 步是四次挥手阶段。从中可以看出，四次挥手和三次握手的过程其实是类似的，都是双方发出断开请求并回应对方，只不过四次挥手的过程是将服务端发送的 ACK 和 FIN 分开发送了，而三次握手的过程中服务端发送的 ACK 和 SYN 是放在一个数据包内发送的。

以上所述是客户端请求的断开，服务端也可以请求断开，这时过程是完全一致的，只不过角色互换了。还需注意的是，**如果是客户端请求断开，那么服务端就是被动断开端，可能会保留大量的 CLOSE_WAIT 状态的连接，如果是服务端主动请求断开，则可能会保留大量的 TIME_WAIT 状态的连接。**由于每个连接都需要占用一个文件描述符，高并发情况下可能会耗尽这些资源。因此，需要找出对应问题，做出对应的防治，一般来说，可以修改内核配置文件/etc/sysctl.conf 来解决一部分问题。

6.3.3 syn flood 攻击

syn 洪水攻击是一种常见的 DDos 攻击手段。攻击者可以通过工具在极短时间内伪造大量随机不存在的 ip 向服务器指定端口发送 tcp 连接请求，也就是发送了大量 syn=1 ack=0 的数据包，当服务器收到了该数据包后会回复并同样发送 syn 请求 tcp 连接，也就是发送 ack=1 syn=1 的数据包，此后服务器进入 SYN-RECV 状态，正常情况下，服务器期待收到客户端的 ACK 回复。但问题是服务器回复的目标 ip 是不存在的，所以回复的数据包总被丢弃，也一直无法收到 ACK 回复，于是不断重发 ack=1 syn=1 的回复包直至超时。

在服务器被 syn flood 攻击时，由于不断收到大量伪造的 syn=1 ack=0 请求包，它们将长时间占用资源队列，使得正常的 SYN 请求无法得到正确处理，而且服务器一直处于重传响应包的状态，使得 cpu 资源也被消耗。总之，syn flood 攻击会大量消耗网络带宽和 cpu 以及内存资源，使得服务器运行缓慢，严重时可能会引起网络堵塞甚至系统瘫痪。

因此，防范 syn flood 攻击非常重要。当然，首先需要判断出是否受到了 syn flood 攻击。可以通过抓包工具或者 netstat 等工具获取处于 SYN_RECV 状态的半连接，如果有大量处于 SYN_RECV 且源地址都是乱七八糟的，说明受到了 syn 洪水攻击。

例如使用 netstat 工具判断的方法如下：

```
[root@xuexi ~]# netstat -tnlpa | grep tcp | awk '{print $6}' | sort | uniq -c
      1 ESTABLISHED
      7 LISTEN
    256 SYN_RECV
```

6.4 防火墙的判断范围

从设备上分类，防火墙分为软件防火墙、硬件防火墙、芯片级防火墙。后文所说的可能是软件防火墙、也可能是硬件防火墙，在理解上它们没什么区别，只是将防火墙剥离成了独自的服务器而已。

从技术上分类，防火墙分为数据包过滤型防火墙、应用代理型防火墙。这是因为四层模型的每一层都可以应用防火墙。

6.4.1 从链路层来判断是否处理

基于链路层的防火墙是控制 MAC 的。例如，可以将公司内网员工电脑的 MAC 地址全部记录到防火墙上，从而限制他们上外网。再例如，可以将公司电脑的 MAC 地址全部记录到防火墙使他们能够上网，但是非本公司的电脑就无法从本公司上网。

但是，基本上不会有公司这样做，这样的行为太死板，而且记录 MAC 地址本身就是一件很麻烦的事。

6.4.2 从网络层来判断是否处理

网络层的核心是 IP（也包括 icmp 等）。所以从网络层来判断，可以基于源 IP、目标 IP 来指定防火墙的规则。例如，来自 38.68.100.61 的主机不能穿过防火墙；访问目标是 192.168.109.19 的服务器的请求不能让其穿过防火墙；还可以设置 icmp 协议作为判断依据，使得外网人员的 ping 包被挡住。

在网络层可以用来制定防火墙规则的内容有很多。如下表。最常用的也就是后三个而已。

字段	说明
Header Length	IP包的长度
Differentiated服务	差别服务判断，执行QOS时可能用到，一般正常值为0
Total Length	数据包不包含链路层报头的整体长度
Flags	标明该数据报是否被分割，或者不可被分割
Time to Live	数据包的存活时间
Protocol	上层协议，如TCP为06、UDP为17、icmp为01
Source	源IP
Destination	目标IP

6.4.3 从传输层来判断是否处理

可以从 TCP 或者 UDP 来判断。以 TCP 为例，例如限制目标端口是 22 端口的请求，这样 SSH 就无法连接上服务器了。

下表是 TCP 数据包中可以用来制定防火墙规则的字段。

字段	说明
Source Port	源端口号
Destination Port	目标端口号
Header Length	TCP包头的整体长度
Flags	TCP包头内的连接控制标志，如SYN/ACK/FIN，它们是TCP包头中很重要的信息

6.4.4 从应用层来判断是否处理

到了这一层的处理就属于应用代理型的防火墙了。他需要解开数据包并还原数据，也就是说它可以获取到数据包中的所有内容，但也因此负担很重，所需 CPU 和内存较大。它的适用面较窄。

6.4.5 特殊的防火墙判断

除了以上 4 种判定方式，还有几种特殊的判断方式也较为常用。

◇ 根据数据包内容判断

例如，不允许内网的客户端连上 taobao.com 上的任何主机，可以在防火墙上检查 DNS 的解析包中是否包含"taobao.com"这个字符串，如果有就丢弃，这样就可以让 DNS 对任何 taobao.com 上的主机解析失败达到限制上该网的目的。

注意：虽说数据部分是应用层的，但是有些防火墙在网络层就可以进行检查。Linux 默认自带的 iptables/netfilters 就是其中一种。

◇ 根据关联状态判断

假如现在不允许任何 internet 上的主机进入到公司内部，但是允许企业内的计算机可以上网。这样的设定目的是为了防止来自外网的攻击。但是如果"禁止源地址为外网的所有地址穿过防火墙、允许源地址为公司内部的地址穿过防火墙"来设置防火墙，将导致一个问题：内网连上 internet 后请求

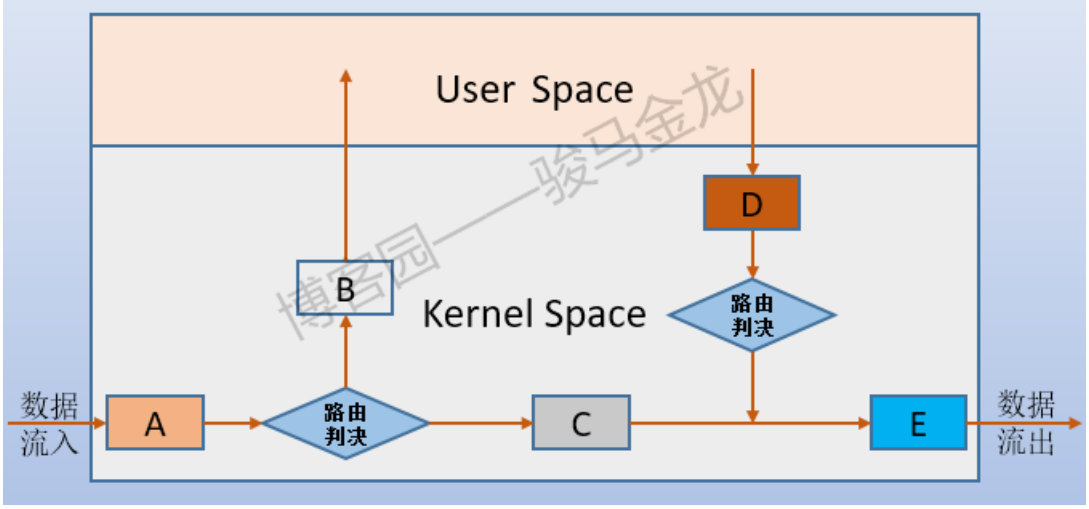
某个网页，要能正常上网，内网计算机必然要接收外网网页的返回数据。但外网数据包无法穿过防火墙，这样并没有实现内网主机上网的目的。

现在假设内网某机器上外网时的套接字为 192.168.100.8:9000，想要访问 10.0.0.5:80，也即是说数据流向是 192.168.100.8:9000→10.0.0.5:80，那么返回的数据包流向必定是 10.0.0.5:80→192.168.100.8:9000。根据这种关联性，防火墙可以设定允许这样的数据包通过。

这属于连接跟踪的行为。FTP 服务器对于防火墙的设置是一个考验，如果没有连接跟踪的功能，数据通道的端口不固定性将导致防火墙设置极其困难。

6.4.6 数据包过滤

以下是协议栈(TCP/IP 协议栈)底层大致机制。数据包都要通过 A 流入，再根据路由决策决定数据包的流向(网络层)。如果是流入本机，则经过 B 进入用户空间层，对于每个从本机流出的数据包，也都要经过路由决策来决定从哪个网络接口出去，然后路经 D，从 E 出去。如果不是流入本机则流向 C，然后顺着 E 点出去。



上图有一处容易理解错误，从用户空间层出去的数据包是本机新产生的数据包，可能是对流入数据包的响应数据，也可能是本机应用向外发出的请求数据包。总之，数据包走不完 A-->B 点-->User Space-->D-->E 这条路，在数据包进入用户空间层时已经被处理了，从用户空间层出来的数据已经不是原流入的数据，所以在上图中我将用户空间的两个箭头断开了。

用英文来说明 ABCDE 这 5 个点就比较浅显易懂：

A 表示：altering packets as soon as they come in,

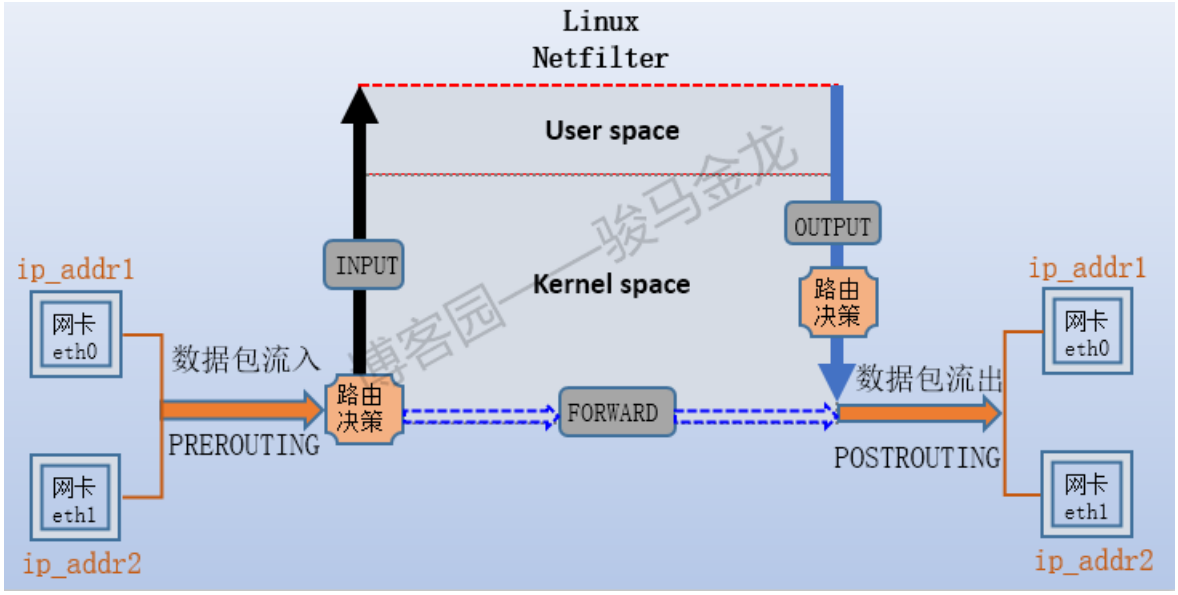
B 表示：destined to local sockets,

C 表示：be routed through the box,

D 表示：locally-generated packets, and altering before routing,

E 表示：altering packets as they are about to go out。

在上图中的 ABCDE 这五个点，其实就是防火墙发挥作用的点。在 Linux 主机上，防火墙是由内核空间的 netfilter 实现的，其中作用在 ABCDE 这五个点的分别称为 PREROUTING 链、INPUT 链、FORWARD 链、OUTPUT 链和 POSTROUTING 链。这几个术语在后文会做非常详细的说明。



