边缘安全加速平台 EO

四层代理

产品文档

腾讯云

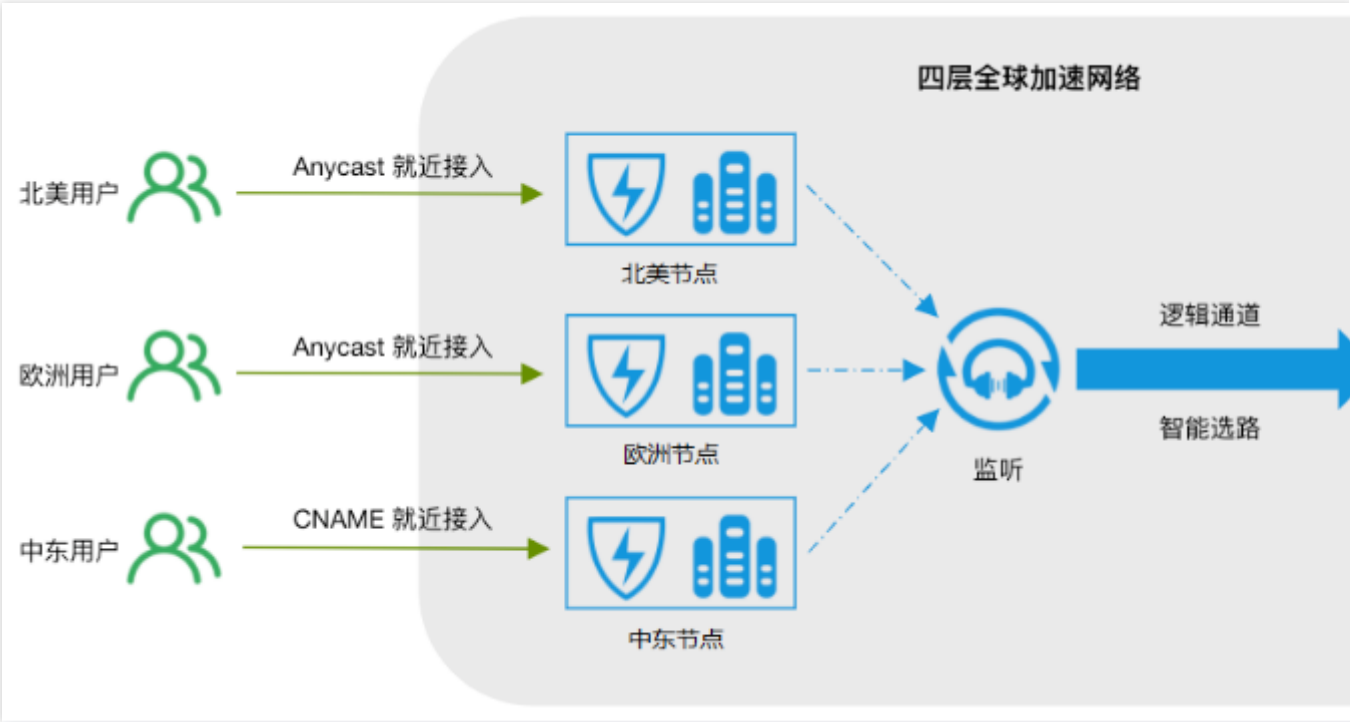
文档目录

四层代理
概述

最近更新时间：2024-06-19 17:31:21

原理介绍

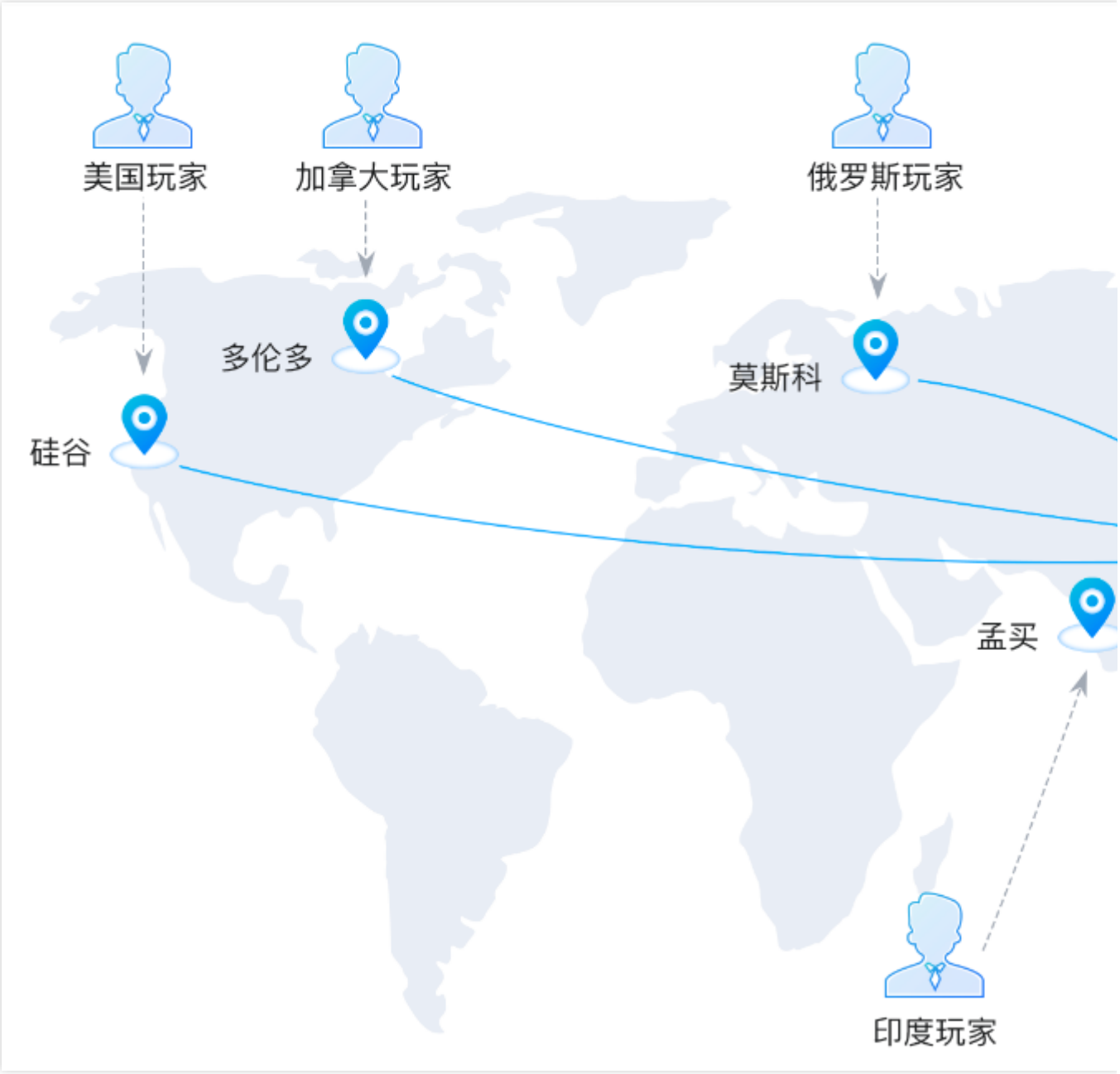
四层代理是 EdgeOne 提供的基于 TCP/UDP 协议加速服务，通过 EdgeOne 分布广泛的四层代理节点、独有的 DDoS 防护模块和智能路由技术，实现终端用户就近接入、边缘流量清洗和端口监听转发，为四层应用提供高可用低延迟的 DDoS 防护和四层加速服务。



应用场景

游戏加速

四层代理可为实时对战类游戏、全球同服等手游、端游、游戏平台提供基于 TCP/UDP 的传输加速，针对游戏场景内各区域网络差异而导致游戏延迟高、丢包等问题，玩家可通过四层代理实现就近接入高速通道，降低游戏的丢包率和延时。



办公应用加速

对跨区域的办公场景，通常公司内业务数据存储于总部数据中心内，往往会因为跨区域产生的网络问题导致高延时和高丢包，影响跨区域的业务访问和数据同步。通过四层代理，可以让用户就近接入加速节点，优化访问链接，有效解决跨区域访问带来的网络质量问题，提升业务访问体验。

实时音视频

在视频会议、连麦互动的场景下，可通过四层代理的 UDP 转发加速，帮助在实时音视频互动的场景下，保障音视频传输的可靠性，解决跨运营商、长路径以及跨国场景下的音视频卡顿、丢包、延时高等问题。

配额说明

四层代理服务默认提供1个 CNAME 类型接入的实例配额，如需通过 Anycast IP 接入或添加更多 CNAME 类型实例，您可以通过在四层代理页面单击**调整配额**进行购买，实例价格详见：四层代理实例。

新建四层代理实例

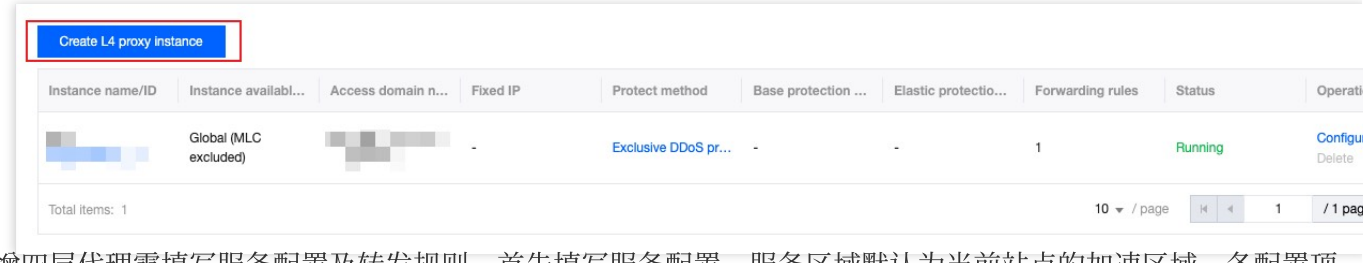
最近更新时间：2024-08-01 21:32:16

使用场景

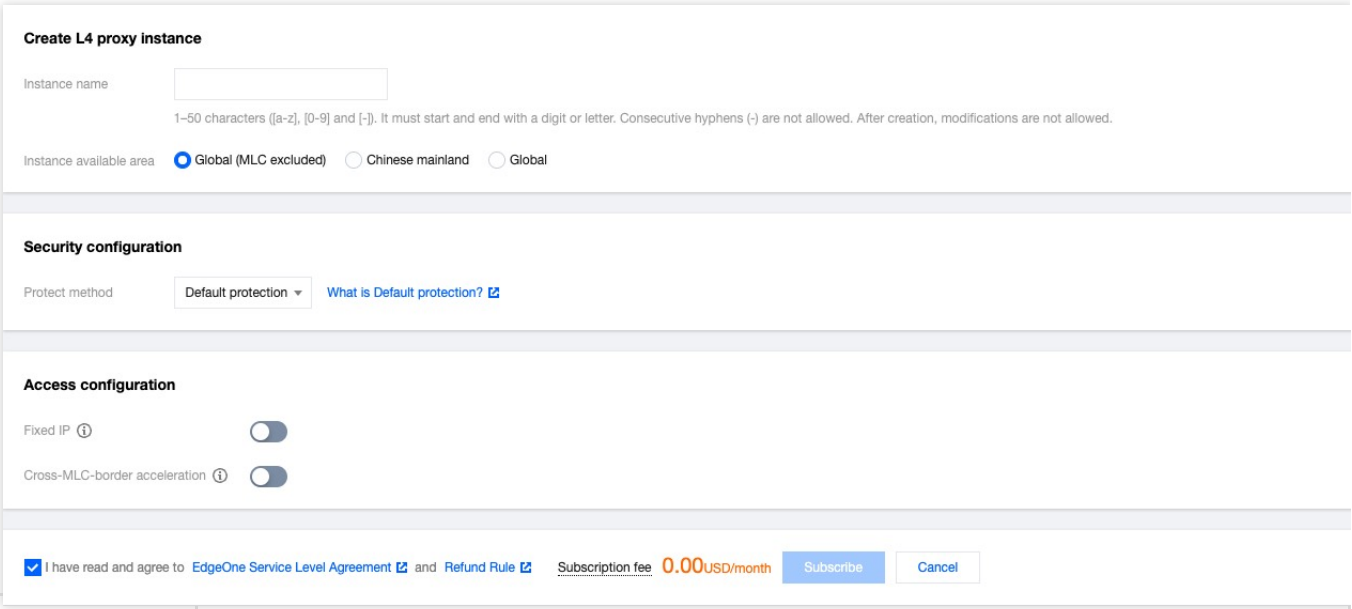
用户需新建一个四层代理实例时，可参考本文档查看如何进行实例配置。

操作步骤

1. 登录 边缘安全加速平台 EO 控制台，在左侧菜单栏中，单击**站点列表**，在站点列表内单击需配置的**站点**。
2. 在站点详情页面，单击**四层代理**。
3. 在四层代理页面，单击**新增四层代理实例**。



4. 新增四层代理需填写服务配置及转发规则，首先填写服务配置，服务区域默认为当前站点的加速区域，各配置项说明如下：



配置项	说明
实例名称	可输入1-200个字符，允许的字符为a-z，A-Z，0-9，_和-。
安全防护配置	**平台默认防护**：默认启用，详情可参考 DDoS 防护概述。 **独立DDoS防护**：详情可参考 使用独立 DDoS 防护。
固定 IP	开启后，支持用户通过固定 IP 地址访问。
IPv6 访问	开启后，支持用户通过 IPv6 协议访问。
中国大陆网络优化	开启后，将针对中国大陆地区用户优化访问性能，详情可参考 跨地域安全加速（海外站点）。

注意：

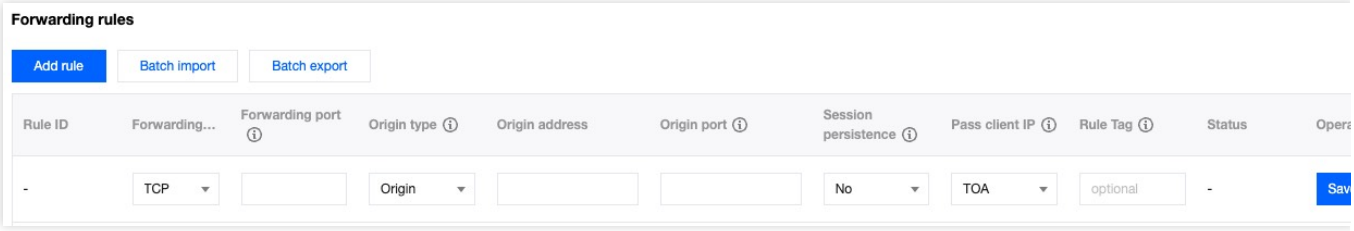
固定 IP、IPv6 访问、中国大陆网络优化不可同时开启，且在不同的加速区域，安全防护配置与接入配置之间存在冲突关系，冲突如下：

安全防护配置	功能	全球可用区（不含中国大陆）	中国大陆可用区	全球可用区
平台默认防护	固定 IP	✓	×	×
	IPv6 访问	✓	✓	✓
	中国大陆网络优化	✓	×	×
独立 DDoS 防护	固定 IP	✓	×	×
	IPv6 访问	✓	×	×

	中国大陆网络优化	✓	×	×

5. 查看订阅费用，勾选并同意下方的 边缘安全加速平台服务协议 和 退款规则，单击订阅。计费说明可参考计费说明。

6. 配置转发规则，在四层代理页面，选择刚才新建的四层代理实例，单击**配置**，进入实例详情页面配置转发规则，转发规则也支持批量导入规则，详情见：批量配置转发规则；转发规则的各配置项说明如下：



注意：

1. 如果源站类型为源站组，目前仅允许配置为自有源站，不支持 COS 源。
2. 每个四层代理实例支持最多配置2000条转发规则。

配置项	说明
规则 ID	后台自动生成，不支持修改，规则唯一标识 ID。
转发协议	对应的四层代理转发协议，支持选择 TCP 或 UDP。
转发端口	支持端口范围1-64999，支持输入多个端口，用分号隔开，支持连字符输入端口段，例如：80-90。一个转发规则最多可输入20个端口。 以下端口为内部保留端口，请不要选择： 转发协议为 TCP 时：3943、3944、6088、36000、56000。 转发协议为 UDP 时：4789、4790、6080、61708。
源站类型&源站地址	**单一源站**：支持输入单个源站 IP 地址或者域名。 **源站组**：从已有的源站组中选择源站，也可以在此新建源站组。
源站端口	可填写单一端口或端口段，如果是端口段，转发端口必须也为端口段，且源站端口与转发端口的端口段长度一致。 例如：转发端口段为80-90，则源站端口段可为80-90，或90-100。
会话保持	源站 IP 不变的情况下，同一个客户端 IP 始终回到同一个源站 IP。
传递客户端 IP	TOA：通过 TCP Option (type 200) 传递客户端 IP。支持 TCP 协议，不支持 UDP 协议。 Proxy Protocol V1（推荐）：Proxy Protocol V1 协议通过 TCP Header 传递客户端 IP，V1版本采用明文传递。支持 TCP 协议，不支持 UDP 协议。

	Proxy Protocol V2：通过 Header 传递客户端 IP，V2版本采用二进制格式，支持 TCP/UDP 协议。TCP 每个连接的第一个数据包都会携带 PPv2 头部，UDP 只有数据流的第一个报文会携带。 不传递：配置不传递真实客户端 IP。
规则标签	选填，可输入1-50个任意字符，对转发规则进行标识。