6.4.7 iptables 和 Netfilter 的关系

防火墙起作用的是 Netfilter，而 **iptables 只是管理控制 netfilter 的工具**，可以使用该工具进行相关规则的制定以及其他的动作。iptables 是用户层的程序，netfilter 是内核空间的，在 netfilter 刚加入到 Linux 中时，netfilter 是一个 Linux 的一个内核模块，要实现其他的防火墙行为还需要加载其他对应的模块，到了后来 netfilter 一部分必须的模块已经加入到内核中了。

也就是说，iptables 命令工具操作的 netfilter，真正起"防火"作用的是 netfilter。

6.5 Linux 上防火墙相关基础

6.5.1 netfilter 与其模块

Linux 是一个极其模块化的内核。netfilter 也是以模块化的形式存在于 Linux 中，所以每添加一个和 netfilter 相关的模块，代表着 netfilter 就多一个功能。

但是有些模块是使用 netfilter 所必须的，所以这些模块已经默认编译到内核中而非需要时加载。

存放 netfilter 模块的目录有三个：
/lib/modules/$kernel_ver/net/{netfilter,ipv4/netfilter,ipv6/netfilter}。$kernel_ver 代表内核版本号。

其中 ipv4/netfilter/存放的 ipv4 的 netfilter，ipv6/netfilter/存放的 ipv6 的 netfilter，/lib/modules/$kernel_net/kernel/netnetfilter/存放的是同时满足 ipv4 和 ipv6 的 netfilter。在最后一个目录中放入更多的模块，是 netfilter 团队发展的目标，因为要维护 ipv4 和 ipv6 两个版本挺累的。

6.5.2 netfilter 的结构

要使 netfilter 能够工作，就需要将所有的规则读入内存中。netfilter 自己维护一个内存块，在此内存块中有 4 个表：filter 表、NAT 表、mangle 表和 raw 表。在每个表中有相应的链，链中存放的

是一条条的规则，规则就是过滤防火的语句或者其他功能的语句。也就是说表是链的容器，链是规则的容器。实际上，每个链都只是一个 hook 函数（钩子函数）而已。

说到这里，需要纠正一个概念，Linux 上的防火墙是由 netfilter 实现的，但是 netfilter 的功能不仅仅只有"防火"，一般可以认为"防火"的功能只是 filter 表的功能。

关于这 4 个表，它们的结构如下：

	PREROUTING	INPUT	FORWARD	OUTPUT	POSTROUTING
FILTER	×	√	√	√	×
NAT	√	×	×	√	√
mangle	√	√	√	√	√
raw	√	×	×	√	×

注：从内核 2.6.34 开始，NAT 表支持操作 INPUT 链。它只为 SNAT 服务。和 snat on postrouting 类似，只不过 snat on input 用来转换"目标是本机的数据包"的源地址。

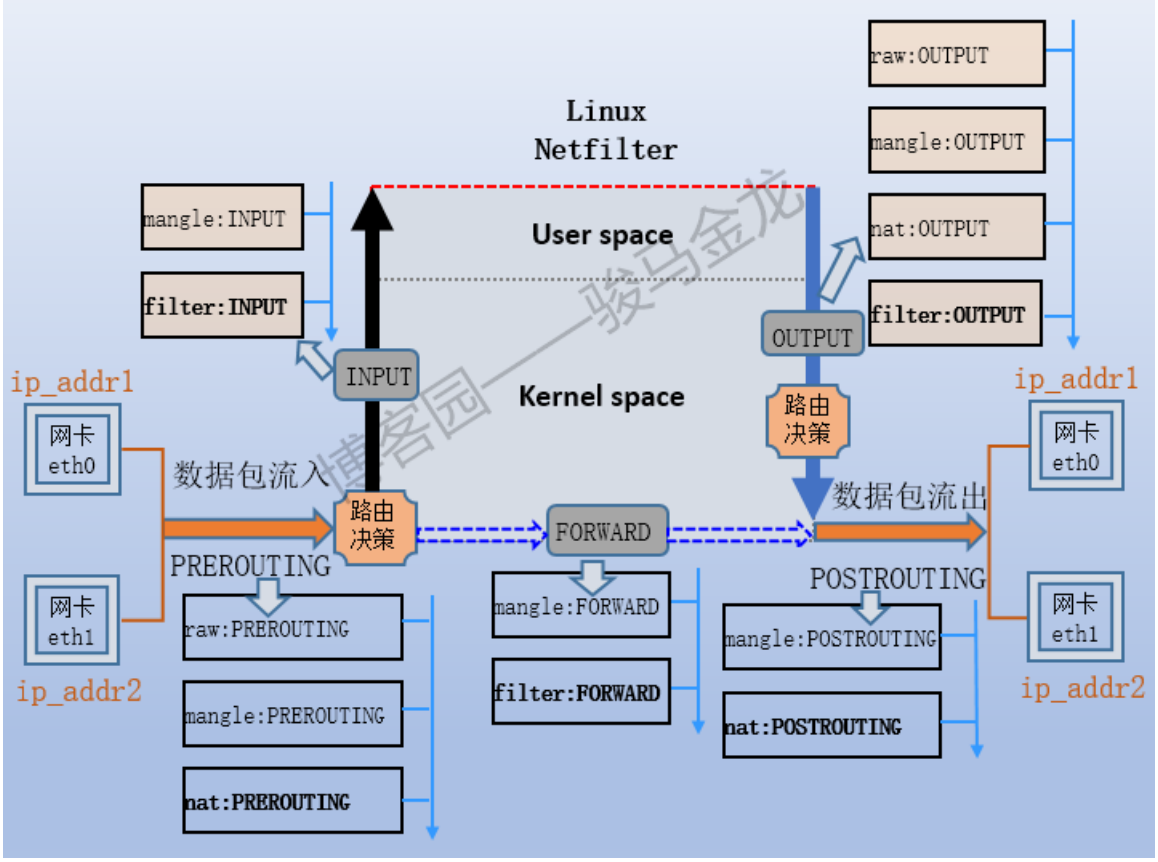
◇ filter 表：netfilter 中最重要的表，负责过滤数据包，也就是防火墙实现"防火"的功能。filter 表中只有 OUTPUT/FORWARD/INPUT 链。

◇ NAT 表：实现网络地址转换的表。可以转换源地址、源端口、目标地址、目标端口。NAT 表中的链是 PREROUTING/POSTROUTING/OUTPUT。

◇ mangle 表：一种特殊的表，通过 mangle 表可以实现数据包的拆分和还原。mangle 表中包含所有的链。

◇ raw 表：加速数据包穿过防火墙的表，也就是增强防火墙性能的表。只有 PREROUTING/OUTPUT 表。

由于这几个表中有重复的链，所以数据被不同链中规则处理时是由顺序的。下图是完整的数据包处理流程。



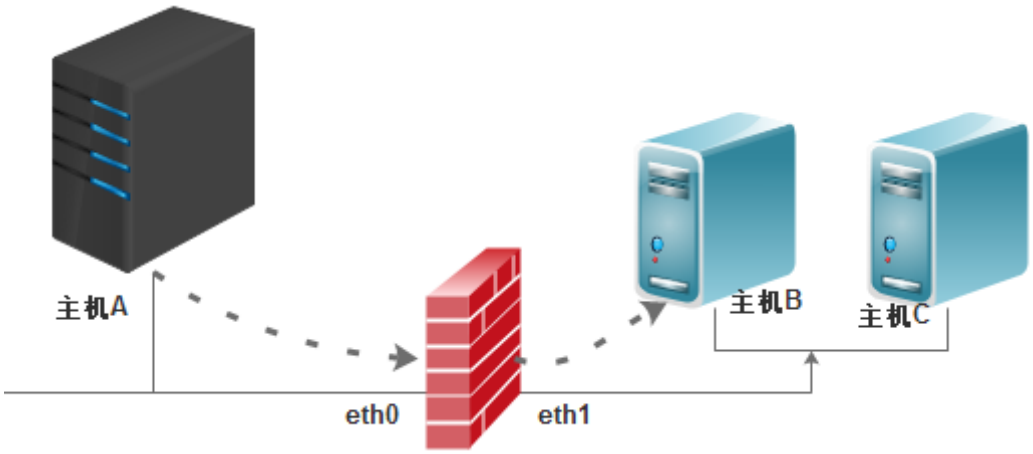
6.5.3 INPUT、OUTPUT、FORWARD 链

每个链对应的都是同名称的数据包，如 INPUT 链针对的是 INPUT 数据包。

INPUT 链的作用是为了**保护本机**。例如，如果进入的数据包的目标是本机的 80 端口，且发来数据包的地址为 192.168.100.9 时则丢弃，这样的规则应该写入本机的 INPUT 链。但是要注意，这个本机指的是防火墙所在的机器，如果是硬件防火墙，那么一定会配合 FORWARD 链。

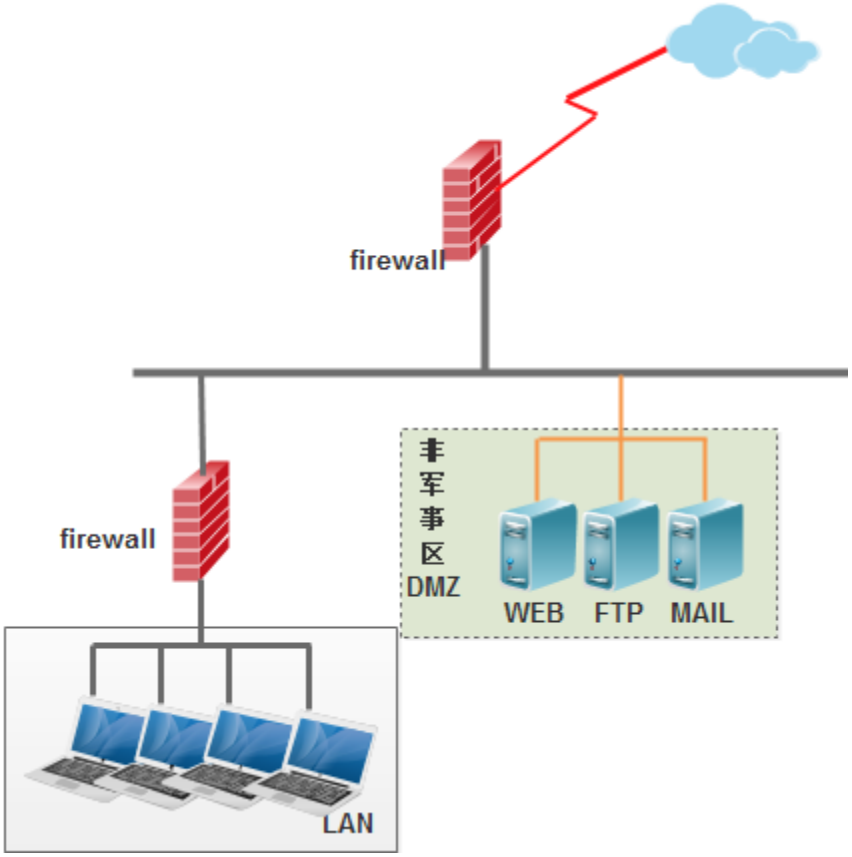
OUTPUT 链的作用是为了**管制本机**。例如，限制浏览 www.taobao.com 网页。

INPUT 和 OUTPUT 链很容易理解，FORWARD 链起的是什么作用呢？数据包从一端流入，但是不经过本机，那么就要从另一端流出。对于硬件防火墙这很容易理解，如下图，数据总要转发到另一个网卡接口然后进入防火墙负责为其"防火"的网段。也就是说，**FORWARD 链的作用是保护"后端"的机器。**