7. 单击**保存**，即可完成四层代理的规则配置。

修改四层代理实例配置

最近更新时间：2023-10-11 10:46:20

使用场景

用户需对已有四层代理实例修改配置时，可参考本文档查看如何修改四层代理实例配置。

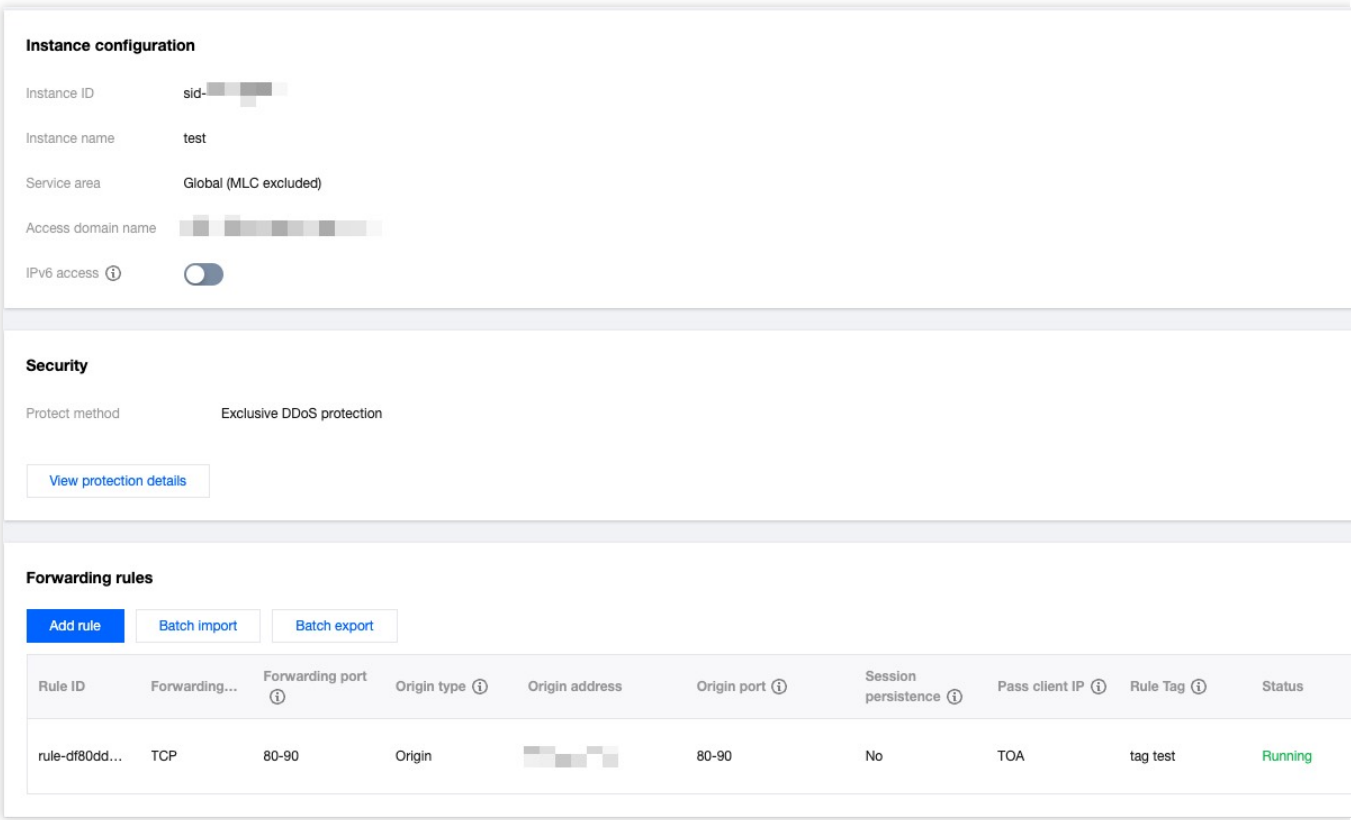
注意：

已创建的四层代理实例不支持修改调度模式和代理模式，如需修改，请删除该实例后重新创建。

如需删除转发规则，需暂停该规则后方可删除。

操作步骤

1. 登录 边缘安全加速平台 EO 控制台，在左侧菜单栏中，单击**站点列表**，在站点列表内单击需配置的**站点**。
2. 在站点详情页面，单击**四层代理**。
3. 在四层代理页面，选择需要修改的四层代理规则，单击**配置**。
4. 对已创建的四层代理模式，支持修改 IPv6 访问配置及中国大陆网络优化功能，也可以在该页面添加、编辑、暂停/开启、删除转发规则。


Instance configuration
Instance ID
sid-
Instance name
test
Service area
Global (MLC excluded)
Access domain name
IPv6 access
Security
Protect method
Exclusive DDoS protection
View protection details
Forwarding rules
Add rule
Batch import
Batch export
Rule ID
Forwarding...
Forwarding port
Origin type
Origin address
Origin port
Session persistence
Pass client IP
Rule Tag
Status
rule-df80dd...
TCP
80-90
Origin
80-90
No
TOA
tag test
Running

停用/删除四层代理实例

最近更新时间：2023-10-11 10:46:43

使用场景

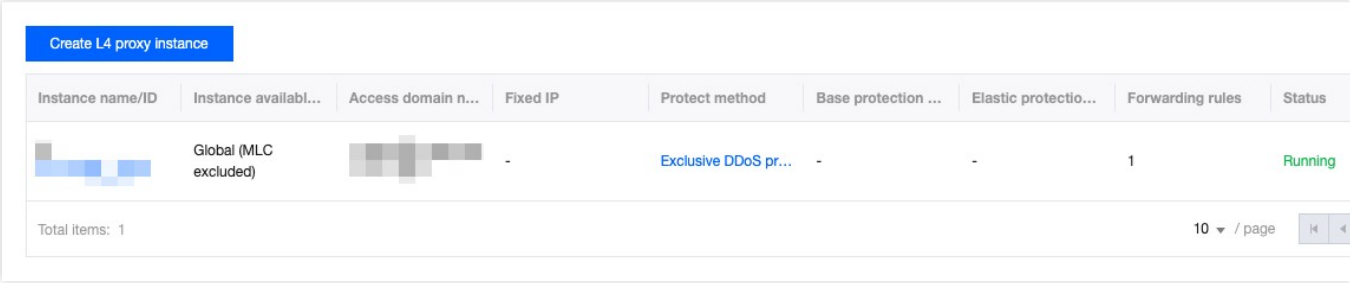
用户需停用当前四层代理实例或删除四层代理实例时，可参考本文进行操作。

说明：

停用四层代理实例需要等待一段时间，一般需要几分钟时间即可；实例停用后，方可删除。

操作步骤

1. 登录 边缘安全加速平台 EO控制台，在左侧菜单栏中，单击**站点列表**，在站点列表内单击需配置的**站点**。
2. 在站点详情页面，单击**四层代理**。
3. 在四层代理页面，选择需要停用的四层代理实例，单击**停用**。



4. 停用后，如需删除该实例，可单击**删除**，删除该实例配置。

批量配置转发规则

最近更新时间：2023-10-11 10:47:05

使用场景

如果您的四层代理实例中有大量的转发规则需要维护，可参考本文来了解如何通过导入/导出功能来帮助您批量配置转发规则。

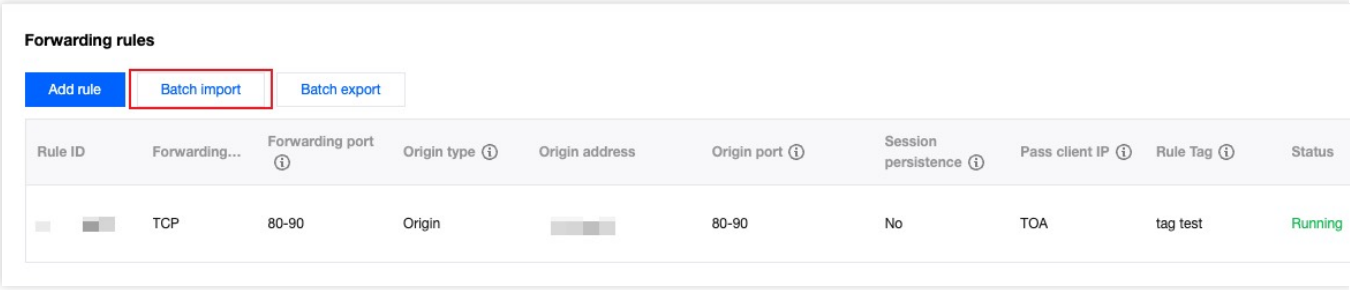
注意：

1. 批量导入单次最多可输入2000条，每个四层代理实例最多支持2000条实例。
2. 批量导入规则不区分大小写。
3. 导入规则的转发端口不可与现有规则的转发端口重复。

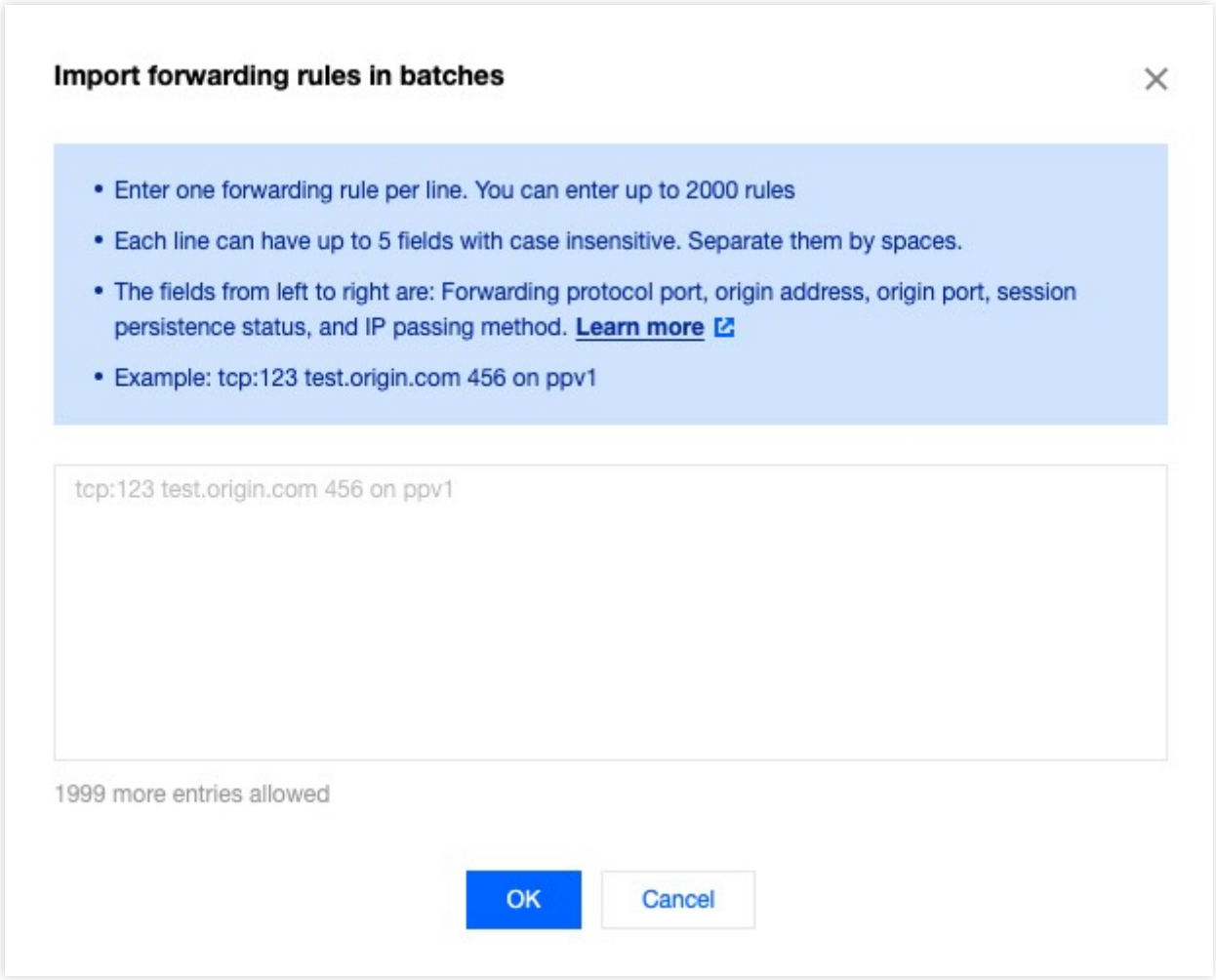
操作步骤

批量导入规则

1. 登录 边缘安全加速平台 EO控制台，在左侧菜单栏中，单击**站点列表**，在站点列表内单击需配置的**站点**。
2. 在站点详情页面，单击**四层代理**。
3. 在四层代理页面，选择需要修改的四层代理规则，单击配置。
4. 在转发规则页面，单击**批量导入**。



5. 输入需要导入的转发规则，一行对应一条转发规则，需包含转发协议：端口、源站地址、源站端口、会话保持状态、传递 IP 方式，各字段以空格间隔。例如：`tcp:123 test.origin.com 456 on ppv1` 。



各字段说明如下：

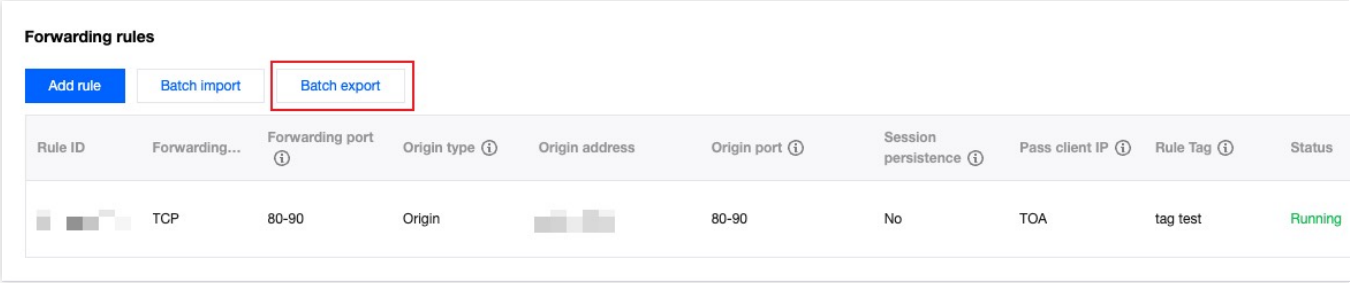
字段	说明
转发协议:端口	转发协议可选 TCP/UDP。 支持端口范围1-64999，支持输入多个端口，用分号隔开，支持连字符输入端口段，例如：80-90。一个转发规则最多可输入20个端口。 以下端口为内部保留端口，请不要选择： 转发协议为 TCP 时：3943、3944、6088、36000、56000。 转发协议为 UDP 时：4789、4790、6080、61708。
源站地址	支持输入单一源站 IP 地址或者域名。 支持输入源站组名称，源站组名称格式为：`og:{OriginGroupName}`。例如：`og:testorigin`。
源站端口	支持单一端口或端口段，如果是端口段，转发端口必须也为端口段，且源站端口与转发端口的端口段长度一致。 例如：转发端口段为80-90，则源站端口段可为80-90，或90-100。
会话保持状态	可选 ON/OFF

传递客户端 IP	可选 TOA/PPv1/ppv2/OFF。
规则标签	选填，可输入1-50 个任意字符，对该条转发规则进行标识。

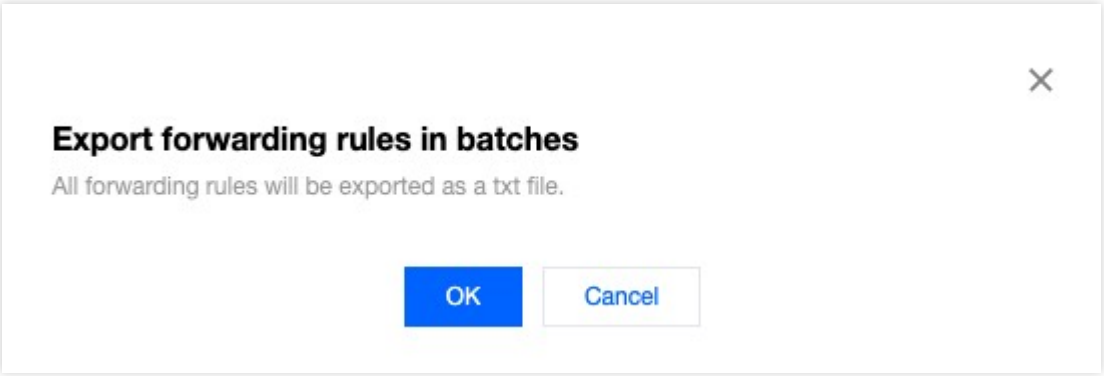
6. 单击**确定**，即可导入转发规则。

批量导出规则

1. 登录 边缘安全加速平台控制台，在左侧菜单栏中，单击**站点列表**，在站点列表内单击需配置的**站点**。
2. 在站点详情页面，单击**四层代理**。
3. 在四层代理页面，选择需要修改的四层代理规则，单击配置。
4. 在转发规则页面，单击**批量导出**。