前文说过，Linux 主机自身也有 ip_forward 功能用于网卡间的数据包转发，这个 ip_forward 和 netfilter 的 forward 链有什么区别呢？这就是防火墙的意义，当数据包需要转发时，仅使用 ip_forward 时将不问是非对错总是进行转发，而使用 forward 链时可以筛选这些需要转发的数据包，以决定是否要被转发。显然，ip_forward 的功能是转发数据包，而 forward 链是筛选允许转发的数据包，然后让 ip_forward 转发。这也说明 forward 链要正常工作，要求开启 ip_forward 功能。

6.5.4 防火墙布线示例



这样的布置不仅给了服务器对外的一层防火墙，也给了服务器对内的一层防火墙，可以防外人也可以防内贼。

图中的 DMZ 称为"非军事区"，一般是放在两个防火墙中间做内网和外网的缓冲。有些必须对外提供服务的服务器应该放在这种区域，而不能直接放在内网，因为将其放入内网又对外提供服务，当它被外界攻击时就可以借助它(肉机)作为跳板攻陷其它内网主机。

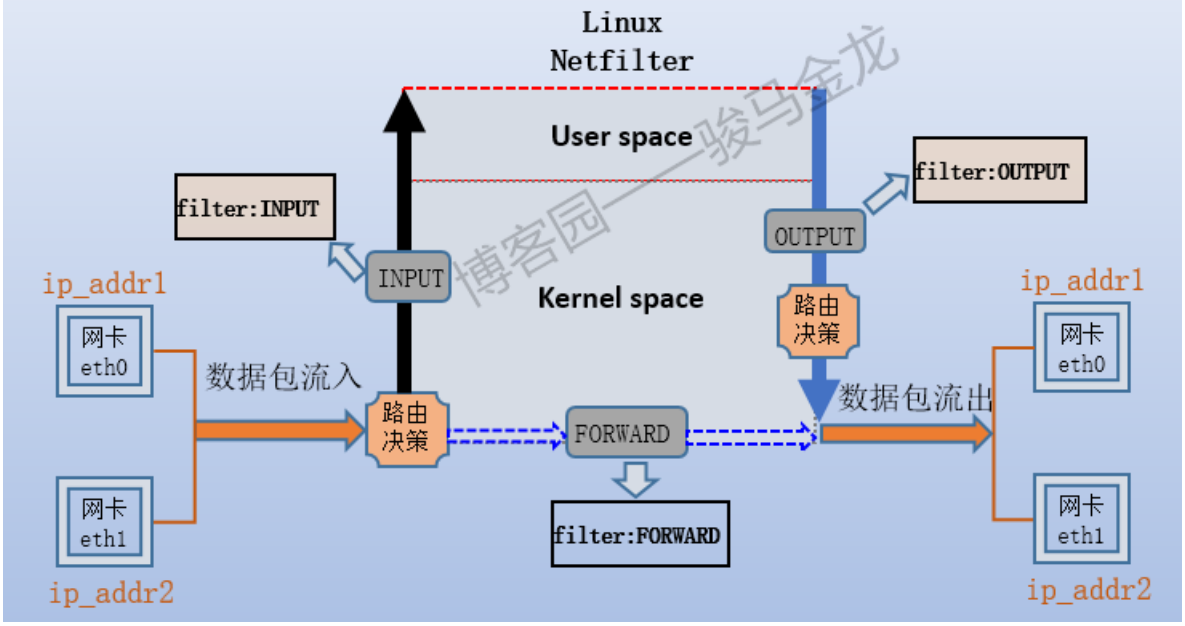
一般构建 DMZ 区有以下几个要求：

1. **内网可以访问外网。显然，内网用户需要自由地访问外网。**
2. **内网仅部分主机可以访问 DMZ，此策略是为了方便内网用户使用和管理 DMZ 中的服务器。不应该全部放行内网访问 DMZ，否则有内贼的风险。**
3. **外网不能访问内网。内网中存放的是公司内部数据，这些数据不允许外网的用户进行访问。**
4. **外网可以访问 DMZ。DMZ 中的服务器本身就是要给外界提供服务的。**

5. **DMZ 不能访问内网。如果违背此策略，则当入侵者攻陷 DMZ 时，就可以进一步进攻到内网的重要数据。**

6.6 filter 表

只考虑 filter 表的时候，防火墙处理数据包的过程如下图。



当数据包流入时，首先经过路由判决该数据包是流入防火墙本机还是转发到其他机器的。对于流入本机的数据包经过 INPUT 链，INPUT 链所在位置是一个钩子函数，数据经过时将被钩子函数检查一番，检查后如果发现是被明确拒绝或需要丢弃的则直接丢弃，明确通过的则放行，不符合匹配规则的同样丢弃。但是钩子函数毕竟是半路劫财的角色，所以不管怎么样都要告诉一声 netfilter，说这个数据包是怎样怎样的，即使是拒绝或丢弃了数据包也还是会给出一个通知。

当用户空间产生的数据包要发送出去时，首先经过 OUTPUT 链并被此处的钩子函数检查一番，检查通过则放行，然后经过路由决策判决从哪个接口出去，并最终从网卡流出。

例如，当外界主机请求 WEB 服务的时候，请求数据包流入到本机，路经 INPUT 链被放行。进入到本机用户空间的进程后，本机进程根据请求给出响应报文，该响应报文的源地址是本机，目标地址是外界主机。当响应报文要出去时必须先流经 OUTPUT，然后经过路由决定从哪个接口出去，最终流出并路由到外界主机。

再例如 ping 本机地址(如 127.0.0.1)的时候，ping 请求从用户空间发送后，数据包从用户空间流出，于是流经到 OUTPUT，经过路由决策发现是某网卡的地址，然后从此网卡流出。但是由于 ping 的目标为本机地址，所以数据包仍从流出的网卡流入，并被 INPUT 链检查，最后返回 ping 的结果信息。

注意：**是先经过 OUTPUT，再经过路由决策。**这样一来，对那些检查不通过的数据包会直接丢弃，而无需再消耗资源去做路由决策。

6.7 iptables 命令书写规则

iptables 用法比较复杂，有很多命令、选项和参数。所以，我先绝大多数命令、选项和模块选项列出，然后再举例说明 iptables 命令的用法。

```
Usage: iptables [-t TABLE] COMMAND CHAIN [ expressions -j target ]
```

这表示要操作 TABLE 表中的链，操作动作由 COMMAND 决定，例如添加一条规则、删除一条规则、列出规则列表等。如果是向链中增加规则，则需要写出规则表达式用来检查数据包，并指明数据包被规则匹配上时应该做什么操作，例如允许该数据包 ACCEPT、拒绝该数据包 REJECT、丢弃该数据包 DROP，这些操作称为 target，由"-j"选项来指定。

6.7.1 语法

```
Commands:
Either long or short options are allowed.
  --append  -A chain                    链尾部追加一条规则
  --delete  -D chain                    从链中删除能匹配到的规则
  --delete  -D chain rulenum            从链中删除第几条链，从 1 开始计算
  --insert  -I chain [rulenum]          向链中插入一条规则使其成为第 rulenum 条规则，从 1
开始计算
  --replace -R chain rulenum            替换链中的地 rulenum 条规则，从 1 开始计算
  --list    -L [chain [rulenum]]        列出某条链或所有链中的规则
  --list-rules -S [chain [rulenum]]     打印出链中或所有链中的规则
  --flush   -F [chain]                  删除指定链或所有链中的所有规则
  --zero    -Z [chain [rulenum]]        置零指定链或所有链的规则计数器
  --new     -N chain                    创建一条用户自定义的链
  --delete-chain  -X [chain]            删除用户自定义的链
  --policy  -P chain target             设置指定链的默认策略(policy)为指定的 target
  --rename-chain  -E old new            重命名链名称，从 old 到 new

Options:
[!] --proto  -p proto                   指定要检查哪个协议的数据包：可以是协议代码也可以
是协议名称，
                                        如 tcp,udp,icmp 等。协议名和代码对应关系存放在
/etc/protocols 中
                                        省略该选项时默认检查所有协议的数据包，等价于 all
和协议代码 0
[!] --source -s address[/mask][...]     指定检查数据包的源地址，或者使用"—src"
[!] --destination -d address[/mask][...] 指定检查数据包的目标地址，或者使用"--dst"
[!] --in-interface  -i input name[+]    指定数据包流入接口，若接口名后加"+"，表示匹配该
接口开头的所有接口
[!] --out-interface -o output name[+]   指定数据包流出接口，若接口名后加"+"，表示匹配该
接口开头的所有接口
  --jump        -j target               为规则指定要做的 target 动作，例如数据包匹配上规
则时将要如何处理
```

```
  --goto        -g chain                直接跳转到自定义链上
  --match       -m match                指定扩展模块
  --numeric     -n                      输出数值格式的 ip 地址和端口号。默认会尝试反解为
主机名和端口号对应的服务名
  --table       -t table                指定要操作的 table，默认 table 为 filter
  --verbose     -v                      输出更详细的信息
  --line-numbers                        当 list 规则时，同时输出行号
  --exact       -x                      默认统计流量时是以 1000 为单位的，使用此选项则使
用 1024 为单位
```

iptables 支持 extension 匹配。支持两种扩展匹配：使用"-p"时的隐式扩展和使用"-m"时的显式扩展。根据指定的扩展，随后可使用不同的选项。在指定扩展项的后面可使用"-h"来获取该扩展项的语法帮助。

"-p"选项指定的是隐式扩展，用于指定协议类型，每种协议类型都有一些子选项。常见协议和子选项如下说明：

```
-p tcp 子选项
  子选项：
    [!] --source-port,--sport port[:port]
          指定源端口号或源端口范围。指定端口范围时格式为"range_start:range_end"，最大范围为
0:65535。
    [!] --destination-port,--dport port[:port]
          指定目标端口号或目标端口号范围。
    [!] --tcp-flags mask comp
          匹配已指定的 tcp flags。mask 指定的是需要检查的 flag 列表，comp 指定的是必须设置的
flag。
          有效的 flag 值为：SYN ACK FIN RST URG PSH ALL NONE。
          如果以 0 和 1 来表示，意味着 mask 中 comp 指定的 flag 必须为 1 其余的必须为 0。
          例如：iptables -A FORWARD -p tcp --tcp-flags SYN,ACK,FIN,RST SYN
          表示只匹配设置了 SYN=1 而 ACK、FIN 和 RST 都为 0 数据包，也即只匹配 TCP 三次握手的第一
次握手。
    [!] --syn
         是"--tcp-flags SYN,ACK,FIN,RST SYN"的简写格式。

-p udp 子选项
  子选项：
    [!] --source-port,--sport port[:port]
          指定源端口号或源端口范围。指定端口范围时格式为"range_start:range_end"，最大范围为
0:65535。
    [!] --destination-port,--dport port[:port]
          指定目标端口号或目标端口号范围。

-p icmp 子选项
  子选项：
    [!] --icmp-type {type[/code]|typename}
          用于指定 ICMP 类型，可以是 ICMP 类型的数值代码或类型名称。有效的 ICMP 类型
```

可由 iptables -p icmp -h 获取。常用的是"echo-request"和"echo-reply"，分别表示 ping 和 pong，数值代号分别是 8 和 0
ping 时先请求后响应：ping 别人先出去 8 后进来 0；别人 ping 自己，先进来 8 后出去 0

"-m"选项指定的是显式扩展。其实隐式扩展也是要指定扩展名的，只不过默认已经知道所使用的扩展，于是可以省略。例如：-p tcp --dport = -p tcp -m tcp --dport。