5. 在弹出的对话窗口中，单击**确定**，即可导出所有转发规则。导出的转发规则格式为 TXT 文件，内容格式与导入规则一致。



获取客户端真实IP
通过 TOA 获取 TCP 协议客户端真实 IP

最近更新时间：2023-09-11 17:40:34

本文介绍了使用四层代理加速时，如何通过 TOA 获取 TCP 协议的客户端真实 IP。

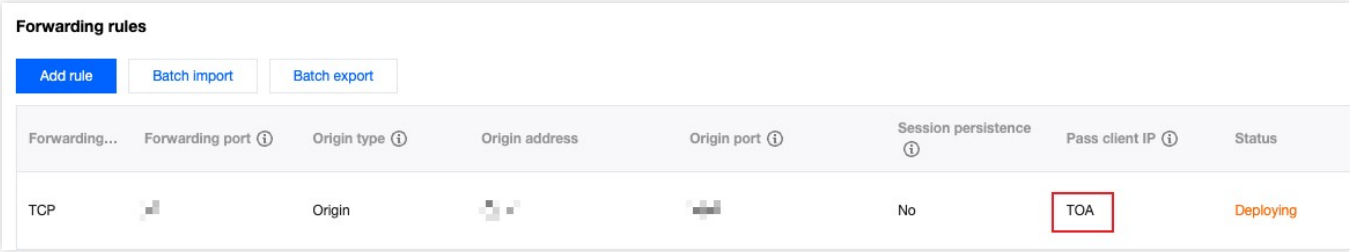
使用场景

当数据报文通过四层加速通道进行加速时，数据报文的源 IP 地址和源 Port 均会发生修改，导致源站无法直接获取到真实客户端的 IP 和 Port 信息。为了将客户端真实 IP 和 Port 信息可传递给源站服务器，在创建加速通道时，您可选择通过 TOA 来传递客户端 IP 和 Port 信息。四层加速通道会将真实客户端的 IP 和 Port 信息放入自定义的 tcp option 字段中。您需要在源站服务器上通过安装 TOA 模块来获取真实客户端地址信息。

操作步骤

步骤一：传递客户端 IP 方式选择为 TOA

使用 TOA 获取 TCP 协议客户端真实 IP，需在控制台内将四层代理转发规则的传递客户端 IP 方式配置为 TOA，如何修改四层代理规则详见：修改四层代理实例配置。



步骤二：后端服务加载 TOA 模块

您可以通过以下两种方式加载 TOA 模块：
方法一（推荐）：根据源站 Linux 版本，下载对应版本已编译好的 toa.ko 文件直接进行加载。
方法二：如果方法一中没有找到您当前的源站 Linux 版本，您可以通过下载 TOA 源码文件自行编译并加载。
注意：
因不同安装环境的差异，如果您使用方法一加载过程中遇到问题，请尝试使用方法二，自行安装编译环境后加载。
方法一：下载已编译的 TOA 模块并加载

方法二：自行编译并加载 TOA 模块

1. 根据腾讯云上 Linux 的版本，下载对应的 TOA 包并解压。

centos

CentOS-7.2-x86_64.tar.gz

CentOS-7.3-x86_64.tar.gz

CentOS-7.4-x86_64.tar.gz

CentOS-7.5-x86_64.tar.gz

CentOS-7.6-x86_64.tar.gz

CentOS-7.7-x86_64.tar.gz

CentOS-7.8-x86_64.tar.gz

CentOS-7.9-x86_64.tar.gz

CentOS-8.0-x86_64.tar.gz

CentOS-8.2-x86_64.tar.gz

debian

Debian-11.1-x86_64.tar.gz

Debian-10.2-x86_64.tar.gz

Debian-9.0-x86_64.tar.gz

suse linux

openSUSE-Leap-15.3-x86_64.tar.gz

ubuntu

Ubuntu-14.04.1-LTS-x86_64.tar.gz

Ubuntu-16.04.1-LTS-x86_64.tar.gz

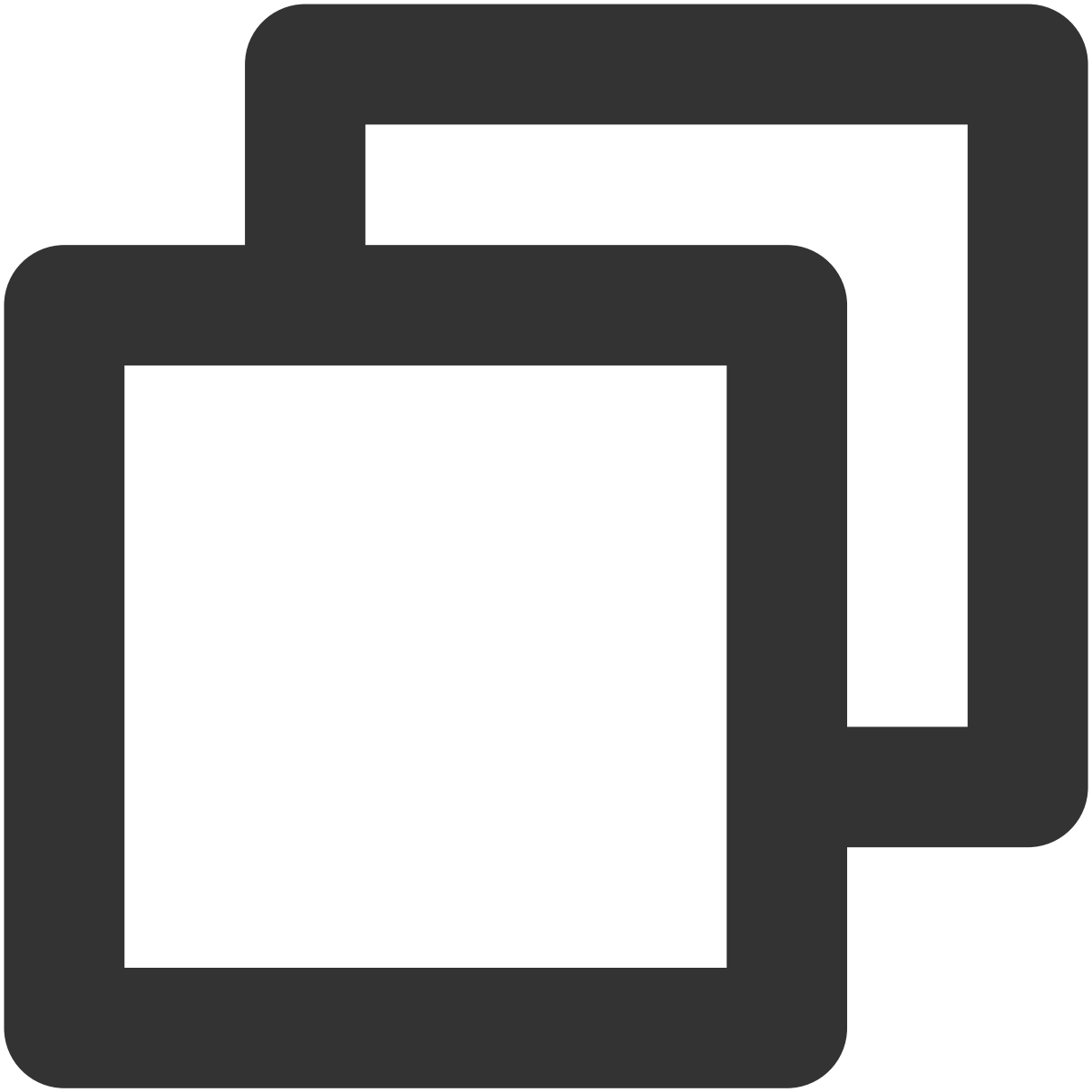
Ubuntu-18.04.1-LTS-x86_64.tar.gz

Ubuntu-20.04.1-LTS-x86_64.tar.gz

2. 解压完成后，执行 cd 命令进入刚解压的文件夹后，按照以下方法执行加载 TOA 模块：

脚本一键执行

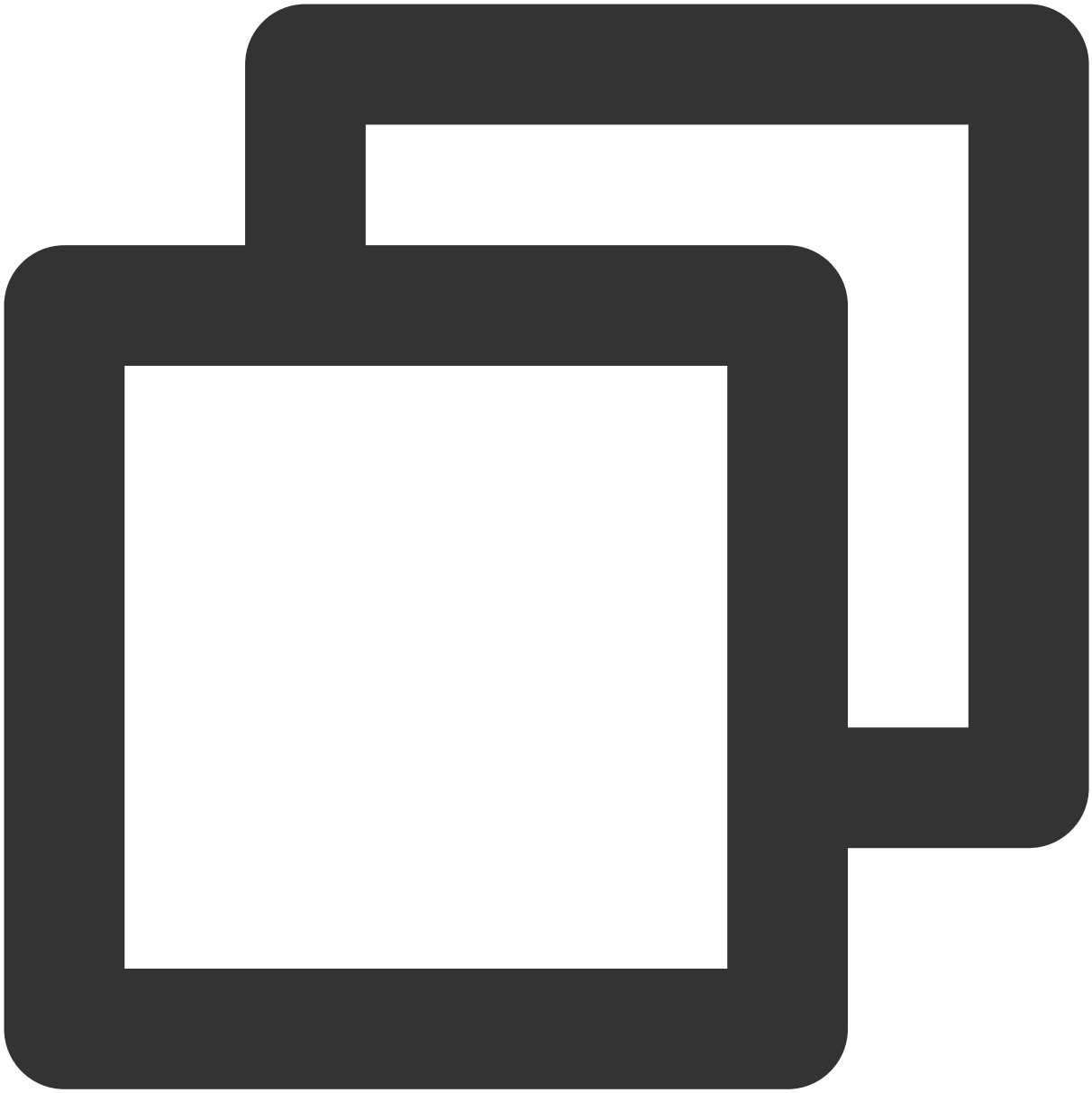
手工配置加载



```
/bin/bash -c "$(curl -fsSL https://edgeone-document-file-1258344699.cos.ap-guangzho
```

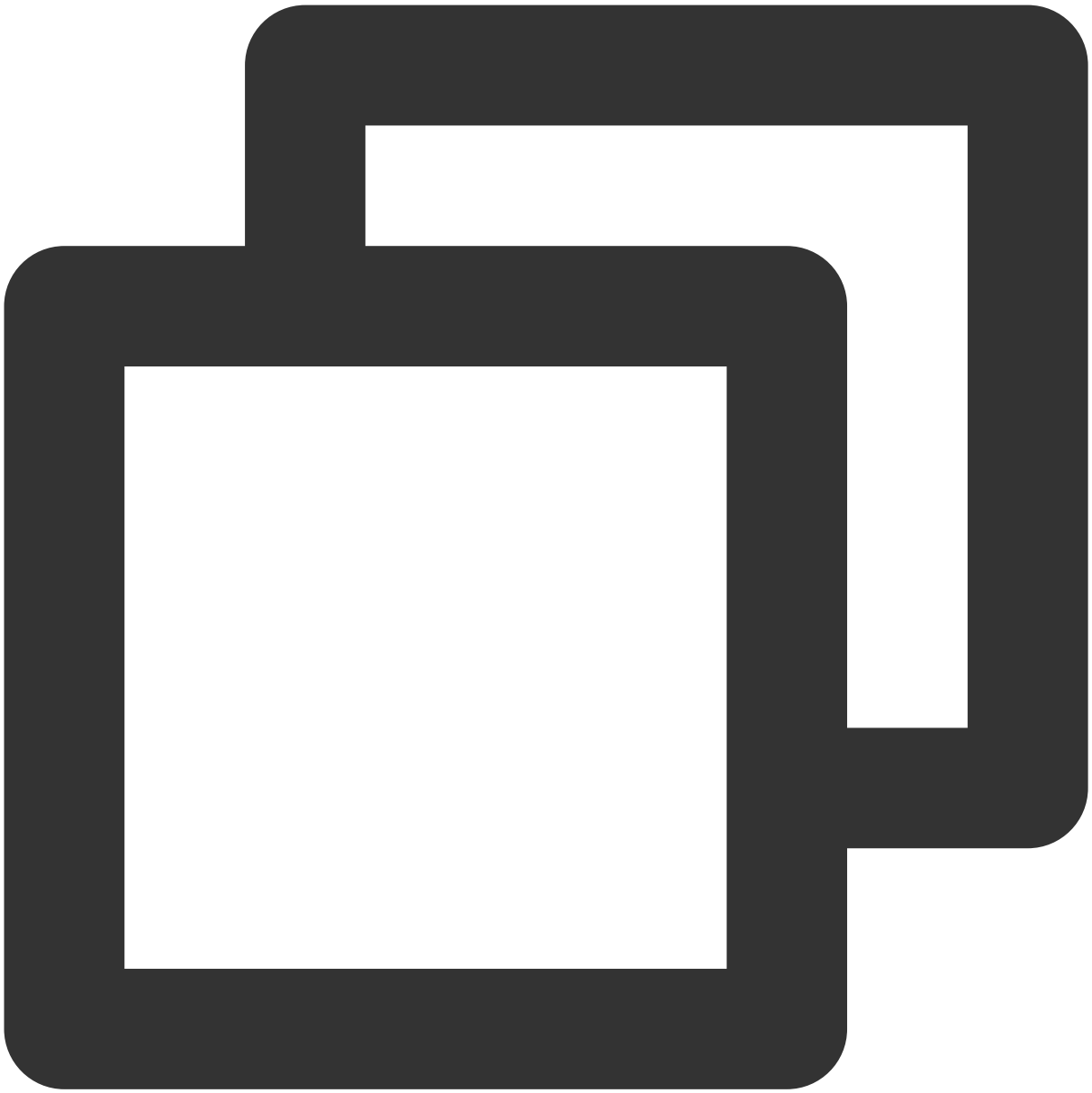
加载成功后显示如下：

```
[root@VM-0-14-centos toa]# /bin/bash -c "$(curl -fsSL https://eo-toa-1258348367.cos.ap-shanghai.myqcloud.com/
toa.ko install successfully
[root@VM-0-14-centos toa]#
```



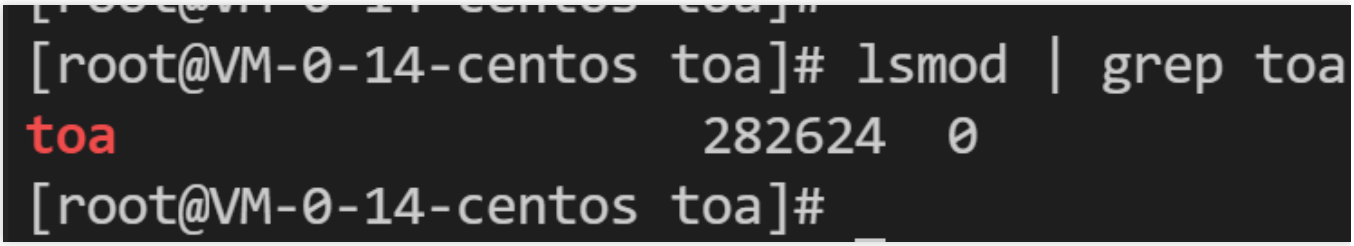
```
# 解压tar包
tar -zxvf CentOS-7.2-x86_64.tar.gz
# 进入解压后的包目录
cd CentOS-7.2-x86_64
# 加载toa模块
insmod toa.ko
# 拷贝到内核模块目录下
cp toa.ko /lib/modules/`uname -r`/kernel/net/netfilter/ipvs/toa.ko
# 设置系统启动时自动加载toa模块
echo "insmod /lib/modules/`uname -r`/kernel/net/netfilter/ipvs/toa.ko" >> /etc/rc.l
```