常用的扩展和它们常用的选项如下：

(1).iprange：匹配给定的 IP 地址范围。

[!] --src-range from[-to]：匹配给定的源地址范围
[!] --dst-range from[-to]：匹配给定的目标地址范围

(2).multiport：离散的多端口匹配模块，如将 21、22、80 三个端口的规则合并成一条。

最多支持写 15 个端口，其中"555:999"算 2 个端口。只有指定了-p tcp 或-p udp 时该选项才生效。

[!] --source-ports,--sports port[,port|,port:port]...
[!] --destination-ports,--dports port[,port|,port:port]...
[!] --ports port[,port|,port:port]... ：不区分源和目标，只要是端口就行

(3).state：状态扩展。结合 ip_conntrack 追踪会话的状态。

[!] --state state
其中 state 有如下 4 种：
INVALID：非法连接（如 syn=1 fin=1）
ESTABLISHED：数据包处于已建立的连接中，它和连接的两端都相关联
NEW：新建连接请求的数据包，且该数据包没有和任何已有连接相关联
RELATED：表示数据包正在新建连接， 但它和已有连接是相关联的(如被动模式的 ftp 的命令连接和数据连接)

例如：-m state --state NEW,ESTABLISHED -j ACCEPT

关于这 4 个状态，在下文还有更详细的描述。

(4).string：匹配报文中的字符串。

--algo {kmp|bm}：两种算法，随便指定一种
--string "string_pattern"

如：

iptables -A OUTPUT -m string --algo bm --sting "taobao.com" -j DROP

(5).mac：匹配 MAC 地址，格式必须为 XX:XX:XX:XX:XX:XX。

[!] --mac-source address

(6).limit：使用令牌桶(token bucket)来限制过滤连接请求数。

--limit RATE[/second/minute/hour/day]：允许的平均数量。如每分钟允许 10 次 ping，即 6 秒一次 ping。默认为 3/hour。
--limit-burst：允许第一次涌进的并发数量。第一次涌进超出后就按 RATE 指定数来给出响应。默认值为 5。

例如：允许每分钟 6 次 ping，但第一次可以 ping 10 次。10 次之后按照 RATE 计算。所以，前 10 个 ping 包每秒能正常返回，从第 11 个 ping 包开始，每 10 秒允许一次 ping：iptables -A INPUT -d ServerIP -p icmp --icmp-type 8 -m limit --limit 6/minute --limit-burst 10 -j ACCEPT

(7).connlimit：限制每个客户端的连接上限。

```
--connlimit-above n：连接数量高于上限 n 个时就执行 TARGET
```

如最多只允许某 ssh 客户端建立 3 个 ssh 连接，超出就拒绝。两种写法：

```
iptables -A INPUT -d ServerIP -p tcp --dport 22 -m connlimit --connlimit-above 3 -j DROP
iptables -A INPUT -d ServerIP -p tcp --dport 22 -m connlimit ! --connlimit-above 3 -j ACCEPT
```

这个模块虽然限制能力不错，但要根据环境计算出网页正常访问时需要建立的连接数，另外还要考虑使用 NAT 转换地址时连接数会翻倍的问题。

最后剩下"-j"指定的 target 还未说明，target 表示对匹配到的数据包要做什么处理，比如丢弃 DROP、拒绝 REJECT、接受 ACCEPT 等，除这 3 个 target 外，还支持很多种 target。以下是其中几种：

```
DNAT：目标地址转换
SNAT:源地址转换
REDIRECT：端口重定向
MASQUERADE：地址伪装(其实也是源地址转换)
RETURN：用于自定义链，自定义链中匹配完毕后返回到自定义的前一个链中继续向下匹配
```

6.7.2 ip_conntrack 功能和 iptstate 命令

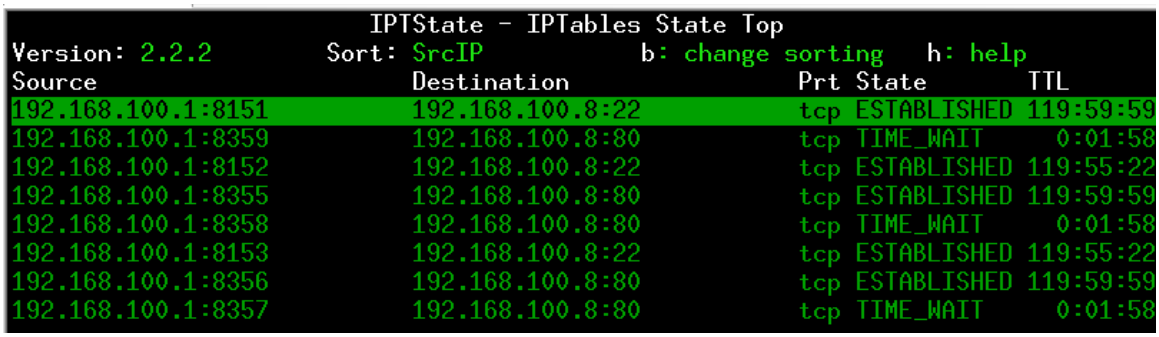
ip_conntrack 提供追踪功能，后来改称为 nf_conntrack，由 nf_conntrack 模块提供。

只要一加载该模块，/proc/net/nf_conntrack 文件中就会记录下追踪的连接状态。虽然会追踪 TCP/UDP/ICMP 的所有连接，但是在此文件中只保存 tcp 的连接状态。

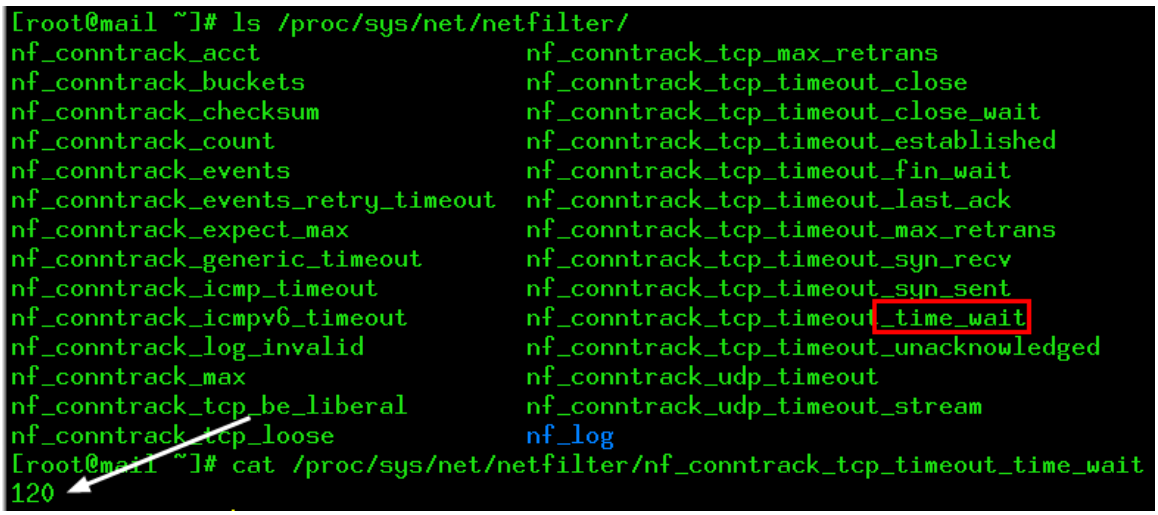
```
[root@mail ~]# cat /proc/net/nf_conntrack
ipv4     2 tcp      6 431714 ESTABLISHED src=192.168.100.1 dst=192.168.100.8 sport=1586 dport=22 src=192.168.100.8 dst=192.168.100.1 sport=22 dport=1586 [ASSURED] mark=0 secmark=0 use=2
ipv4     2 tcp      6 427822 ESTABLISHED src=192.168.100.8 dst=192.168.100.1 sport=22 dport=1343 src=192.168.100.1 dst=192.168.100.8 sport=1343 dport=22 [ASSURED] mark=0 secmark=0 use=2
ipv4     2 tcp      6 299 ESTABLISHED src=192.168.100.1 dst=192.168.100.8 sport=1608 dport=22 src=192.168.100.8 dst=192.168.100.1 sport=22 dport=1608 [ASSURED] mark=0 secmark=0 use=2
```

第一条显示的是 ESTABLISHED 状态的连接，该连接是 192.168.100.1:1586-->192.168.100.8:22，以及返回的连接 192.168.100.8:22 --> 192.168.100.1:1586。

也可以使用 iptstate 命令实时显示连接状态，它是像 top 工具一样的显示。该命令工具在 iptstate 包中，可能需要手动安装。如图为 iptstate 的一次结果。



可以发现 TTL（超时时间）值以及状态，还有其他的一些信息。这些 TTL 值的设置位置都在 /proc/sys/net/netfilter/目录下的文件中。例如上图中 web 服务 TIME_WAIT 的超时时间为 2 分钟，可以将其修改，对应的文件为下图中标示的。可以直接修改该文件的值，如 180 秒，即等待 TIME_WAIT 的时间 3 分钟后就断开连接。(TIME_WAIT 处于 TCP 连接 4 次挥手主动段开方的倒数第二个阶段)



nf_conntrack 好处是大大的，但是很悲剧，每一个监控和追踪的工具都是消耗性能的，而 nf_conntrack 也有其瓶颈所在。nf_conntrack 也会消耗一定的资源，所以在设计的时候默认给出了其最大的追踪数量，最大追踪数量值由/proc/sys/net/netfilter/nf_conntrack_max 文件决定。默认是 31384 个。这个值显然是无法满足较高并发量的服务器的，所以可以将其增大一些，否则追踪数达到了最大值后，后续的所有连接都将排队被阻塞，可能会因此给出警告。但是无论如何要明白的是追踪是会消耗性能的，所以该值应该酌情考虑。

```
[root@mail ~]# cat /proc/sys/net/netfilter/nf_conntrack_max
31384
```

并且要注意的一点是，nf_conntrack 模块不是一定需要显式装载才会被装载的，有些依赖它的模块被装载时该模块也会被装载。例如 iptables 命令中包含 iptables -t nat 时，就会装载该模块自动开启追踪，进而可能导致达到追踪 max 值而出错。

6.7.3 -m state 的状态解释

使用-m state 表示使用简称为"state"的模块。该模块提供 4 种状态：NEW、ESTABLISHED、RELATED 和 INVALID。但是这些状态和 TCP 三次握手四次挥手的十几种状态没任何关系。而且 state 提供的 4 种状态对于 tcp/udp/icmp 类型的数据包都是通用的。

注意：**这四种状态是数据包的状态，不是客户端或者服务器当时所处的状态**。也可以认为是防火墙 state 模块的状态，因为 state 模块在收到对应状态的包时会设置为相同的状态。

(1).NEW 状态与 TCP/UDP/ICMP 数据包的关系

为了建立一条连接，发送的第一个数据包(如 tcp 三次握手的第一次 SYN 数据包)的状态为 NEW。如果第一次连接没建立成功，则第二个继续请求的数据包已经不是 NEW 数据包了。

所以，如果不允许 NEW 状态的数据包表示不允许主动和对方建立连接，也不允许外界和本机建立连接。

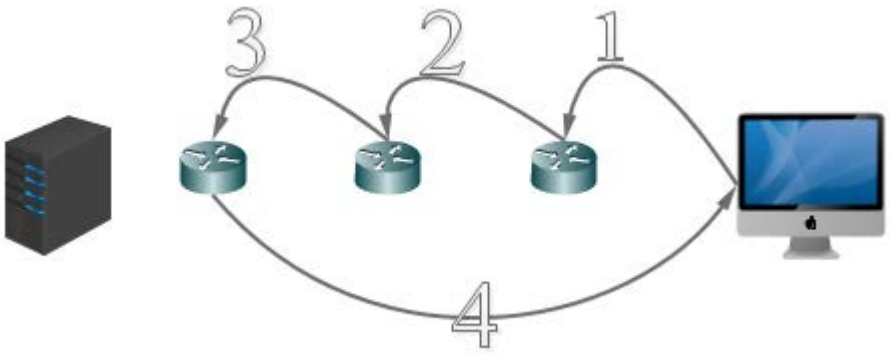
(2).ESTABLISHED 状态与 tcp/udp/icmp 数据包的关系

无论是 tcp 数据包、udp 数据包还是 icmp 数据包，**只要发送的请求数据包穿过了防火墙，那么接下来双方传输的数据包状态都是 ESTABLISHED，也就是说发过去的和返回回来的都是 ESTABLISHED 状态数据包。**

(3).RELATED 数据包的解释

对于 RELATED 数据包的解释是：与当前任何连接都无关，完全是被动或临时建立的连接之间传输的数据包。

例如，下图中 tracert 发送数据包的探测过程。



图中客户端为了探测服务器的地址发送了 tracert 命令。这个命令首先会标记一个 tcp 数据包的 TTL 值为 1，当数据包到达第一个路由器该值就减 1，所以 TTL 变为 0 表示该数据包到了寿终正寝该 DROP 掉的时候，然后该路由器就会发送一个 icmp 数据包（icmp-type=11）返回给客户端，这样客户端就知道了第一个路由器的地址。然后客户端的 tracert 命令继续标记一个 TTL 为 2 的数据包向外发送，直到第二个路由器才被丢弃，第二个路由器又发送一个 icmp 包给客户端，这样客户端就知道了第二个路由器的地址。同理第三次也一样。

在 tracert 探测的过程中，由路由器返回的 icmp 包都是 RELATED 状态的数据包。因为可以肯定的说，客户端发送给路由器的 tcp 数据包是走的一条连接，数据包被路由器丢弃后路由器发送的 icmp

数据包与原来的连接已经无关了，这是另外一条返回的连接。但是之所以有这个数据包，完全是因为前面的连接结束而产生的应答数据包。

不过 RELATED 状态和协议无关，只要数据包是因为本机先送出一个数据包而导致另一条连接的产生，那么这个新连接的所有数据包都属于 RELATED 状态的数据包。

这样就容易理解 ftp 被动模式设置的 related 状态了。在 ftp 服务器上的 21 号端口上开启了命令通道（也就是命令连接）后，以后无论是被动模式的随机数据端口还是主动模式的固定 20 数据端口，可以肯定的是数据通道的建立是由命令通道指定要开启的，所以这个数据通道中传输的数据包都是 RELATED 状态的。

(4).INVALID 状态的数据包

所谓的 INVALID 状态，就是恶意的数据包。只要不是 ESTABLISHED、NEW、RELATED 状态的数据包，那么就一定是 INVALID 状态。对于 INVALID 数据包最应该放在链中的第一条，以防止恶意的循环攻击。

(5).网关式防火墙的 NEW 状态、ESTABLISHED 状态和 RELATED

网关式的防火墙挡在客户端和服务器端中间，用于过滤或改变数据包，但是它的状态却不好判断了。

关于它的状态变化，可以总结为"墙头草"：客户端送到防火墙的数据包是什么状态，防火墙的 state 模块就设置为什么状态，转发给服务器的数据包就是什么状态；服务端发给防火墙的数据包是什么状态，防火墙的 state 模块就设置为什么状态，转发出去的数据包就是什么状态。也就是说，防火墙并不改变数据包状态的性质。

虽说数据包的状态只有防火墙才有资格判断，但是这样归纳却不妨碍理解。

例如，TCP 三次握手的第一次，客户端发送一个 SYN 数据包给服务器要建立连接，这个 SYN 数据包传到防火墙上，防火墙的 state 模块也会将自己的状态设置为 SYN_SENT，并认为这个数据包是 NEW 状态的数据包，然后转发给服务器，转发过程的数据包的状态也是 NEW。当服务器收到 SYN 后应答一个 SYN+ACK 数据包，当 SYN+ACK 数据包到达防火墙时，防火墙也和服务器一样将自己设置为 SYN_RECV 状态，并认为这个数据包已经是 ESTABLISHED 的数据包了，然后将这个数据包以 ESTABLISHED 的状态转发给客户端。

RELATED 状态也是一样的，只要双方的连接是"另起炉灶"的数据包，客户端和服务端之间的防火墙会随着数据包的流向而做一支"墙头草"。

其实这些状态以及转变都会在/proc/net/nf_conntrack 文件中记录，只是比较难以被人为追踪到。

6.7.4 filter-iptables 命令示例

iptables 实验主机地址：172.16.10.9。首先启动 iptables。

```
service iptables start
```

(1).清空自定义链、清空规则、清空规则计数器。

```
[root@xuexi ~]# iptables -X

[root@xuexi ~]# iptables -F
```

```
[root@xuexi ~]# iptables -Z
```

(2).允许 172.16.10.0 网段连接 ssh(端口 22)。

```
[root@xuexi ~]# iptables -A INPUT -s 172.16.10.0/24 -d 172.16.10.9 -p tcp --dport 22 -j ACCEPT

[root@xuexi ~]# iptables -A OUTPUT -s 172.16.10.9 -p tcp --sport 22 -j ACCEPT
```

(3).设置 filter 表默认规则为 DROP。

```
[root@xuexi ~]# iptables -P INPUT DROP

[root@xuexi ~]# iptables -P FORWARD DROP
```

一般防火墙对外是 ACCEPT 的，所以 OUTPUT 链采用默认的 ACCEPT。

由于将 INPUT 链设置为全部 DROP，因此除了前面设置的目标为 22 端口的数据包允许通过，其余全部丢弃，即使是 ping 环回地址。

```
[root@xuexi ~]# ping -c 4 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.

--- 127.0.0.1 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 13000ms
```

(4).查看规则列表和统计数据。

```
[root@xuexi ~]# iptables -L -n
Chain INPUT (policy DROP)
target     prot opt source               destination
ACCEPT     tcp  --  172.16.10.0/24      172.16.10.9         tcp dpt:22

Chain FORWARD (policy DROP)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     tcp  --  172.16.10.9         0.0.0.0/0           tcp spt:22
```

如果加上"-v"选项，则会显示每条规则上的流量统计数据。

```
[root@xuexi ~]# iptables -L -n -v
Chain INPUT (policy DROP 10 packets, 993 bytes)
 pkts bytes target     prot opt in     out     source               destination
  655 64963 ACCEPT     tcp  --  *      *       172.16.10.0/24       172.16.10.9         tcp
dpt:22

Chain FORWARD (policy DROP 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
```

```
Chain OUTPUT (policy ACCEPT 9 packets, 756 bytes)
 pkts bytes target     prot opt in     out     source               destination
  242 42857 ACCEPT     tcp  --  *      *       172.16.10.9          0.0.0.0/0           tcp
spt:22
```

(3).放行环回设备的进出数据包(环回地址的放行很重要)。

```
[root@xuexi ~]# iptables -A INPUT -i lo -s 127.0.0.1 -d 127.0.0.1 -j ACCEPT
[root@xuexi ~]# iptables -A OUTPUT -o lo -s 127.0.0.1 -d 127.0.0.1 -j ACCEPT
```

此时已经可 ping 127.0.0.1。

```
[root@xuexi ~]# ping 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.067 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.051 ms
^C
--- 127.0.0.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1648ms
rtt min/avg/max/mdev = 0.051/0.059/0.067/0.008 ms
```

但是不建议直接写 127.0.0.1，而是省略目标地址和源地址，因为 ping 本机 ip 地址最后交给环回设备但是不是交给 127.0.0.1 的，而是交给 127.0.0 网段的其他地址。所以应该这么写：

```
[root@xuexi ~]# iptables -A INPUT -i lo -j ACCEPT
[root@xuexi ~]# iptables -A OUTPUT -o lo -j ACCEPT
```

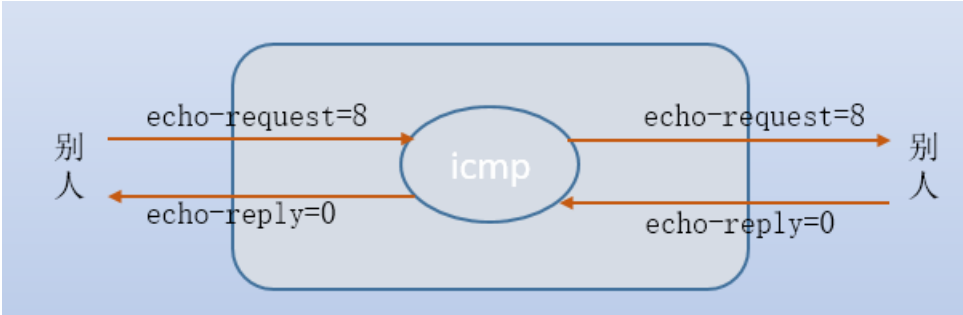
所以可以将前面多余的规则删除掉。

```
[root@xuexi ~]# iptables -D INPUT 2
[root@xuexi ~]# iptables -D OUTPUT 2
```

(4).能自己 ping 自己的 IP，也能 ping 别人的 IP，但是别人不能 ping 自己。

ping 的过程实际上是 ping 请求对方，然后对方 pong 回应，协议类型为 icmp。其中 ping 请求时，icmp 类型为 echo-request，数值代号为 8，pong 回应时的 icmp 类型为 echo-reply，数值代号为 0。

所以本机向外 ping 时，流出的是 echo-request 数据包，流入的是 echo-reply 数据包。而外界 ping 本机时，则是流入 echo-request 数据包，流出 echo-reply 数据包。因此，要允许本机向外 ping，只需允许 icmp-type=8 的流出包、icmp-type=0 的流入包即可，又由于前面的试验中设置了 INPUT 和 OUTPUT 链的默认规则为 DROP，所以外界主机无法 ping 本机。




```
[root@xuexi ~]# iptables -A OUTPUT -p icmp --icmp-type=8 -j ACCEPT

[root@xuexi ~]# iptables -A INPUT -p icmp --icmp-type=0 -j ACCEPT
```

当然，OUTPUT 链本身就是放行所有数据包的，所以只需写 INPUT 链规则即可。

(5).安装 httpd，让外界能够访问 web 页面(端口为 80)。

```
[root@xuexi ~]# iptables -A INPUT -d 172.16.10.9 -p tcp --dport 80 -j ACCEPT

[root@xuexi ~]# iptables -A OUTPUT -s 172.16.10.9 -p tcp --sport 80 -j ACCEPT
```

(6).删除（或替换）放行 ssh 服务和 web 服务的规则，并写出基于 ip_conntrack 放行 ssh 和 web 的规则(进入的数据包的状态只可能会是 NEW 和 ESTABLISHED，出去的状态只可能是 ESTABLISHED)

放行 ssh：

```
iptables -R INPUT 1 -s 172.16.10.0/24 -d 172.16.10.9 -p tcp --dport 22 -m state --state=NEW,ESTABLISHED -j ACCEPT

iptables -R OUTPUT 1 -s 172.16.10.9 -p tcp --sport 22 -m state --state=ESTABLISHED -j ACCEPT
```

放行 web:

```
iptables -R INPUT 4 -d 172.16.10.9 -p tcp --dport 80 -m state --state=NEW,ESTABLISHED -j ACCEPT

iptables -R OUTPUT 4 -s 172.16.10.9 -p tcp --sport 80 -m state --state=ESTABLISHED -j ACCEPT

iptables -L -n --line-number
```

```
Chain INPUT (policy DROP)
num  target     prot opt source               destination
1    ACCEPT     tcp  --  172.16.10.0/24       172.16.10.9         tcp dpt:22 state
NEW,ESTABLISHED
2    ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
3    ACCEPT     icmp --  0.0.0.0/0            0.0.0.0/0           icmp type 0
4    ACCEPT     tcp  --  0.0.0.0/0            172.16.10.9         tcp dpt:80 state
NEW,ESTABLISHED

Chain FORWARD (policy DROP)
```

```
num  target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
num  target     prot opt source               destination
1    ACCEPT     tcp  --  172.16.10.9          0.0.0.0/0           tcp spt:22 state
ESTABLISHED
2    ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
3    ACCEPT     icmp --  0.0.0.0/0            0.0.0.0/0           icmp type 8
4    ACCEPT     tcp  --  172.16.10.9          0.0.0.0/0           tcp spt:80 state
ESTABLISHED
```