可通过下面命令确认是否已加载成功：



```
lsmod | grep toa
```

出现 TOA 时表示已加载成功，如下图所示：



1. 安装编译环境。

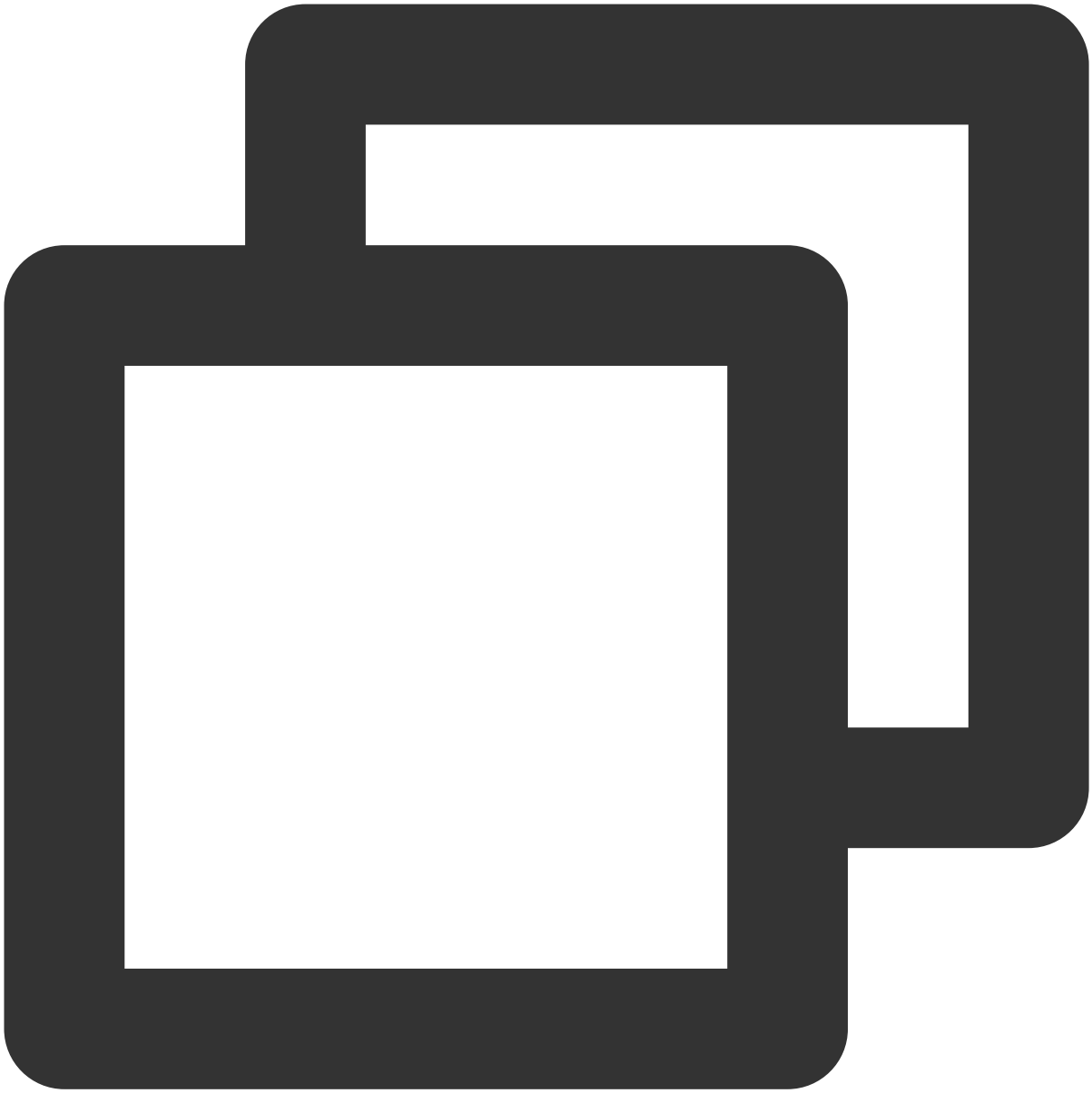
1.1 查看当前内核版本号，确认 kernel-devel ， kernel-headers 已安装，并保证版本号与内核版本保持一致。

1.2 确认已安装 gcc 和 make。

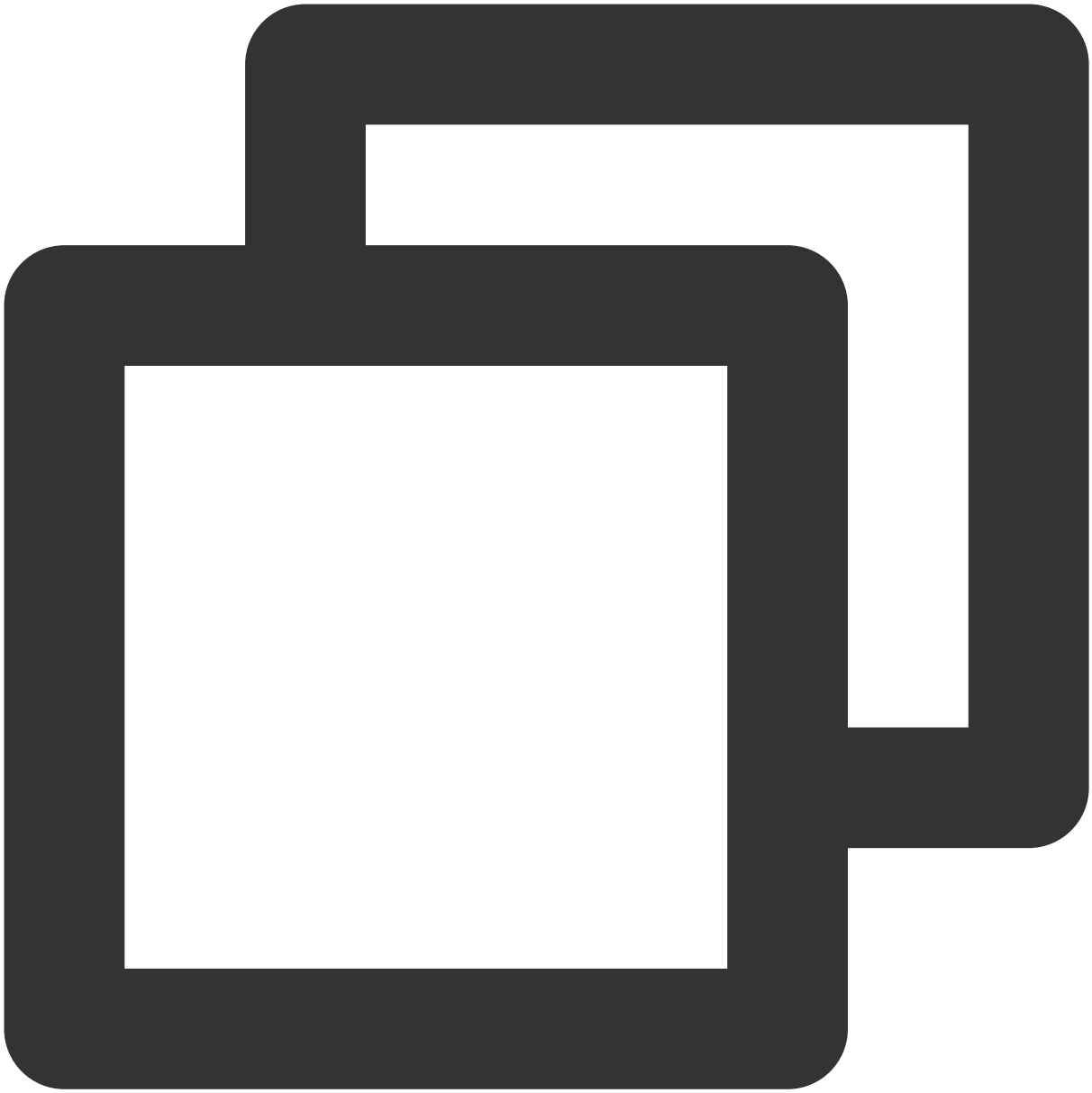
1.3 如果以上环境依赖没有安装，可参考如下命令进行安装：

Centos

Ubuntu/Debian



```
yum install -y gcc
yum install -y make
yum install -y kernel-headers kernel-devel
```

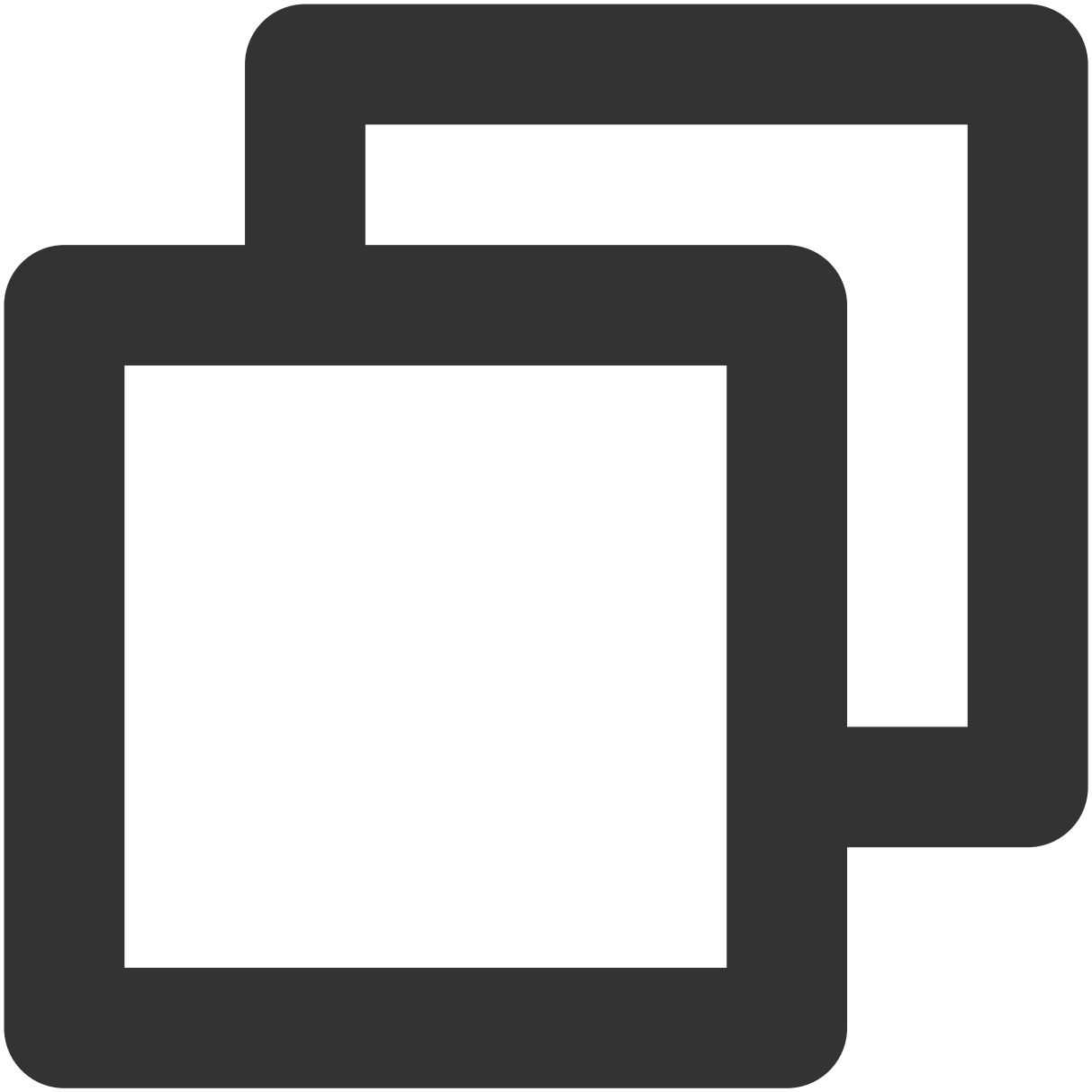


```
apt-get install -y gcc
apt-get install -y make
apt-get install -y linux-headers-$(uname -r)
```

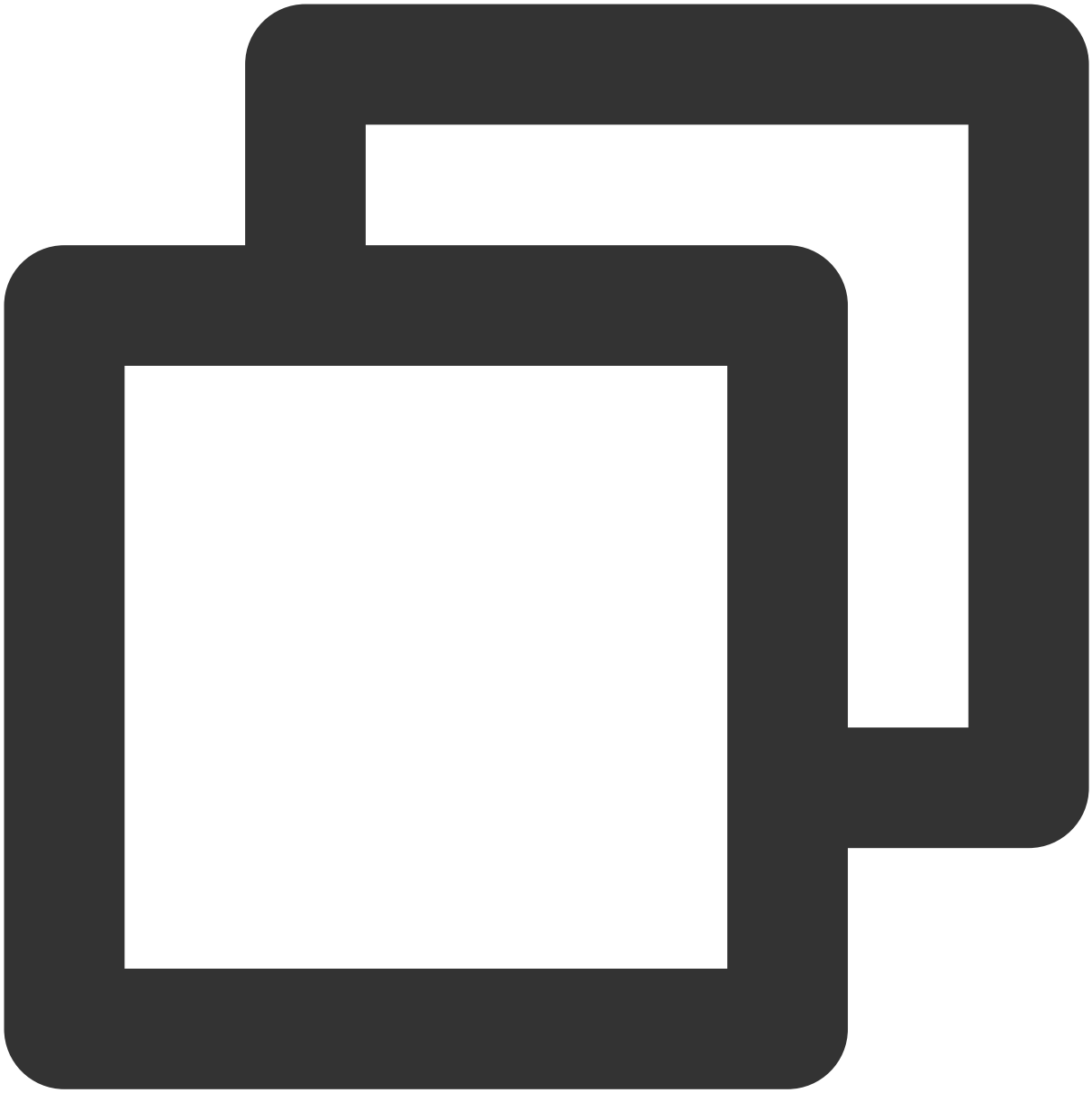
2. 安装完编译环境后，执行以下命令完成源码下载，编译和加载。

脚本一键编译并加载

手工编译并加载



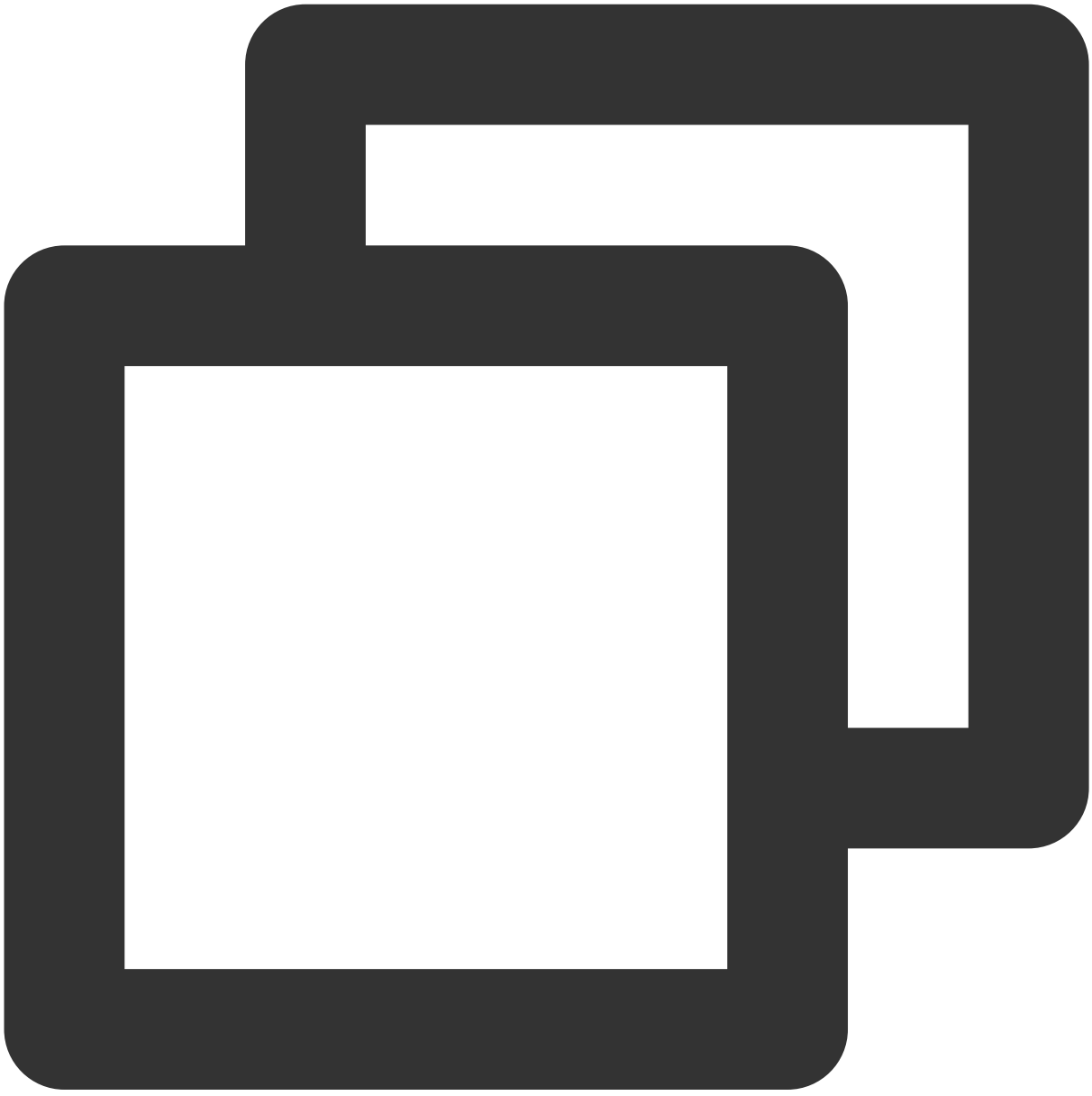
```
/bin/bash -c "$(curl -fsSL https://edgeone-document-file-1258344699.cos.ap-guangzho
```



```
# 创建并进入编译目录
mkdir toa_compile && cd toa_compile
# 下载源代码tar包
curl -o toa.tar.gz https://edgeone-document-file-1258344699.cos.ap-guangzhou.myqclo
# 解压tar包
tar -zxvf toa.tar.gz
# 编译toa.ko文件，编译成功后会在当前目录下生成toa.ko文件
make
# 加载toa模块
insmod toa.ko
# 拷贝到内核模块目录下
```

```
cp toa.ko /lib/modules/`uname -r`/kernel/net/netfilter/ipvs/toa.ko
# 设置系统启动时自动加载toa模块
echo "insmod /lib/modules/`uname -r`/kernel/net/netfilter/ipvs/toa.ko" >> /etc/rc.l
```

3. 执行下面指令确认是否已加载成功：



```
lsmod | grep toa
```

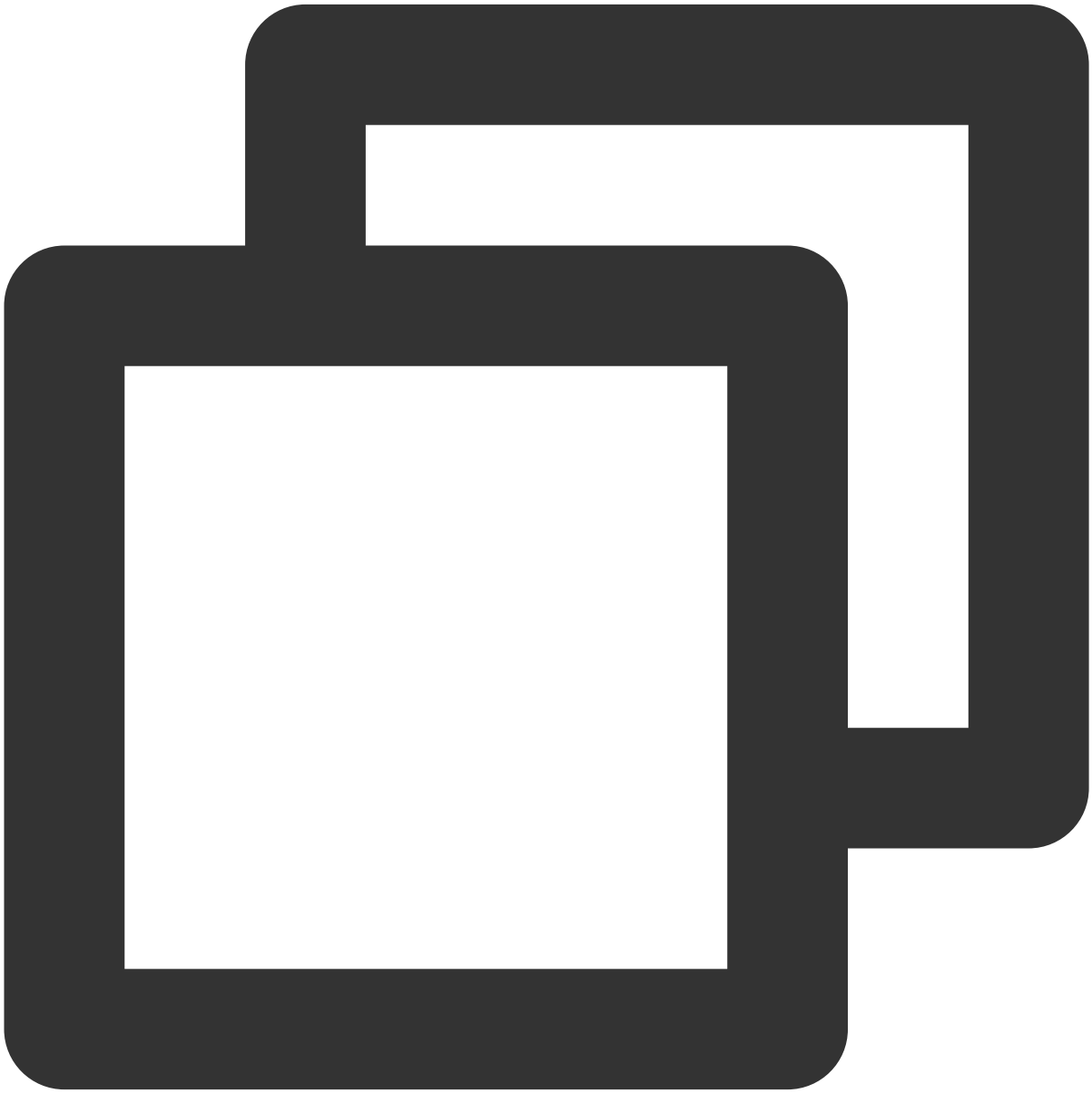
出现 toa 则表示已加载成功，如下图所示：

```
[root@VM-16-42-centos ~]# lsmod | grep toa
toa                    278528  0
```

步骤三：验证获取客户端 IP 信息

您可以通过搭建 TCP 服务，并通过另外一台服务器模拟客户端请求进行验证，示例如下：

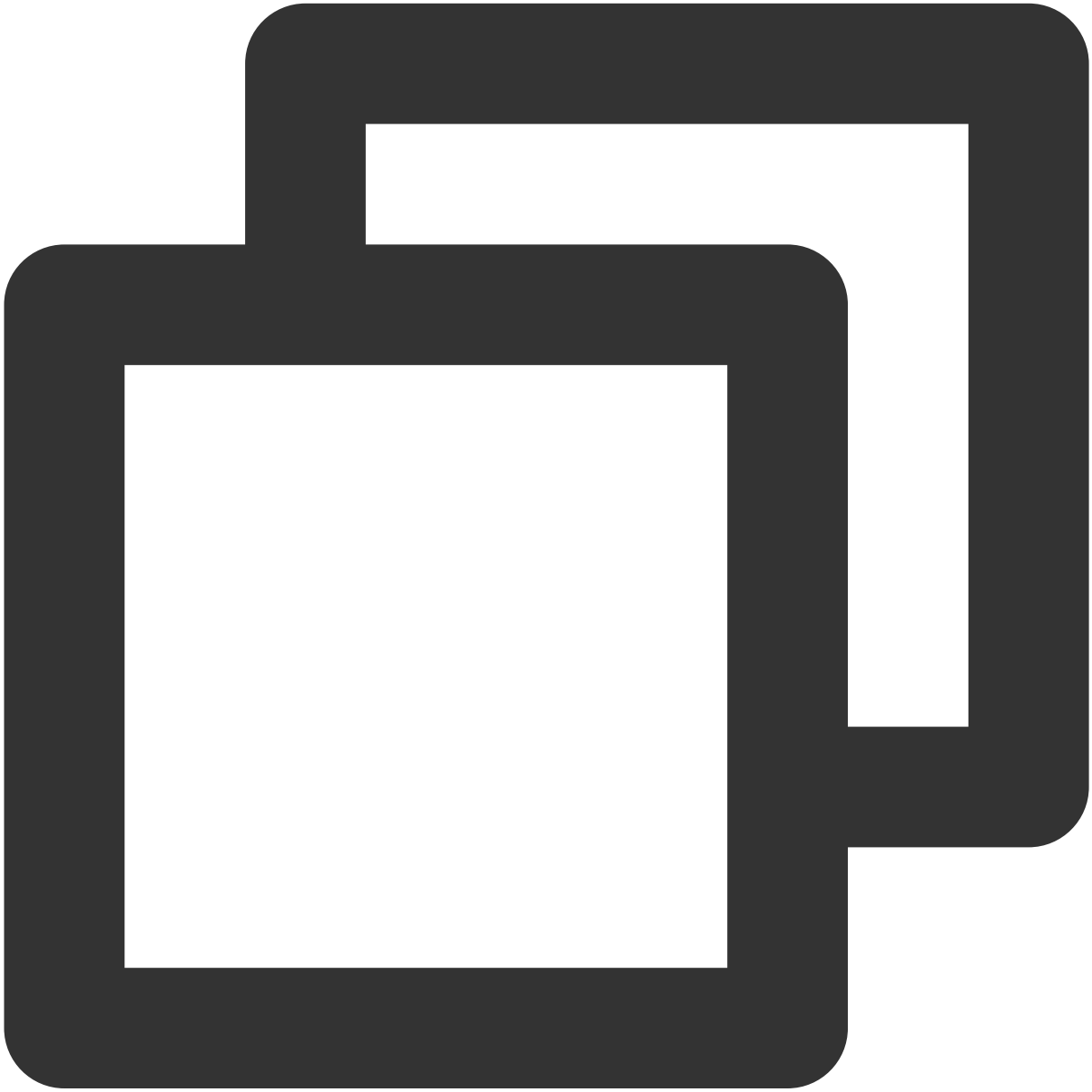
1. 在当前服务器上，可以通过 Python 创建一个 HTTP 服务来模拟 TCP 服务，如下所示：



```
# 基于python2
```

```
python2 -m SimpleHTTPServer 10000

# 基于python3
python3 -m http.server 10000
```

2. 用另一台服务器充当客户端，构造客户端请求，以 Curl 请求来模拟 TCP 请求：



```
# 利用curl发起http请求，其中域名为四层代理域名，10000为四层代理转发端口
curl -i "http://a8b7f59fc8d7e6c9.example.com.edgeonedy1.com:10000/"
```

3. 如果 TOA 已加载完成，在已加载 TOA 的服务器会看到客户端的真实地址信息，如下图红框所示：

如果您当前的业务是以下两种场景，只需要获取 IPv4 或 IPv6 其中一种类型客户端地址，那么参照上述步骤完成服务端加载 TOA 模块即可获取到客户端真实 IP 地址。

源站是 IPv4，只需要获取 IPV4 客户端地址。

源站是 IPv6，只需要获取 IPV6 客户端地址。

但是，如果您当前的业务源站需要同时获取到 IPv4 和 IPv6 两种类型客户端地址，则需要在加载 TOA 模块的同时修改源站业务代码，请继续参考如下指引：修改源站业务代码，支持同时获取 IPv4/IPv6 客户端真实地址信息。

修改源站业务代码，同时获取IPv4/IPv6客户端真实 IP

说明：

本章节操作仅在源站需同时获取 IPv4 和 IPv6 客户端地址信息时参考，该操作将指引您如何修改源站业务代码。

源站在建立服务监听时，可参考采用如下两种方式：

1. 采用 IPv4 的地址结构（ `struct sockaddr_in` ）搭建服务，其监听的是 IPv4 格式的地址。
2. 采用 IPv6 的地址结构（ `struct sockaddr_in6` ）搭建服务，其监听的是 IPv6 格式的地址。