这样的设置使得外界可以和主机建立会话，但是由主机出去的数据包则一定只能是 ESTABLISHED 状态的服务发出的，这样本机想主动和外界建立会话是不可能的。这样就实现了状态监测的功能，防止黑客通过开放的 22 端口或 80 端口植入木马并主动联系黑客。

(7).放行外界 ping 自己，但是要基于 ip_conntrack 来放行。

```
iptables -A INPUT -d 172.16.10.9 -p icmp --icmp-type=8 -m state --
state=NEW,ESTABLISHED -j ACCEPT

iptables -A OUTPUT -s 172.16.10.9 -p icmp --icmp-type=0 -m state --
state=ESTABLISHED -j ACCEPT

iptables -L -n --line-number
```

```
Chain INPUT (policy DROP)
num  target     prot opt source               destination
1    ACCEPT     tcp  --  172.16.10.0/24       172.16.10.9         tcp dpt:22 state
NEW,ESTABLISHED
2    ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
3    ACCEPT     icmp --  0.0.0.0/0            0.0.0.0/0           icmp type 0
4    ACCEPT     tcp  --  0.0.0.0/0            172.16.10.9         tcp dpt:80 state
NEW,ESTABLISHED
5    ACCEPT     icmp --  0.0.0.0/0            172.16.10.9         icmp type 8 state
NEW,ESTABLISHED

Chain FORWARD (policy DROP)
num  target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
num  target     prot opt source               destination
1    ACCEPT     tcp  --  172.16.10.9          0.0.0.0/0           tcp spt:22 state
ESTABLISHED
2    ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
3    ACCEPT     icmp --  0.0.0.0/0            0.0.0.0/0           icmp type 8
4    ACCEPT     tcp  --  172.16.10.9          0.0.0.0/0           tcp spt:80 state
ESTABLISHED
```

```
5    ACCEPT     icmp --  172.16.10.9          0.0.0.0/0           icmp type 0 state
ESTABLISHED
```

(8).安装 vsftpd，并设置其防火墙。

由于 ftp 有主动模式和被动模式，被动模式的数据端口不定，且使用哪种模式是由客户端决定的，这使得 ftp 的防火墙设置比较复杂，但是借助 netfilter 的 state 模块，设置就简单的多了。

首先装载其专门的模块 nf_conntrack_ftp。

```
[root@xuexi ~]# modprobe nf_conntrack_ftp
```

也可以写入/etc/sysconfig/iptables-config 的"IPTABLES_MODULES="nf_conntrack_ftp"。

然后编写规则：放行 21 端口的进入数据包，放行 related 关联数据包，放行出去的包(放行出去的包是前面已默认的，但此处为了试验完整性，还是显式指定了)。

```
iptables -A INPUT -d 172.16.10.9 -p tcp --dport 21 -m state --
state=NEW,ESTABLISHED -j ACCEPT

iptables -A INPUT -d 172.16.10.9 -m state --state=RELATED,ESTABLISHED -j
ACCEPT

iptables -A OUTPUT -s 172.16.10.9 -m state --state=ESTABLISHED -j ACCEPT
```

上面一个规则中多个状态列表，状态列表中的状态是“或”的关系，满足其一即可。

```
[root@xuexi ~]# iptables -L -n --line-number
Chain INPUT (policy DROP)
num  target     prot opt source               destination
1    ACCEPT     tcp  --  172.16.10.0/24       172.16.10.9         tcp dpt:22 state
NEW,ESTABLISHED
2    ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
3    ACCEPT     icmp --  0.0.0.0/0            0.0.0.0/0           icmp type 0
4    ACCEPT     tcp  --  0.0.0.0/0            172.16.10.9         tcp dpt:80 state
NEW,ESTABLISHED
5    ACCEPT     icmp --  0.0.0.0/0            172.16.10.9         icmp type 8 state
NEW,ESTABLISHED
6    ACCEPT     tcp  --  0.0.0.0/0            172.16.10.9         tcp dpt:21 state
NEW,ESTABLISHED
7    ACCEPT     all  --  0.0.0.0/0            172.16.10.9         state RELATED,ESTABLISHED

Chain FORWARD (policy DROP)
num  target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
num  target     prot opt source               destination
1    ACCEPT     tcp  --  172.16.10.9          0.0.0.0/0           tcp spt:22 state
ESTABLISHED
2    ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
```

```
3    ACCEPT     icmp --  0.0.0.0/0            0.0.0.0/0           icmp type 8
4    ACCEPT     tcp  --  172.16.10.9          0.0.0.0/0           tcp spt:80 state
ESTABLISHED
5    ACCEPT     icmp --  172.16.10.9          0.0.0.0/0           icmp type 0 state
ESTABLISHED
6    ACCEPT     all  --  172.16.10.9          0.0.0.0/0           state ESTABLISHED
```

现在 iptables 已经有很多规则，但是也足够乱的。不仅想看懂挺复杂，在数据包检查的时候性能也更差，所以有必要将它们合并成简单易懂的规则。

6.7.5 合并规则以及调整规则的顺序

执行 iptables-save 命令，可以 dump 出当前内核维护的 netfilter 指定表中的规则，默认导出 filter 表。

```
[root@xuexi ~]# iptables-save -t filter
# Generated by iptables-save v1.4.7 on Sun Aug 13 05:33:29 2017
*filter
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]
-A INPUT -s 172.16.10.0/24 -d 172.16.10.9/32 -p tcp -m tcp --dport 22 -m state --state
NEW,ESTABLISHED -j ACCEPT
-A INPUT -i lo -j ACCEPT
-A INPUT -p icmp -m icmp --icmp-type 0 -j ACCEPT
-A INPUT -d 172.16.10.9/32 -p tcp -m tcp --dport 80 -m state --state NEW,ESTABLISHED -j
ACCEPT
-A INPUT -d 172.16.10.9/32 -p icmp -m icmp --icmp-type 8 -m state --state NEW,ESTABLISHED -
j ACCEPT
-A INPUT -d 172.16.10.9/32 -p tcp -m tcp --dport 21 -m state --state NEW,ESTABLISHED -j
ACCEPT
-A INPUT -d 172.16.10.9/32 -m state --state RELATED,ESTABLISHED -j ACCEPT
-A OUTPUT -s 172.16.10.9/32 -p tcp -m tcp --sport 22 -m state --state ESTABLISHED -j ACCEPT
-A OUTPUT -o lo -j ACCEPT
-A OUTPUT -p icmp -m icmp --icmp-type 8 -j ACCEPT
-A OUTPUT -s 172.16.10.9/32 -p tcp -m tcp --sport 80 -m state --state ESTABLISHED -j ACCEPT
-A OUTPUT -s 172.16.10.9/32 -p icmp -m icmp --icmp-type 0 -m state --state ESTABLISHED -j
ACCEPT
-A OUTPUT -s 172.16.10.9/32 -m state --state ESTABLISHED -j ACCEPT
COMMIT
# Completed on Sun Aug 13 05:36:22 2017
```

从导出结果中可以看到，input 链中有好几条规则都是针对 state=NEW,ESTABLISHED 而建立的，同理 OUTPUT 链中的 state=ESTABLISHED，且他们的 target 都是一样的，这样的规则可以考虑是否能够合并。

注意：环回接口 lo 一定要显式指定所有类型的数据都通过。

例如，先将 OUTPUT 链中 state=ESTABLISHED 的规则进行合并。

```
[root@xuexi ~]# iptables -I OUTPUT 1 -s 172.16.10.9 -m state --state=ESTABLISHED -j ACCEPT
```

再将 OUTPUT 链中除了 lo 接口的所有规则删除掉即可。

最终 OUTPUT 链剩下以下两条规则。

```
-A OUTPUT -s 172.16.10.9/32 -m state --state ESTABLISHED -j ACCEPT
-A OUTPUT -o lo -j ACCEPT
```

虽然这样使得 OUTPUT 没有再限制端口和协议，但对于流出数据包而言，这样已经足够了。一般来说，**OUTPUT 链的定义方式是：默认所有数据出去，但禁止某些端口(如 80)发送 NEW、INVALID 状态的包。所以，在 OUTPUT 链默认规则为 ACCEPT 的情况下，可以参照如下方式定义该链的规则：**

```
-A OUTPUT -s 172.16.10.9/32 -p tcp -m multiport --sports 21,22,80 -m state --state NEW,INVALID -j DROP
```

现在规则如下：

```
[root@xuexi ~]# iptables -L -n --line-number
Chain INPUT (policy DROP)
num  target     prot opt source               destination
1    ACCEPT     tcp  --  172.16.10.0/24       172.16.10.9         tcp dpt:22 state NEW,ESTABLISHED
2    ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
3    ACCEPT     icmp --  0.0.0.0/0            0.0.0.0/0           icmp type 0
4    ACCEPT     tcp  --  0.0.0.0/0            172.16.10.9         tcp dpt:80 state NEW,ESTABLISHED
5    ACCEPT     icmp --  0.0.0.0/0            172.16.10.9         icmp type 8 state NEW,ESTABLISHED
6    ACCEPT     tcp  --  0.0.0.0/0            172.16.10.9         tcp dpt:21 state NEW,ESTABLISHED
7    ACCEPT     all  --  0.0.0.0/0            172.16.10.9         state RELATED,ESTABLISHED

Chain FORWARD (policy DROP)
num  target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
num  target     prot opt source               destination
1    DROP       tcp  --  172.16.10.9          0.0.0.0/0           multiport sports 21,22,80 state INVALID,NEW
```

再来合并 INPUT 链的规则。INPUT 链的合并应该遵循这样一种规则：先定义好大规则，再逐渐向后添加更具体、针对性更强的小规则。

首先将 INPUT 链中第 4、6 两条规则合并。

```
iptables -I INPUT 4 -d 172.16.10.9 -p tcp -m multiport --dport 21,80 -m state --state=NEW,ESTABLISHED -j ACCEPT
```

再合并 INPUT 链中对内和对外的 ping 规则。

```
iptables -I INPUT 3 -p icmp --icmp-type any -m state --state=NEW,ESTABLISHED -j ACCEPT
```

再删除被合并的多余规则。最终 INPUT 链中规则列表如下：

```
-A INPUT -s 172.16.10.0/24 -d 172.16.10.9/32 -p tcp -m tcp --dport 22 -m state --state NEW,ESTABLISHED -j ACCEPT
-A INPUT -i lo -j ACCEPT
-A INPUT -p icmp -m icmp --icmp-type any -m state --state NEW,ESTABLISHED -j ACCEPT
-A INPUT -d 172.16.10.9/32 -p tcp -m multiport --dports 21,80 -m state --state NEW,ESTABLISHED -j ACCEPT
-A INPUT -d 172.16.10.9/32 -m state --state RELATED,ESTABLISHED -j ACCEPT
```

之后再有其他需求时，只需在此基础上添加更细致、具体的规则即可。如禁止外界 ping 本机，只需在第三条 INPUT 规则前 DROP 掉进来的 icmp-type=8 的包即可。

从上面的规则列表中，也许已经发现了顺序不是很易读。规则列表中规则的顺序是至关重要的，不仅影响易读性，还影响检查的顺序从而影响性能，例如大并发量的数据包应该尽早匹配。以下是调整 INPUT 链中规则顺序的几个建议：

(1).请求量大的尽量放前面。

(2).通用型的规则尽量放前面。

(3).直接拒绝的考虑放前面，主要是防恶意的循环攻击，对于个别拉黑但非攻击意图的其实无需放前面。

其实，iptables 服务脚本配置文件/etc/sysconfig/iptables 中初始的规则就是最佳的框架。

```
[root@xuexi ~]# cat /etc/sysconfig/iptables
# Firewall configuration written by system-config-firewall
# Manual customization of this file is not recommended.
*filter
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
-A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
-A INPUT -p icmp -j ACCEPT
-A INPUT -i lo -j ACCEPT
-A INPUT -m state --state NEW -m tcp -p tcp --dport 22 -j ACCEPT
-A INPUT -j REJECT --reject-with icmp-host-prohibited
-A FORWARD -j REJECT --reject-with icmp-host-prohibited
COMMIT
```

之后有任何特定的需求，都可以直接在这些初始规则基础上进行追加。

6.8 规则的管理方法

6.8.1 保存规则

使用 iptables 写的规则都存放在内存中，内核会维护 netfilter 的每张表中的规则，所以重启 iptables"服务"会使内存中的规则列表全部被清空。

要想手动写的规则长期有效，需要将规则保存到持久存储文件中，例如 iptables"服务"启动时默认加载的脚本配置文件/etc/sysconfig/iptables。

有两种方法保存规则。

方法一：直接保存到/etc/sysconfig/iptables 中

```
service iptables save
```

存方法二：可自定义保存位置

```
iptables-save >/etc/sysconfig/iptables

iptables-save >/etc/sycofnig/iptables.20170103
```

恢复规则的方法:

```
iptables-restore </etc/sysconfig/iptables

iptables-restore </etc/sysconfig/iptables.2170103
```

6.8.2 规则的管理方法

虽然将规则放入/etc/sysconfig/iptables 文件中可以每次加载 netfilter 都能应用相应的规则，但是极其不建议这么做，否则将来很有可以能会欲哭无泪。假如防火墙规则数据库中关于 192.168.100.8 的规则有 100 条，但是该主机改了 IP 地址，难道要去修改所有 192.168.100.8 的规则吗。

管理规则更好的方法是写成脚本。将这些规则全部写入到一个 shell 脚本中，并对多次重复的地址使用变量，例如服务器的地址或网段，内网的网段。如下图所示：

```
#!/bin/bash

IPT=/sbin/iptables
SERVER=192.168.100.8
INT=172.16.100.5

#Remove any existing rules
$IPT -F

#setting default firewall policy
$IPT --policy OUTPUT ACCEPT
$IPT --policy FORWARD DROP
$IPT -P INPUT DROP
```

使用脚本的优点有：

1.管理的便捷。写成脚本可以直接修改该文件，要重新生效时只需执行一次该脚本文件即可。但是要注意脚本的第一条命令最好是 iptables -F，这样每次运行脚本都会先清空已有规则再加载脚本中的其他规则。

2.可以在脚本中使用变量。这样以后某台服务器地址改变了只需修改该服务器地址对应的变量值即可。

3.容易阅读，因为可以加注释，并通过注释来对规则进行分类，未来修改就变得相对容易的多。

4.备份规则变得更容易。备份后，即使硬盘坏了导致现有的规则丢失了也可以简单的拷贝一个脚本过去运行即可，而不用再一条一条命令的敲。

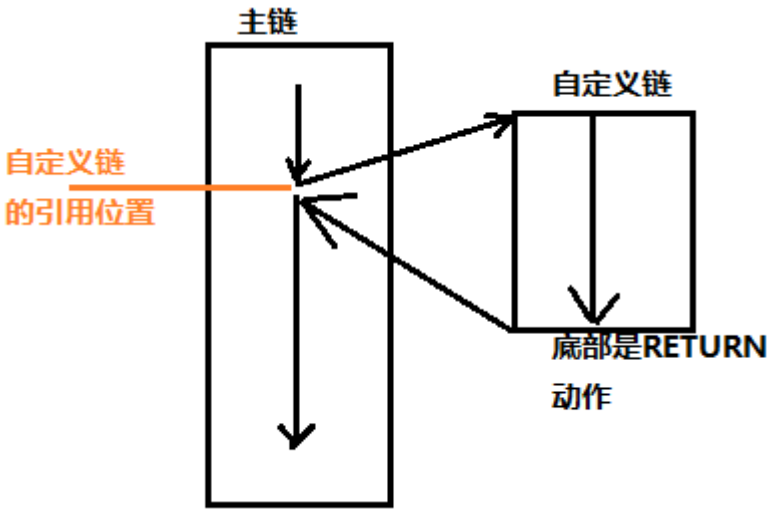
5.也可以实现开机加载规则。只需在/etc/rc.d/rc.local 中加上一条执行该脚本的命令即可。

6.可以将其加入任务计划。

6.9 自定义链

自定义链是被主链引用的。引用位置由"-j"指定自定义链名称，表示跳转到自定义链并匹配其内规则列表。

例如，在 INPUT 链中的第三条规则为自定义链的引用规则，则数据包匹配到了第三条时进入自定义链匹配，匹配完自定义链后如果定义了返回主链的 RETURN 动作，则返回主链继续向下匹配，如果没有定义 RETURN 动作，则匹配结束。



创建一条自定义链。

```
iptables -N mychain
```

向其中加入一些基于安全的攻防规则，让每次数据包进入都匹配一次攻防链。

```
iptables -A mychain -d 255.255.255.255 -p icmp -j DROP
iptables -A mychain -d 192.168.255.255 -p icmp -j DROP
iptables -A mychain -p tcp ! --syn -m state --state NEW -j DROP
iptables -A mychain -p tcp --tcp-flags ALL ALL -j DROP
iptables -A mychain -p tcp --tcp-flags ALL NONE -j DROP
```

在自定义链中的最后一条加上一条返回主链的规则，表示匹配完自定义后继续回到主链进行匹配。

```
iptables -A mychain -d 192.168.100.8 -j RETURN
```

在主链的适当位置加上一条引用主链的规则。表示数据包匹配到了这个位置开始进入自定义链匹配，如果自定义链都没被匹配而是被最后的 RETURN 规则匹配，则回到主链再次匹配。

```
iptables -I INPUT -d 192.168.100.8 -j mychain
```

删除自定义链：需要先清空自定义链，去除被引用记录，然后使用-X 删除空的自定义链。

```
iptables -F mychain
iptables -D INPUT 1
iptables -X mychain
```

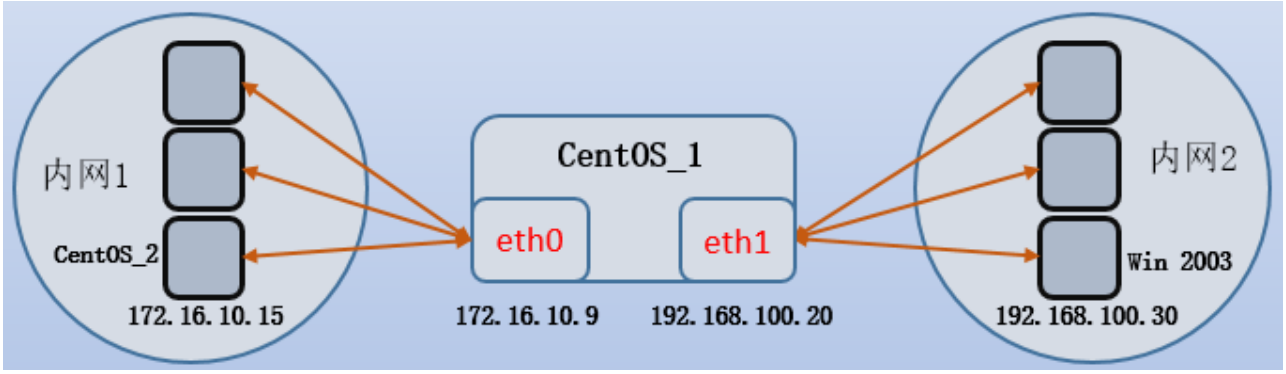
可以使用-E 命令重命名自定义链。

6.10 NAT

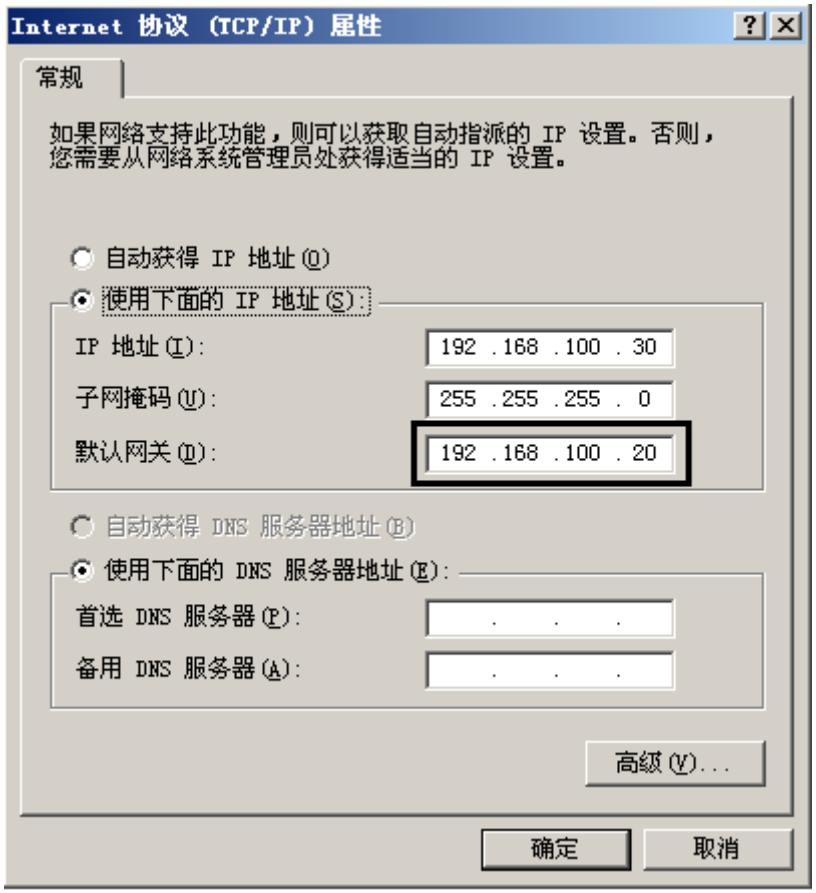
6.10.1 配置网关以及转发

首先是一个网关配置实验。

以 CentOS_1 作为两边内网的网关，让内网 1 和内网 2 可以互相通信。试验过程中，先关闭 CentOS_1 的防火墙。



首先配置 Windows Server 2003 和 CentOS_2 的网关指向 CentOS_1。




```
[root@xuexi ~]# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags M
0.0.0.0         172.16.10.9     0.0.0.0         UG    0
172.16.10.0     0.0.0.0         255.255.255.0   U     1
[root@xuexi ~]#
```

目前 CentOS_1 还没有打开转发功能。测试 CentOS_2 和 Windows Server 2003 都能 ping 通 CentOS_1 的两个地址，但 CentOS_2 和 Windows Server 2003 两者无法互相 ping 通。

为什么到两个内网到 CentOS_1 的两个地址都通，但是到对方内网却不通呢？

```
[root@xuexi ~]# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.100.0   0.0.0.0         255.255.255.0   U     0      0        0 eth1
172.16.10.0     0.0.0.0         255.255.255.0   U     0      0        0 eth0
169.254.0.0     0.0.0.0         255.255.0.0     U     1002   0        0 eth0
169.254.0.0     0.0.0.0         255.255.0.0     U     1003   0        0 eth1
0.0.0.0         172.16.10.2     0.0.0.0         UG    0      0        0 eth0
```

对于 CentOS_1 主机而言，**网络是内核空间中的内容，两个 IP 地址都属于主机而非属于网卡，内核知道 eth0 和 eth1 的存在**。由于默认在路由表中有对应内网 1 和内网 2 的路由条目，这两个路由条目用于维持和自己所在网段的地址通信(Iface 列指定了从 eth1 和 eth0 流出去)。

当 CentOS_2 发起 ping CentOS_1:eth1 的请求时，ping 请求包从 eth0 接口进入 CentOS_1，进入后被路由决策一次，内核发现这个数据包的目标地址是 eth1，可以直接应答给 CentOS_2，于是产生 pong 响应包并被路由一次，决定从 eth0 出去，最终回复给了 CentOS_2。同理从 eth1 进来目标地址是 eth0 的数据包也是一样处理的。这里面并没有涉及到数据包转发的过程。