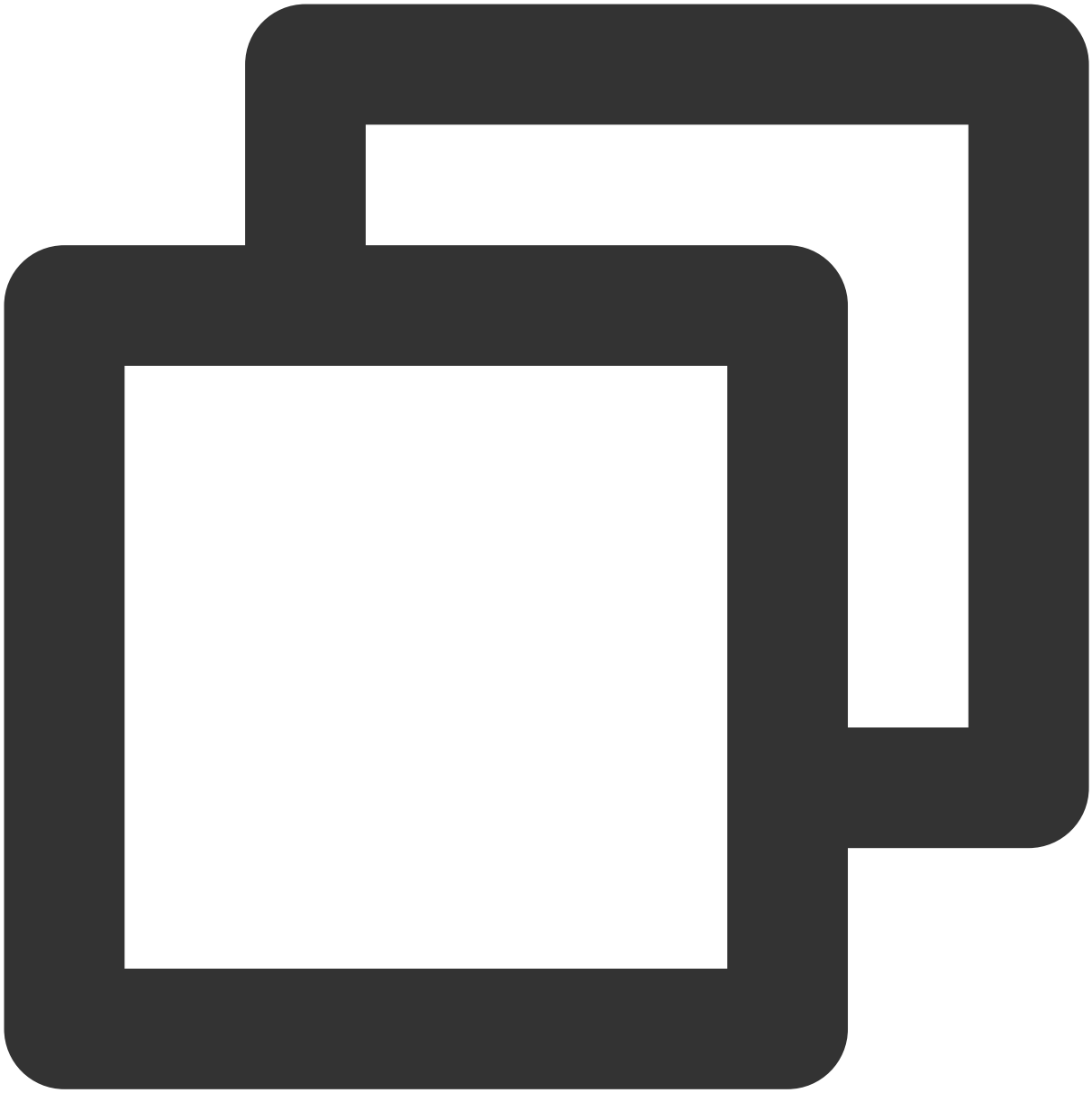
示例代码

监听 IPv4 地址

监听 IPv6 地址

C

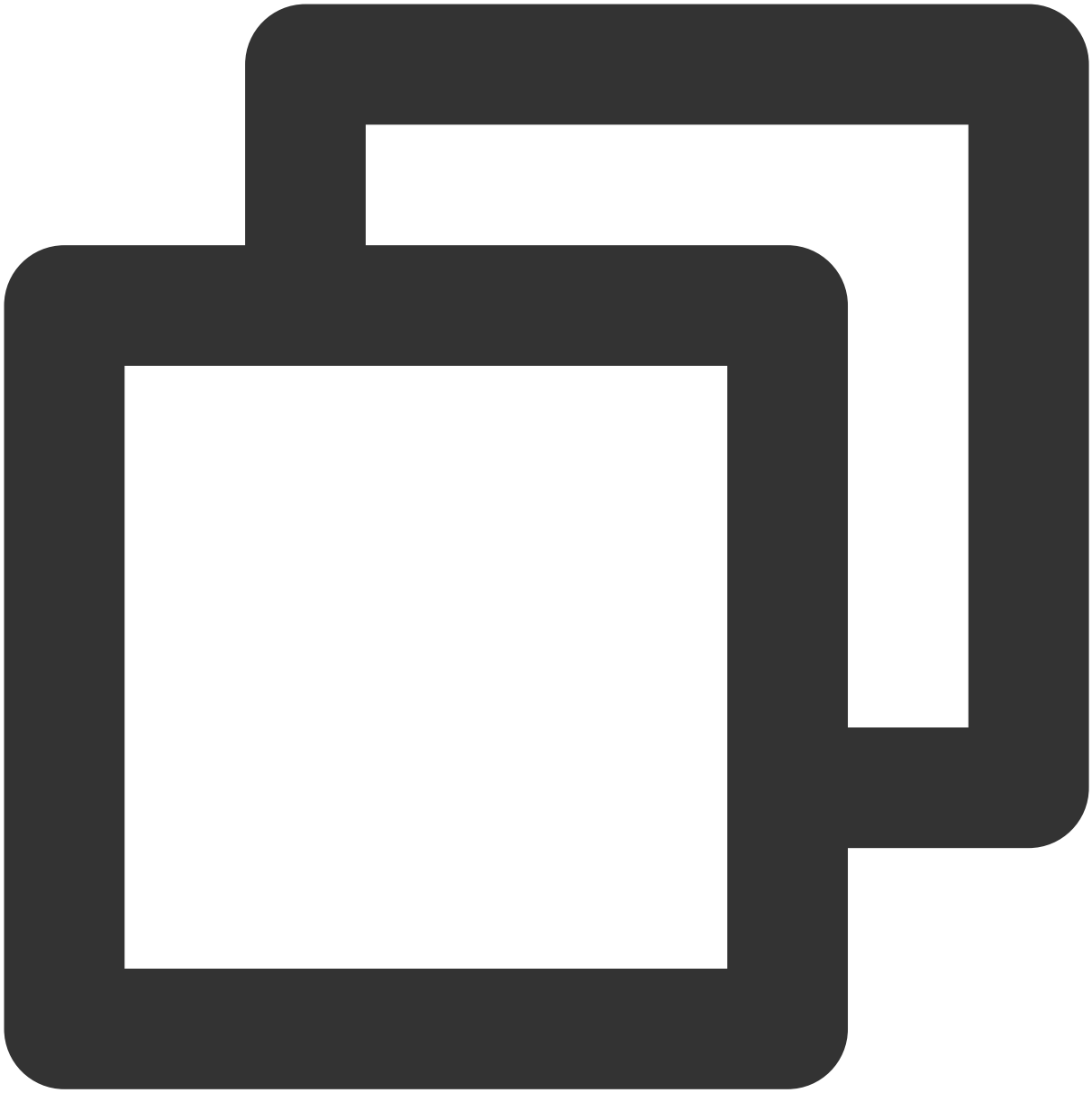
Java



```
#include <sys/socket.h>
#include <stdio.h>
#include <unistd.h>
#include <netinet/in.h>
#include <memory.h>
#include <arpa/inet.h>

int main(int argc,  char **argv){
        int     l_sockfd;
    // 服务器地址采用v4结构
        struct sockaddr_in serveraddr;
```

```
// 业务修改点：客户端地址必须采用v6结构
struct sockaddr_in6 clientAddr;
    int server_port = 10000;

    memset(&serveraddr, 0, sizeof(serveraddr));
// 创建socket
    l_sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (l_sockfd == -1){
            printf("Failed to create socket.\\n");
            return -1;
    }

// 初始化服务器地址信息
    memset(&serveraddr, 0, sizeof(struct sockaddr_in));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(server_port);
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);

    int isReuse = 1;
    setsockopt(l_sockfd, SOL_SOCKET,SO_REUSEADDR,(const char*)&isReuse,sizeof(i

// 关联socket和服务器地址信息
    int nRet = bind(l_sockfd,(struct sockaddr*)&serveraddr, sizeof(serveraddr))
    if(-1 == nRet)
    {
            printf("bind error\\n");
            return -1;
    }
// 监听socket
    listen(l_sockfd, 5);

int clientAddrLen = sizeof(clientAddr);
memset(&clientAddr, 0, sizeof(clientAddr));
// 接受来自客户端的连接
int linkFd = accept(l_sockfd, (struct sockaddr*)&clientAddr, &clientAddrLen);
if(-1 == linkFd)
{
    printf("accept error\\n");
    return -1;
}
// 业务修改点：根据客户端sin6_family的类型，判断客户端是v4地址还是v6地址
//    当为AF_INET时，表示客户端是IPv4，将客户端地址指针转换为struct sockaddr_in*进行获取
//    当为AF_INET6时，表示客户端是IPv6，使用struct sockaddr_in6*进行获取
if (clientAddr.sin6_family == AF_INET) {
    printf("AF_INET accept getpeername %s : %d successful\\n",
            inet_ntoa(((struct sockaddr_in*)&clientAddr)->sin_addr),
            ntohs(((struct sockaddr_in*)&clientAddr)->sin_port));
```

```
    }else if (clientAddr.sin6_family == AF_INET6){
        char addr_p[128] = {0};
        inet_ntop(AF_INET6, (void *)&((struct sockaddr_in6*)&clientAddr)->sin6_addr
        printf("AF_INET6 accept getpeername %s : %d successful\\n",
                addr_p,
                ntohs(((struct sockaddr_in6*)&clientAddr)->sin6_port));
    }else{
        printf("unknow sin_family:%d \\n", clientAddr.sin6_family);
    }
    close(l_sockfd);
        return 0;
}
```



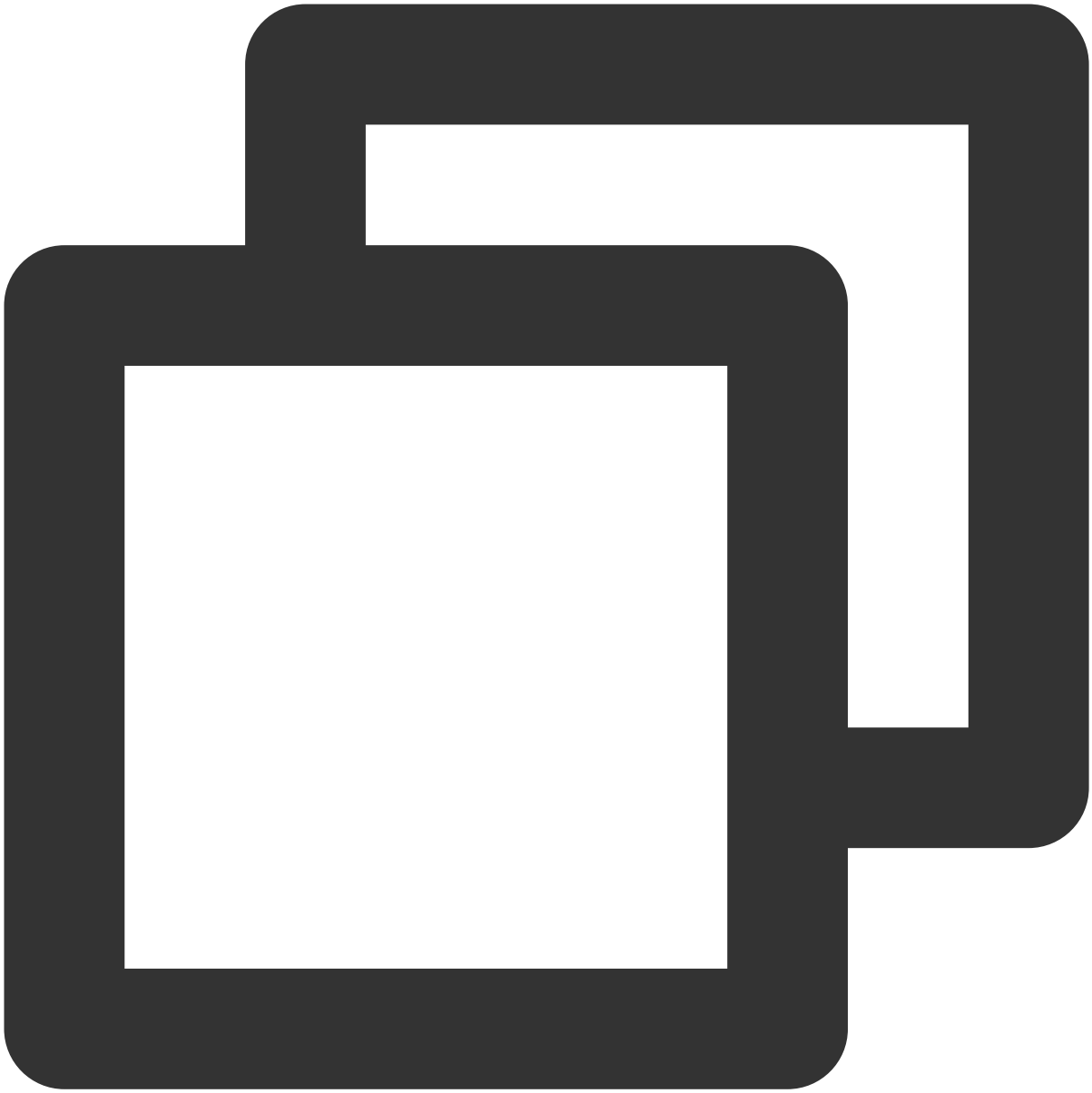
```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketAddress;


public class ServerDemo {
```

```
/** 若采用 IPv4 的地址结构搭建服务，使用 IPV4_HOST */
public static final String IPV4_HOST = "0.0.0.0";


/** 若采用 IPv6 的地址结构搭建服务，使用 IPV6_HOST */
public static final String IPV6_HOST = "::";


public static void main(String[] args) {
    int serverPort = 10000;
    try (ServerSocket serverSocket = new ServerSocket()) {
        // 设置地址复用
        serverSocket.setReuseAddress(true);
        // 绑定服务器地址和端口，这里使用 IPv4
        serverSocket.bind(new InetSocketAddress(InetAddress.getByName(IPV4_HOST
        System.out.println("Server is listening on port " + serverPort);


        while (true) {
            // 接受客户端连接
            Socket clientSocket = serverSocket.accept();
            System.out.println("New client connected: " + clientSocket.getRemot


            // 处理客户端请求
            handleClientRequest(clientSocket);
        }
    } catch (IOException e) {
        System.err.println("Failed to create server socket: " + e.getMessage())
    }
}


/**
 * 处理函数，具体业务具体实现，这里只做为示例
 * 此函数的作用是将 client 的输入原封不动的返回给 client
 */
private static void handleClientRequest(Socket clientSocket) {
    try (InputStream inputStream = clientSocket.getInputStream();
         OutputStream outputStream = clientSocket.getOutputStream()) {


        // 读取客户端发来的数据
        byte[] buffer = new byte[1024];
        int bytesRead;
```

```
            while ((bytesRead = inputStream.read(buffer)) != -1) {
                // 将接收到的数据原样回复给客户端
                outputStream.write(buffer, 0, bytesRead);
            }


        } catch (IOException e) {
            // 当客户端断开连接后
            System.err.println("Failed to handle client request: " + e.getMessage()
        } finally {
            try {
                clientSocket.close();
            } catch (IOException e) {
                System.err.println("Failed to close client socket: " + e.getMessage
            }
        }
    }
}
```

C

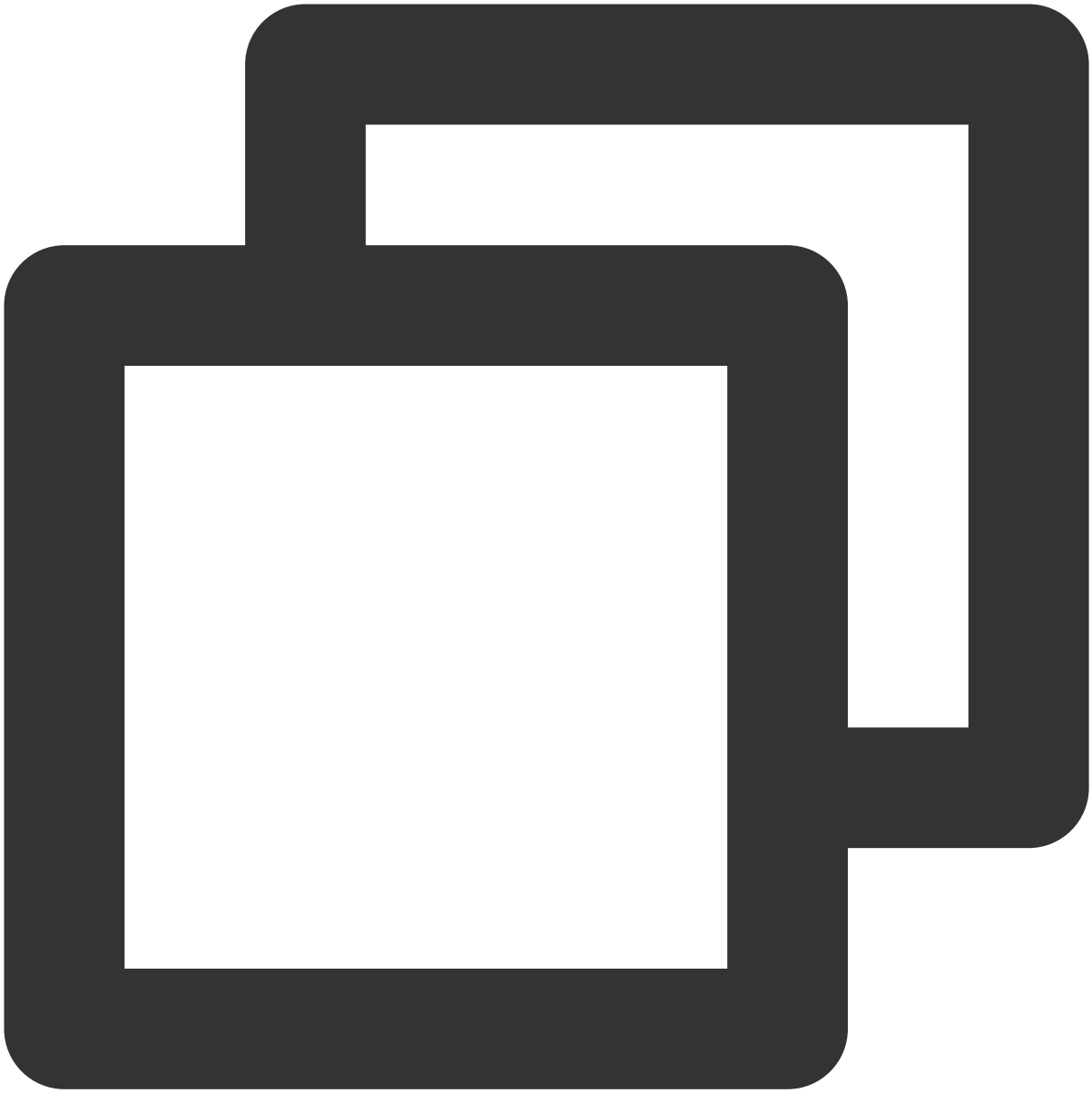
Java



```
#include <sys/socket.h>
#include <stdio.h>
#include <unistd.h>
#include <netinet/in.h>
#include <memory.h>
#include <arpa/inet.h>

int main(int argc,  char **argv)
{
        int     l_sockfd;
    // 服务器地址采用v6结构
```

```
    struct sockaddr_in6 serveraddr;
// 客户端地址采用v6结构
struct sockaddr_in6 clientAddr;
    int server_port = 10000;

    memset(&serveraddr, 0, sizeof(serveraddr));

// 创建socket
    l_sockfd = socket(AF_INET6, SOCK_STREAM, 0);
    if (l_sockfd == -1){
            printf("Failed to create socket.\\n");
            return -1;
    }
// 设置服务器地址信息
    memset(&serveraddr, 0, sizeof(struct sockaddr_in6));
    serveraddr.sin6_family = AF_INET6;
    serveraddr.sin6_port = htons(server_port);
    serveraddr.sin6_addr = in6addr_any;

    int isReuse = 1;
    setsockopt(l_sockfd, SOL_SOCKET,SO_REUSEADDR,(const char*)&isReuse,sizeof(i
// 关联socket和服务器地址信息
    int nRet = bind(l_sockfd,(struct sockaddr*)&serveraddr, sizeof(serveraddr))
    if(-1 == nRet)
    {
            printf("bind error\\n");
            return -1;
    }
// 监听socket
    listen(l_sockfd, 5);

    int clientAddrLen = sizeof(clientAddr);
memset(&clientAddr, 0, sizeof(clientAddr));

// 接受来自客户端的连接请求
int linkFd = accept(l_sockfd, (struct sockaddr*)&clientAddr, &clientAddrLen);
if(-1 == linkFd)
{
    printf("accept error\\n");
    return -1;
}

// 这里收到的客户端地址信息全部都采用v6的结构进行存储
// 其中，客户端的IPv4地址也被映射成了一个IPv6的地址，例如：::ffff:119.29.1.1
char addr_p[128] = {0};
inet_ntop(AF_INET6, (void *)&clientAddr.sin6_addr, addr_p, (socklen_t )sizeof(a
printf("accept %s : %d successful\\n", addr_p, ntohs(clientAddr.sin6_port));
```

```
    // 业务修改点：通过系统宏定义IN6_IS_ADDR_V4MAPPED来判断一个IPv6地址是否是IPv4的映射地址（
    if(IN6_IS_ADDR_V4MAPPED(&clientAddr.sin6_addr)) {
        struct sockaddr_in real_v4_sin;
        memset (&real_v4_sin, 0, sizeof (struct sockaddr_in));
        real_v4_sin.sin_family = AF_INET;
        real_v4_sin.sin_port = clientAddr.sin6_port;
        // 读取最后四个字节即为客户端真实IPv4地址
        memcpy (&real_v4_sin.sin_addr, ((char *)&clientAddr.sin6_addr) + 12, 4);
        printf("connect %s successful\\n", inet_ntoa(real_v4_sin.sin_addr));
    }

        close(l_sockfd);
    return 0;
}
```