但内网 1 主机在 ping 内网 2 主机时，在 CentOS_1 上却需要数据包的转发。因为数据包到达 eth0 上时数据包的目标地址是 win server 2003 的地址 172.16.10.30，路由决策时发现是和 eth1 同网段的主机，但却不是本机，于是决定从 eth1 流出去。要完成这个过程，需要将数据包完完整整地从 eth0 交给 eth1，这要求 CentOS_1 主机能够完成转发，但没有开启 ip_forward 功能时是无法转发的，因此从 eth0 流入的数据包被丢弃，导致内网 1 主机 ping 不通内网 2 主机。

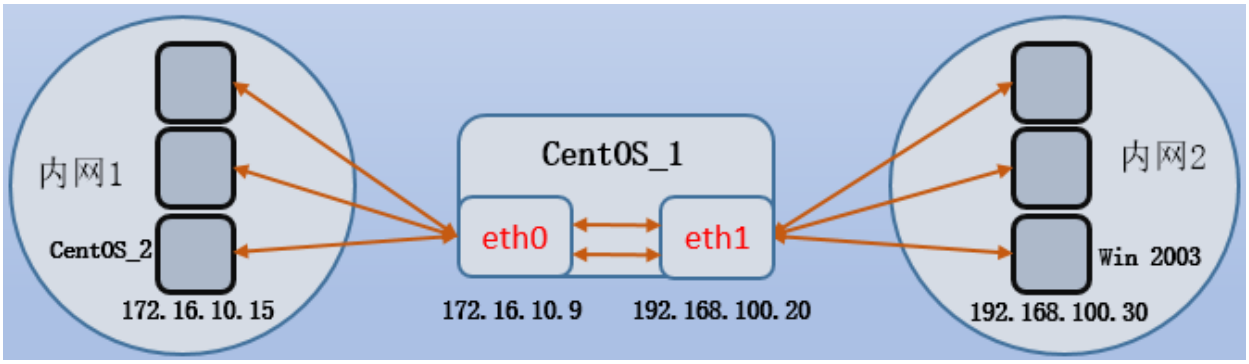
在 CentOS_1 上开启转发功能。

```
[root@xuexi ~]# echo 1 > /proc/sys/net/ipv4/ip_forward
```

再使用 CentOS_2 来 ping Windows Server 2003，结果一定是通的，如果没有通，考虑 CentOS_1 是否开启了防火墙。

6.10.2 配置网关防火墙

打开转发后就可以对 filter 表的 FORWARD 链进行设置了：只要是被 forward 的数据包都会受到防火墙的"钩子伺候"，并进行一番检查。



要注意的是，此时防火墙是负责两个网段的，转发后的数据包状态并不会因为经过 FORWARD 而改变。例如内网 1 发出的 NEW 状态的数据包到内网 2 时途经 FORWARD 时，如果允许通过则转发出去的数据包还是 NEW 状态的，这样也就保证了内网 2 接受到的数据包还是 NEW 状态的，如果内网 2 也配置一个单机防火墙就可以判断这是 NEW 状态的数据包从而进行相关的规则过滤。

例如：

```
iptables -P FORWARD DROP
```

此时已经内网 1 和内网 2 相互 ping 不通了。加上下面的规则，内网 1 的 CentOS_2 就能 ping 通外面，且外面进来的数据包都只能是 ESTABLISHED 状态的。

```
iptables -A FORWARD -m state --state NEW,ESTABLISHED -j ACCEPT
```

再加上这两条，就能保证内网 2 只能向内网 1 主机的 22 和 80 端口发起 NEW 和 ESTABLISHED 状态的数据包，且内网 1 只能向外发送 ESTABLISHED 状态的数据包。

```
iptables -A FORWARD -d 172.16.10.15 -p tcp -m multiport --dports 22,80 -m state --state NEW,ESTABLISHED -j ACCEPT

iptables -A FORWARD -s 172.16.10.15 -m state --state ESTABLISHED -j ACCEPT
```

6.10.3 SNAT 和 DNAT

NAT 依赖于 ip_forward，因此需要先开启它。NAT 的基础是 nf_conntrack，用来记录 NAT 表的映射关系。

注：从内核 2.6.34 开始，NAT 表支持操作 INPUT 链。它只为 SNAT 服务。和 snat on postrouting 类似，只不过 snat on input 用来转换"目标是本机的数据包"的源地址。

NAT 有三个作用：

◇ 地址转换。让内网(私有地址)可以共用一个或几个公网地址连接 Internet。这是从内向外的，需要在网关式防火墙的 POSTROUTING 处修改源地址，这是 SNAT 功能。

◇ 保护内网服务器。内网主机连接 Internet 使用的是公网地址，对外界而言是看不到内网服务器地址的，所以外界想要访问内部主机只能经过防火墙主机的公网地址，然后将目标地址转换为内网服务器地址，这起到了保护内网服务器的作用。转换目标地址需要在网关式防火墙的 PREROUTING 链处修改，这是 DNAT 功能。

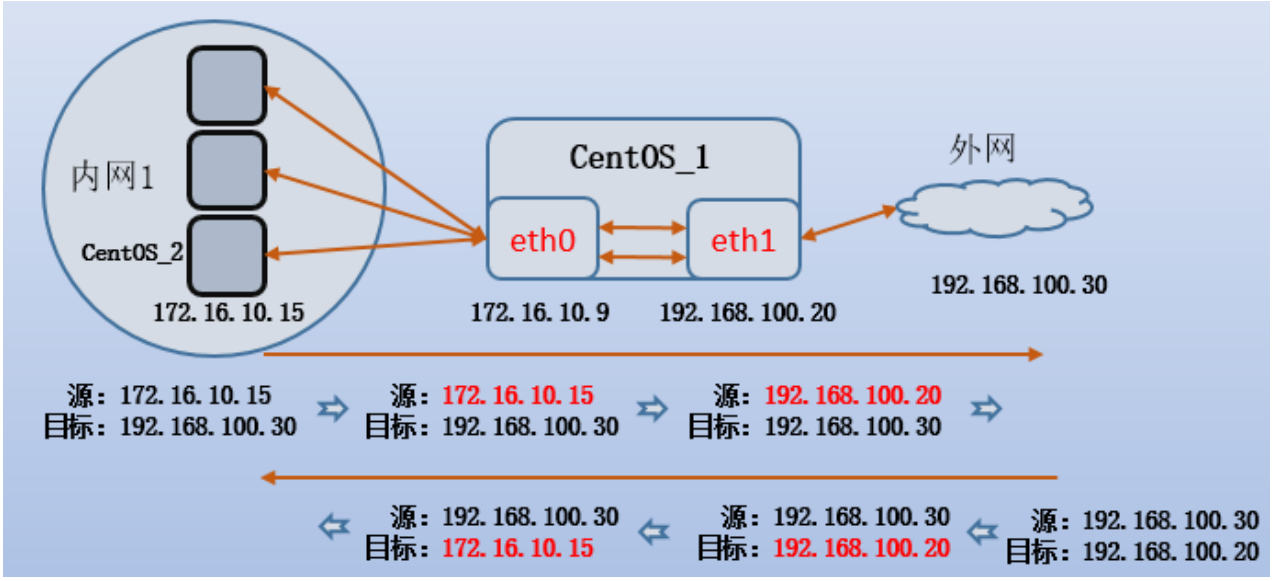
◇ 不仅可以修改目标地址，还可以使用端口映射功能。DNAT 是修改目标地址，端口映射是修改目标端口。如将 web 服务器的 8080 端口映射为防火墙的 80 端口。

外网主机和内网主机通信有两种情况的数据包：一种情况是建立 NEW 状态的新请求连接数据包，一种是回应的数据包。无论哪种情况，在 NEW 状态数据包经过地址转换之后会在防火墙内存中维护一张 NAT 表，保存未完成连接的地址转换记录，这样在回应数据包到达防火墙时可以根据这些记录路由给正确的主机。

也就是说，SNAT 主要应付的是内部主机连接到 Internet 的源地址转换，转换的位置是 POSTROUTING 链；DNAT 主要应付的是外部主机连接内部服务器防止内部服务器被攻击的目标地址转换，转换位置在 PREROUTING；端口映射可以在多个地方转换。

(1).SNAT 转换源地址

SNAT 过程的数据包转换过程如下图所示。



注意 SNAT 设置在 postrouting 链，流出接口 eth1。由于规则中有其他的条件存在，使得可以大多数时候不用指定流出接口，当然如果指定的话更完整。

```
iptables -t NAT -A POSTROUTING -s 172.16.10.0/24 -o eth1 -j SNAT --to-source 192.168.100.20

iptables -t NAT -A POSTROUTING -s 172.16.10.0/24 -o eth1 -j SNAT --to-source 192.168.100.20-192.168.100.25

iptables -t NAT -A POSTROUTING -s 172.16.10.0/24 -o eth1 -j MASQUERADE
```

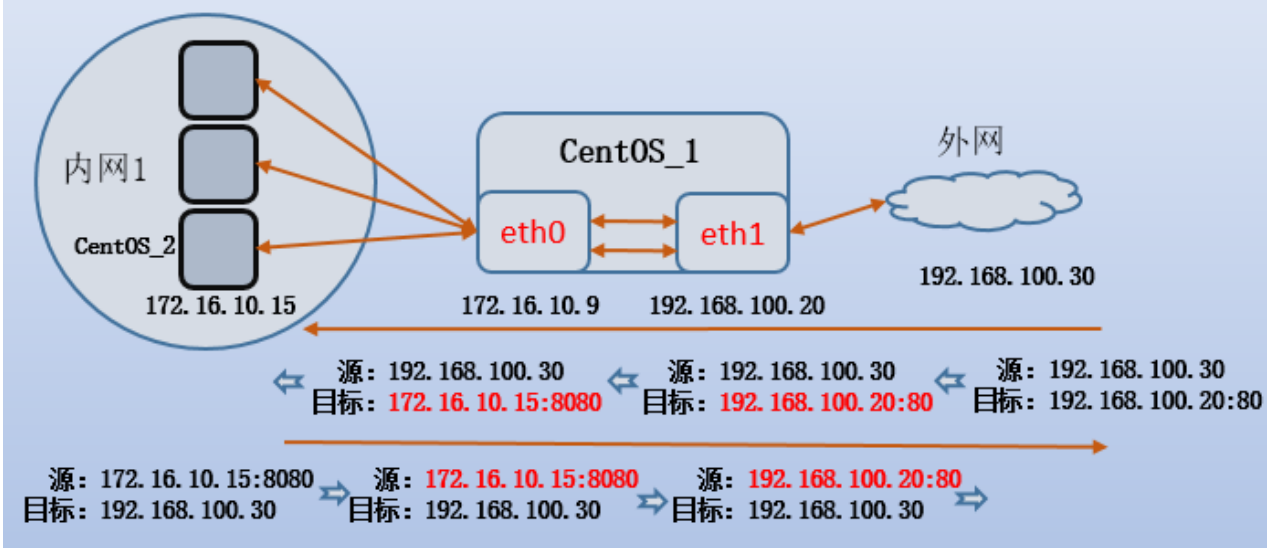
第一条语句表示将内网 1 向外发出的数据包进行处理，将源地址转换为 192.168.100.20。

第二条语句表示将源地址转换成 192.168.100.{20-25}之间的某个地址。

第三条语句表示动态获取流出接口 eth1 的地址，并将源地址转换为此地址。这称为地址伪装(ip masquerade)，地址伪装功能很实用，但是相比前两种，性能要稍差一些，因为处理每个数据包时都要获取 eth1 的地址，也就是说多了一个查找动作。

(2).DNAT 目标地址和端口转换

DNAT 过程的数据包转换过程如下图所示。



注意 DNAT 设置在 prerouting 链，流入接口 eth1。

```
iptables -t NAT -A PREROUTING -i eth1 -d 192.168.100.20 -p tcp --dport 80 -j DNAT --to-destination 172.16.10.15:8080
```

上面的语句表示从外网流入的目标为 192.168.100.20:80 的数据包转换为目标 172.16.10.15:8080，于是该数据包被路由给内网 1 主机 172.16.10.15 的 8080 端口上。