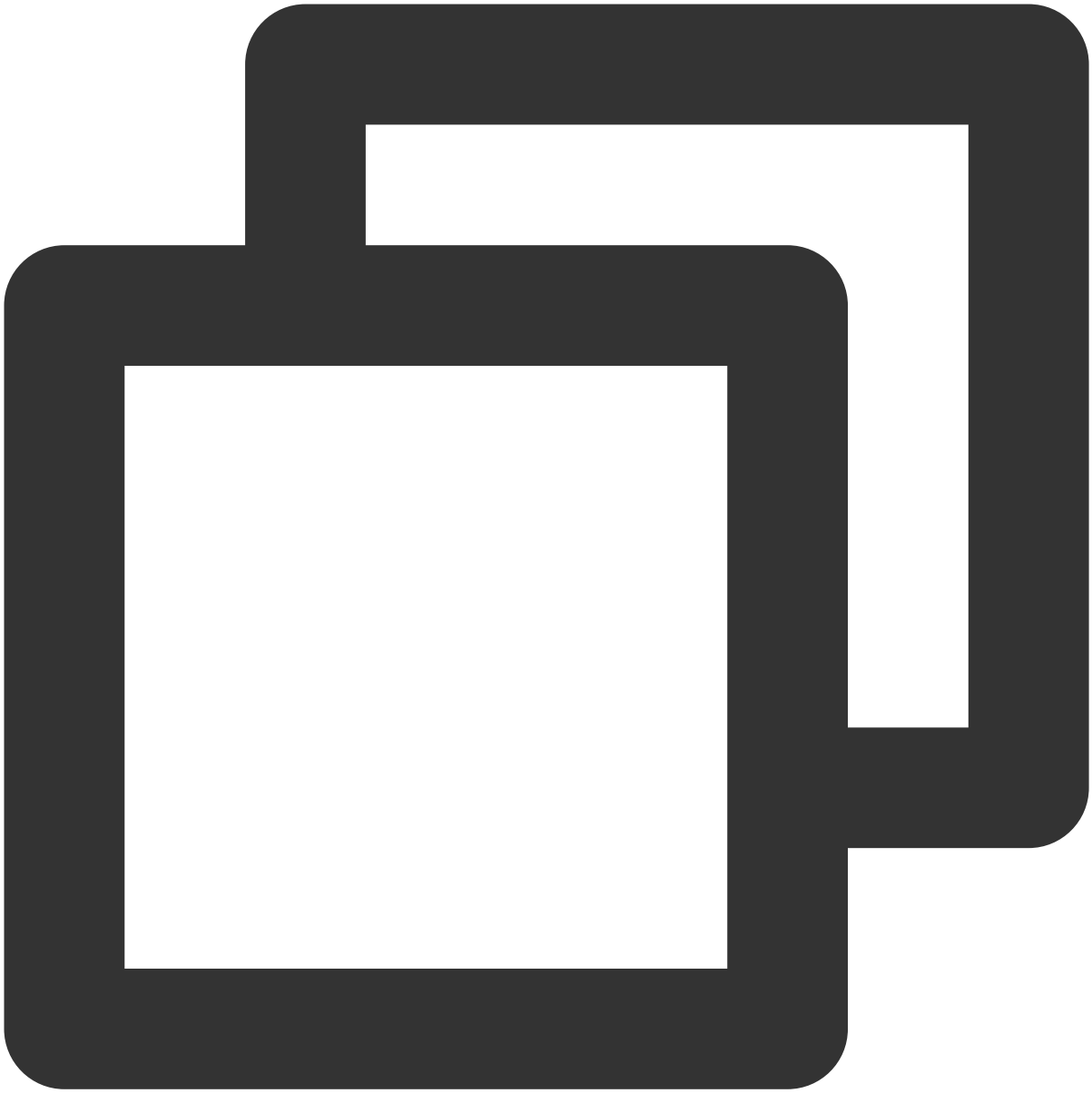


```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketAddress;

public class ServerDemo {
```

```
    /** 若采用 IPv4 的地址结构搭建服务，使用 IPV4_HOST */
    public static final String IPV4_HOST = "0.0.0.0";

    /** 若采用 IPv6 的地址结构搭建服务，使用 IPV6_HOST */
    public static final String IPV6_HOST = "::";

    public static void main(String[] args) {
        int serverPort = 10000;
        try (ServerSocket serverSocket = new ServerSocket()) {
            // 设置地址复用
            serverSocket.setReuseAddress(true);
            // 绑定服务器地址和端口，这里使用 IPv4
            serverSocket.bind(new InetSocketAddress(InetAddress.getByName(IPV6_HOST
            System.out.println("Server is listening on port " + serverPort);

            while (true) {
                // 接受客户端连接
                Socket clientSocket = serverSocket.accept();
                System.out.println("New client connected: " + clientSocket.getRemot

                // 处理客户端请求
                handleClientRequest(clientSocket);
            }
        } catch (IOException e) {
            System.err.println("Failed to create server socket: " + e.getMessage())
        }
    }

    /**
     * 处理函数，具体业务具体实现，这里只做为示例
     * 此函数的作用是将 client 的输入原封不动的返回给 client
     */
    private static void handleClientRequest(Socket clientSocket) {
        try (InputStream inputStream = clientSocket.getInputStream();
             OutputStream outputStream = clientSocket.getOutputStream()) {

            // 读取客户端发来的数据
            byte[] buffer = new byte[1024];
            int bytesRead;
            while ((bytesRead = inputStream.read(buffer)) != -1) {
                // 将接收到的数据原样回复给客户端
                outputStream.write(buffer, 0, bytesRead);
            }

        } catch (IOException e) {
            // 当客户端断开连接后
            System.err.println("Failed to handle client request: " + e.getMessage()
```

```
            } finally {
                try {
                    clientSocket.close();
                } catch (IOException e) {
                    System.err.println("Failed to close client socket: " + e.getMessage
                }
            }
        }
    }
```

控制台输出结果

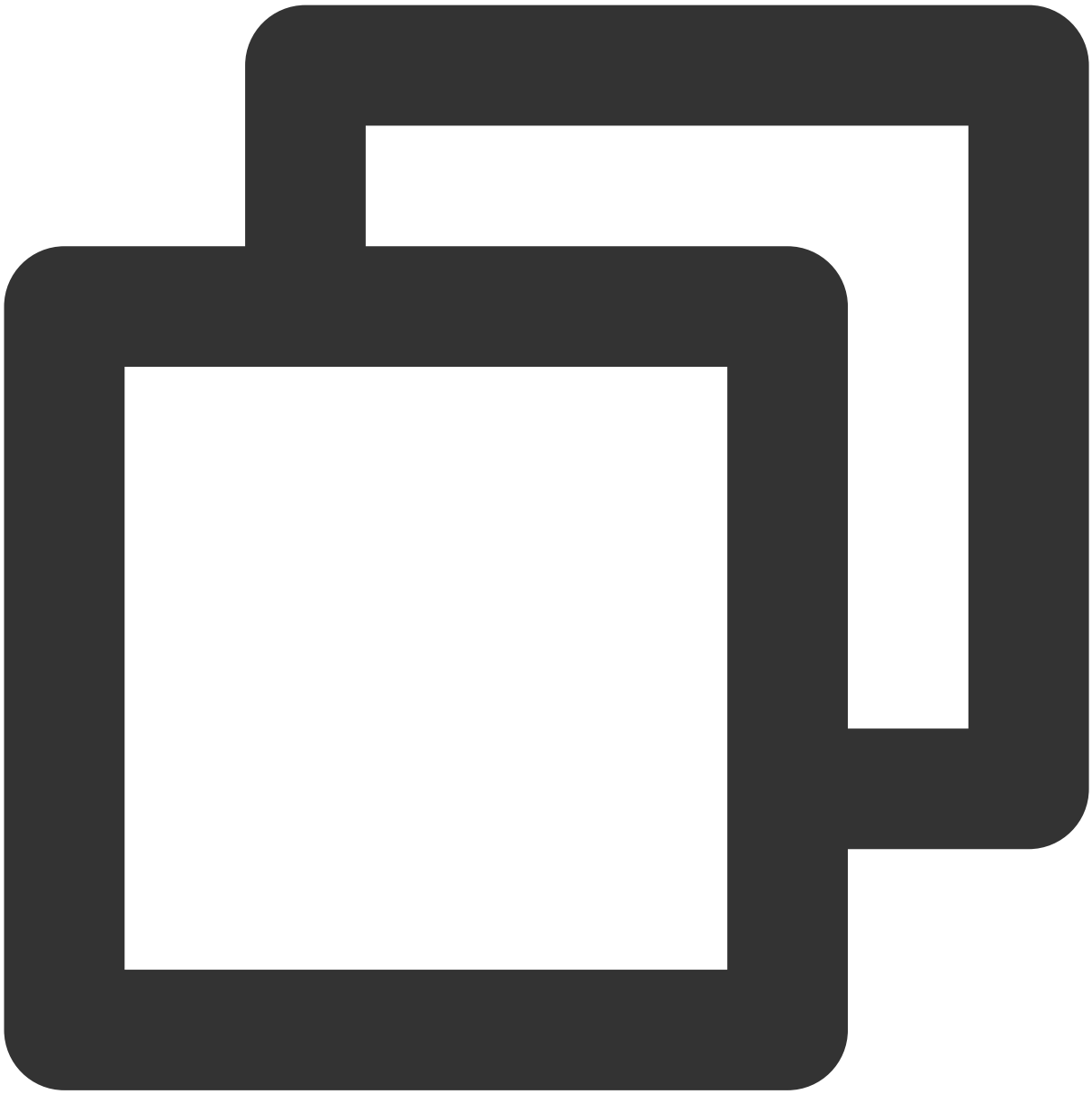


```
Server is listening on port 10000
New client connected: /127.0.0.1:50680
New client connected: /0:0:0:0:0:0:0:1:51124
New client connected: /127.0.0.1:51136
```

相关参考

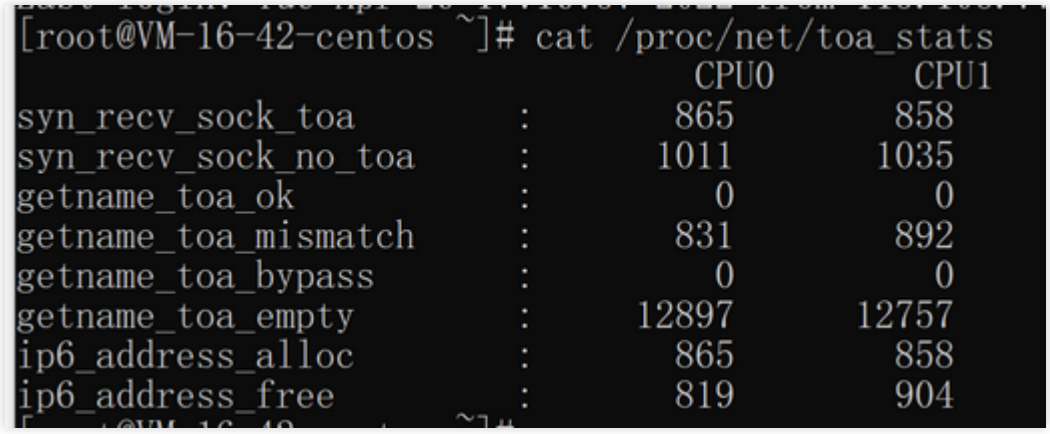
监控 TOA 运行状态

为保障 TOA 内核模块运行的稳定性，TOA 内核模块还提供了监控功能。在插入 toa.ko 内核模块后，可以通过执行以下命令方式监控 TOA 模块的工作状态。



```
cat /proc/net/toa_stats
```

TOA 运行状态如下：



其中主要的监控指标对应的含义如下所示：

指标名称	说明
syn_recv_sock_toa	接收带有 TOA 信息的连接个数。
syn_recv_sock_no_toa	接收并不带有 TOA 信息的连接个数。
getname_toa_ok	调用 getsockopt 获取源 IP 成功即会增加此计数，另外调用 accept 函数接收客户端请求时也会增加此计数。
getname_toa_mismatch	调用 getsockopt 获取源 IP 时，当类型不匹配时，此计数增加。例如某条客户端连接内存放的是 IPv4 源 IP，并非为 IPv6 地址时，此计数便会增加。
getname_toa_empty	对某一个不含有 TOA 的客户端文件描述符调用 getsockopt 函数时，此计数便会增加。
ip6_address_alloc	当 TOA 内核模块获取 TCP 数据包中保存的源 IP、源 Port 时，会申请空间保存信息。
ip6_address_free	当连接释放时，toa 内核模块会释放先前用于保存源 IP、源 port 的内存，在所有连接都关闭的情况下，所有 CPU 的此计数相加应等于 ip6_address_alloc 的计数。

通过 Proxy Protocol V1/V2 协议获取客户端真实 IP

概述

最近更新时间：2023-06-29 15:37:55

本文介绍了使用四层代理加速时，如何通过 Proxy Protocol V1/V2 协议获取客户端真实 IP。

使用场景

当数据报文通过四层加速通道进行加速时，为了将客户端真实 IP 和 Port 信息可传递给源站服务器，您可选择通过 Proxy Protocol V1/V2 协议来传递客户端 IP 和 Port 信息，协议介绍可参考：Proxy Protocol V1/V2。

源站在解析获取客户端真实 IP 时，根据不同的业务场景及部署方式，可以参考以下两种方式了解如何获取客户端真实 IP：
方式一：如果您的源站服务为 TCP 协议时，Nginx 已原生支持 Proxy Protocol 协议，建议在业务服务器前增加已支持 Proxy Protocol V1/V2 协议的 Nginx 服务器来获取客户端真实 IP。具体步骤请参见 通过 Nginx 获取客户端真实 IP。
方式二：如果您的源站服务为 UDP 协议，或者需在业务源站服务内直接解析 TCP 协议场景下的客户端真实 IP 以进行业务调度，可以在业务源站内参考 Proxy Protocol 协议内的示例代码开发自行解析 Proxy Protocol 字段。具体步骤请参见 在业务服务器解析客户端真实 IP。

方式一：通过 Nginx 获取客户端真实 IP

最近更新时间：2023-09-11 17:42:07

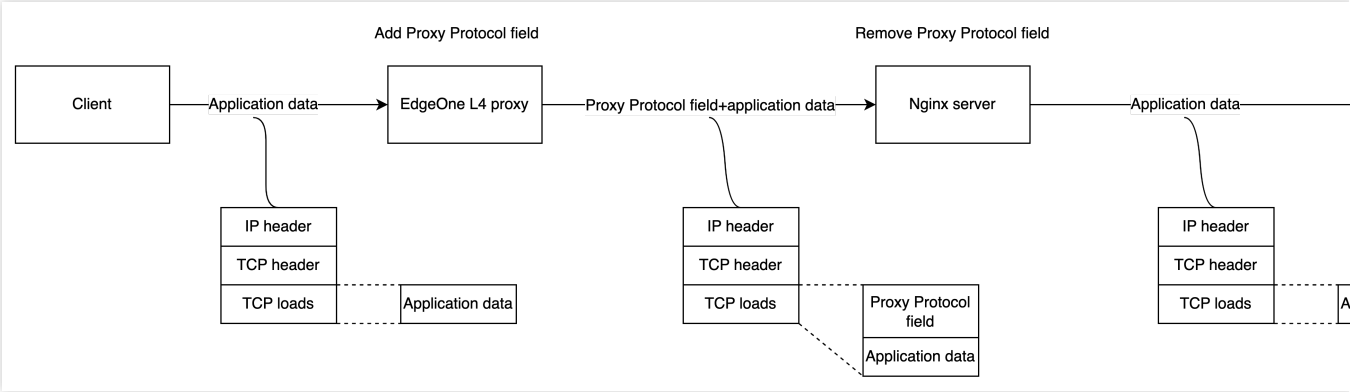
使用场景

如果您的源站服务为 TCP 协议，且当前 Nginx 已原生支持 Proxy Protocol 协议，建议在业务服务器前增加已支持 Proxy Protocol V1/V2 协议的 Nginx 服务器，以获取客户端真实 IP。您可以参考以下步骤来进行操作。

说明：

如果您当前源站服务为 TCP 协议，但是不希望部署 Nginx 服务来单独解析客户端真实 IP，希望在业务服务器内直接解析获取客户端真实 IP 以辅助业务判断逻辑，您可以参考：在业务服务器解析客户端真实 IP。

部署方式



如上图所示，您需要在业务服务器前部署 Nginx 服务器，由 Nginx 服务器来完成 Proxy Protocol 字段的卸载，对真实客户端的 IP 地址收集可以通过在 Nginx 服务器上分析 Nginx 日志来完成，而业务服务器不用去关心真实客户端地址。此时，在 EdgeOne 四层代理服务中配置源站地址时，可将源站地址指向该 Nginx 服务即可。

操作步骤

步骤一：部署 Nginx 服务

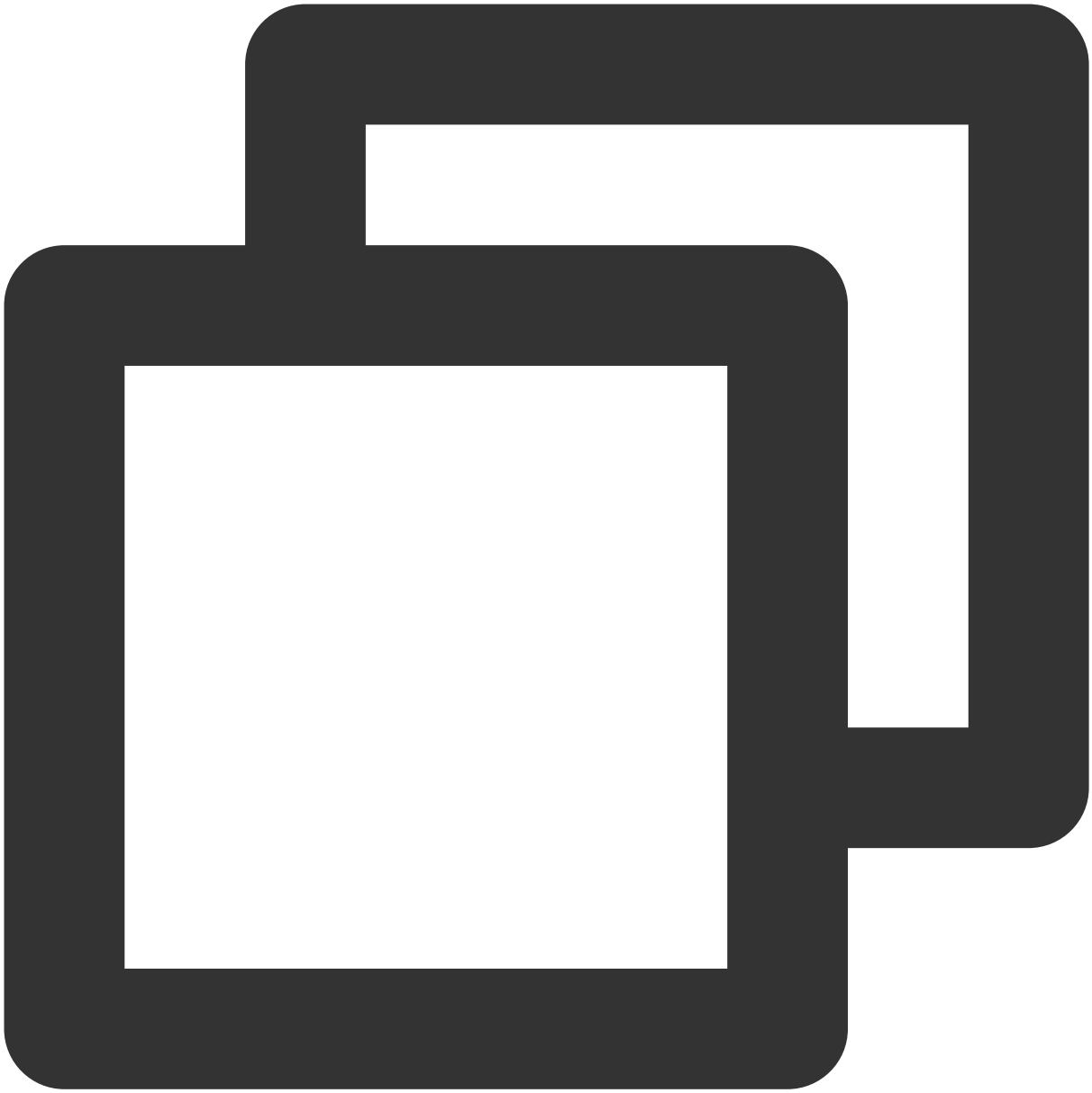
请根据您所需使用的 Proxy Protocol 协议版本，选择对应的 Nginx 版本进行部署：

支持 Proxy Protocol V1：Nginx Plus R11 及以后，Nginx Open Source 1.11.4及以后。

支持 Proxy Protocol V2：Nginx Plus R16 及以后，Nginx Open Source 1.13.11及以后。

如需了解其他 Nginx 版本对 Proxy Protocol 协议的支持，请参考 Nginx 文档：Accepting the PROXY Protocol。

为了在 Nginx 上启用四层代理服务，您需要安装 Nginx-1.18.0 版本及其 stream 模块。以下是安装步骤：

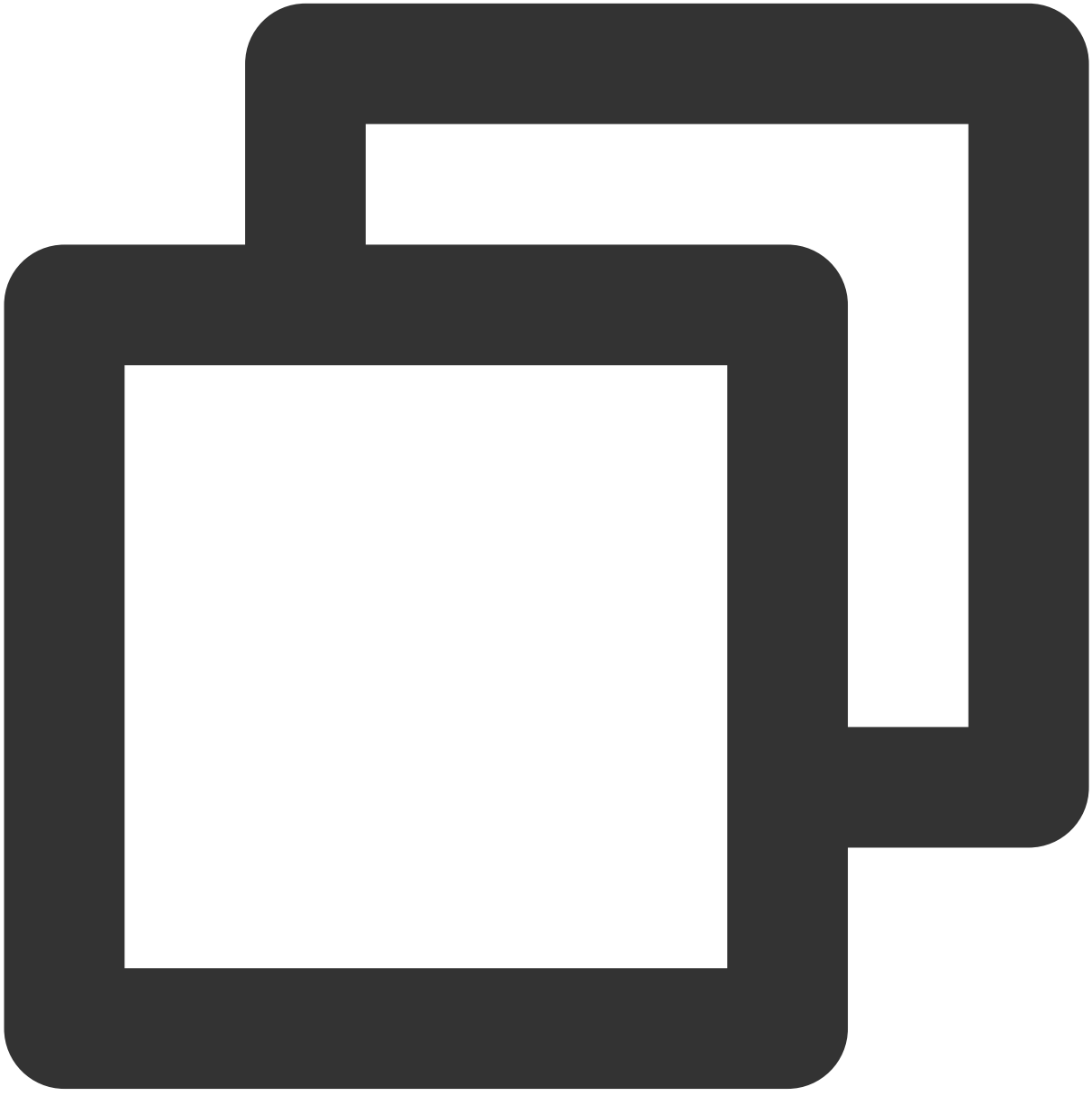


```
# 安装nginx编译环境依赖
yum -y install gcc gcc-c++ autoconf automake
yum -y install zlib zlib-devel openssl openssl-devel pcre-devel

# 解压源码包
tar -zxvf nginx-1.18.0.tar.gz
# 进入目录
cd nginx-1.18.0
# 设置nginx编译安装配置，带上--with-stream
./configure --prefix=/opt/nginx --sbin-path=/opt/nginx/sbin/nginx --conf-path=/opt/
# 编译
```

```
make
# 安装
make install
```

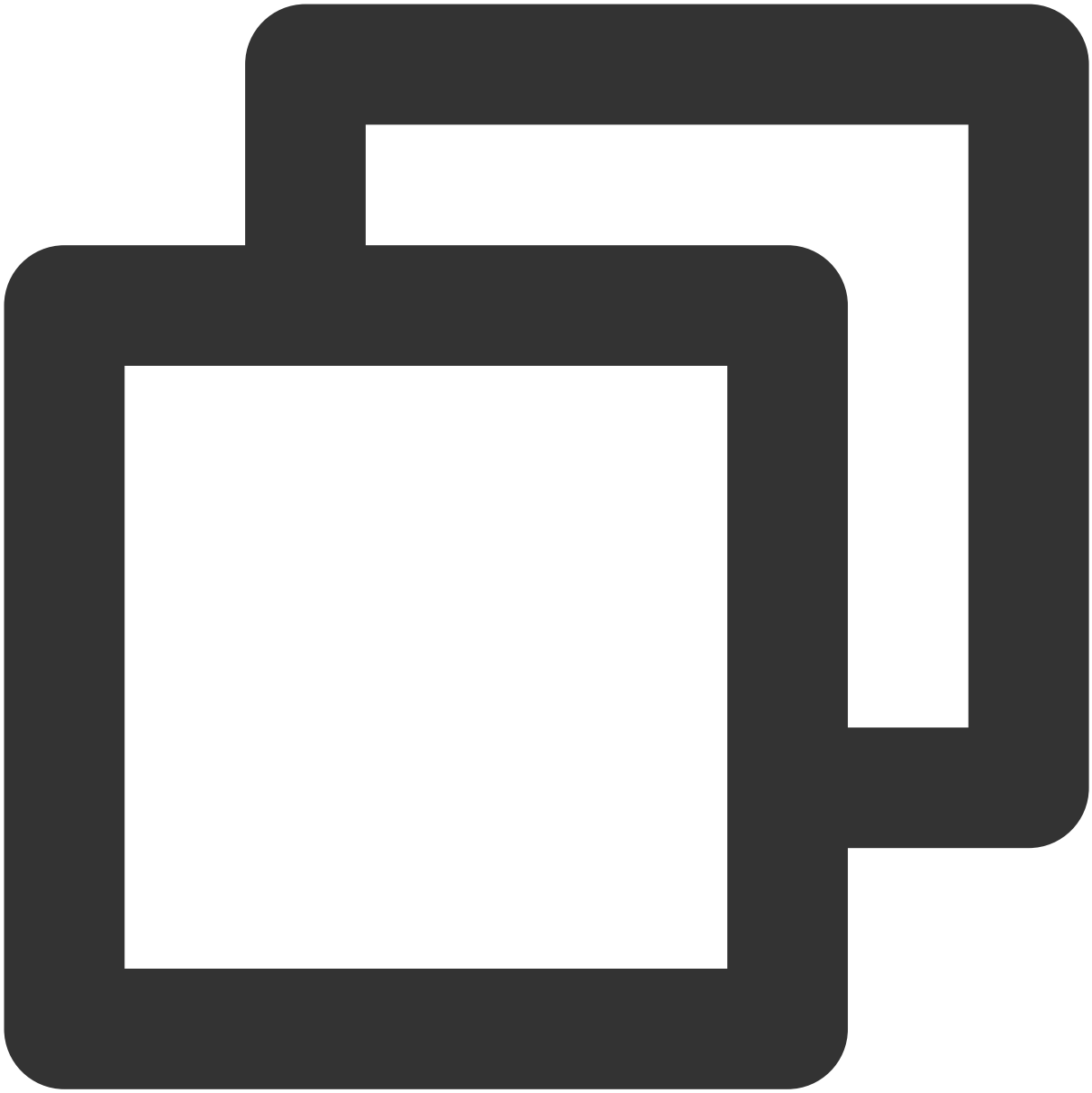
步骤二：配置 Nginx 内 Stream 模块

以 Nginx-1.18.0版本为例，可以执行以下命令来打开 Nginx 的配置文件 nginx.conf：



```
vi /opt/nginx/conf/nginx.conf
```

Stream 模块配置内容参考如下：

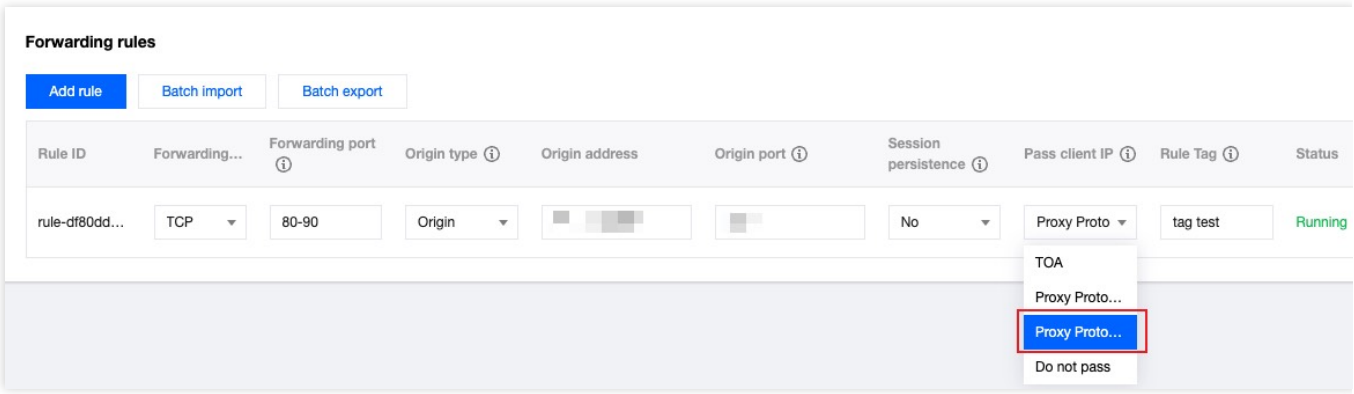


```
stream {
    # 设置日志格式，其中proxy_protocol_addr为解析PP协议拿到的客户端地址，remote_addr为上一跳
    log_format basic '$proxy_protocol_addr -$remote_addr [$time_local] '
                     '$protocol $bytes_sent $bytes_received '
                     '$session_time';

      access_log  logs/stream.access.log  basic;
    # upstream配置
        upstream RealServer {
                hash $remote_addr consistent;
        # 其中127.0.0.1:8888为业务服务器的地址和端口
```

```
                server 127.0.0.1:8888 max_fails=3 fail_timeout=30s;
        }
    # server配置
        server{
        # 四层监听端口，对应着四层代理配置的源站端口，需配置proxy_protocol支持对入包的PP协议解
                listen 10000 proxy_protocol;
                proxy_connect_timeout 1s;
                proxy_timeout 3s;
                proxy_pass RealServer;
        }
}
```

步骤三：配置四层代理转发规则

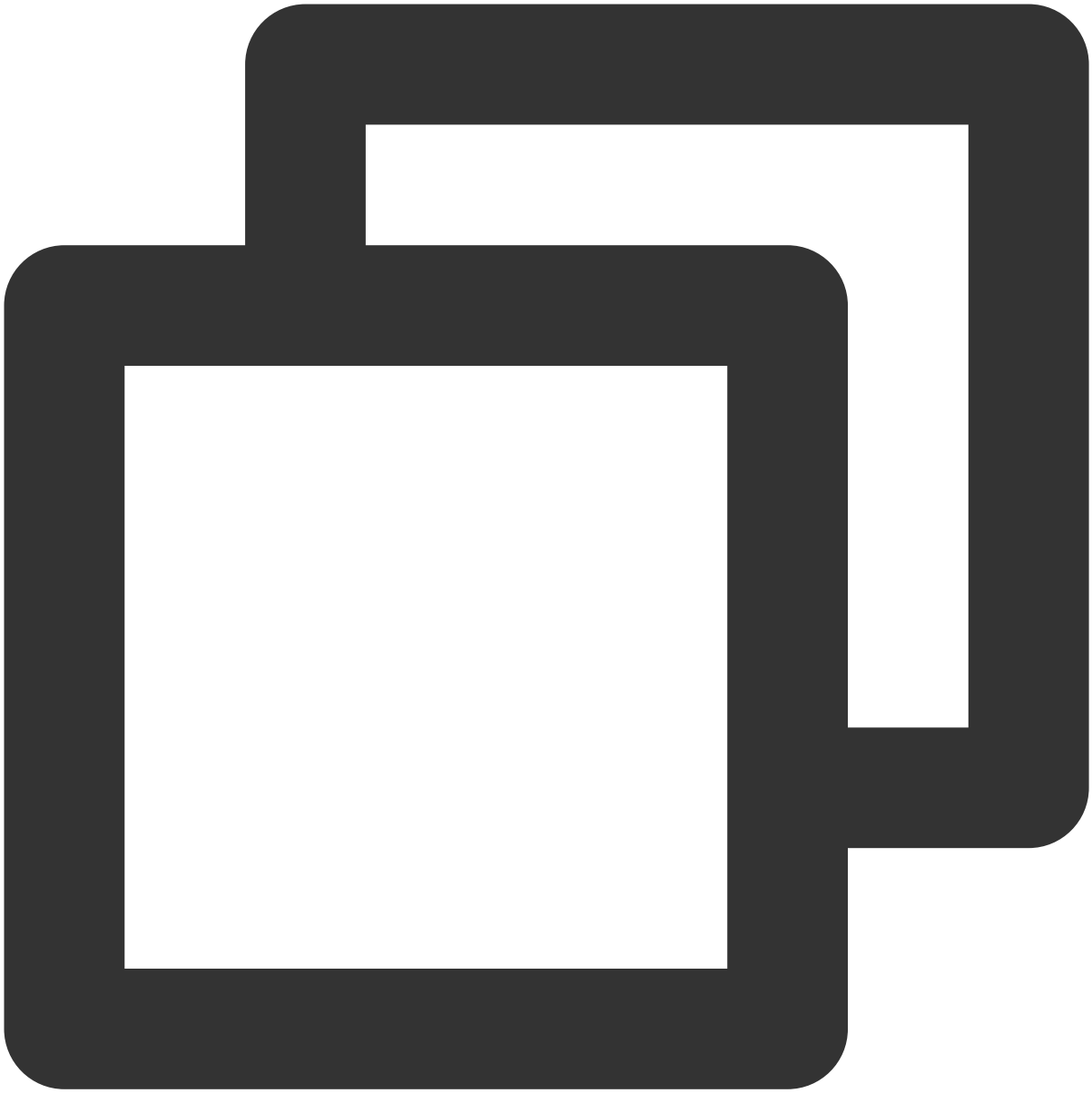
配置完 Nginx 服务后，您可以前往控制台的四层代理服务，修改四层代理转发规则。将源站地址修改为当前 Nginx 服务的 IP，源站端口为 步骤二 内配置的四层监听端口。传递客户端 IP 时，根据您当前使用的 Nginx 版本支持情况，选择 Proxy Protocol V1 或 Proxy Protocol V2。



步骤四：模拟客户端请求，验证结果

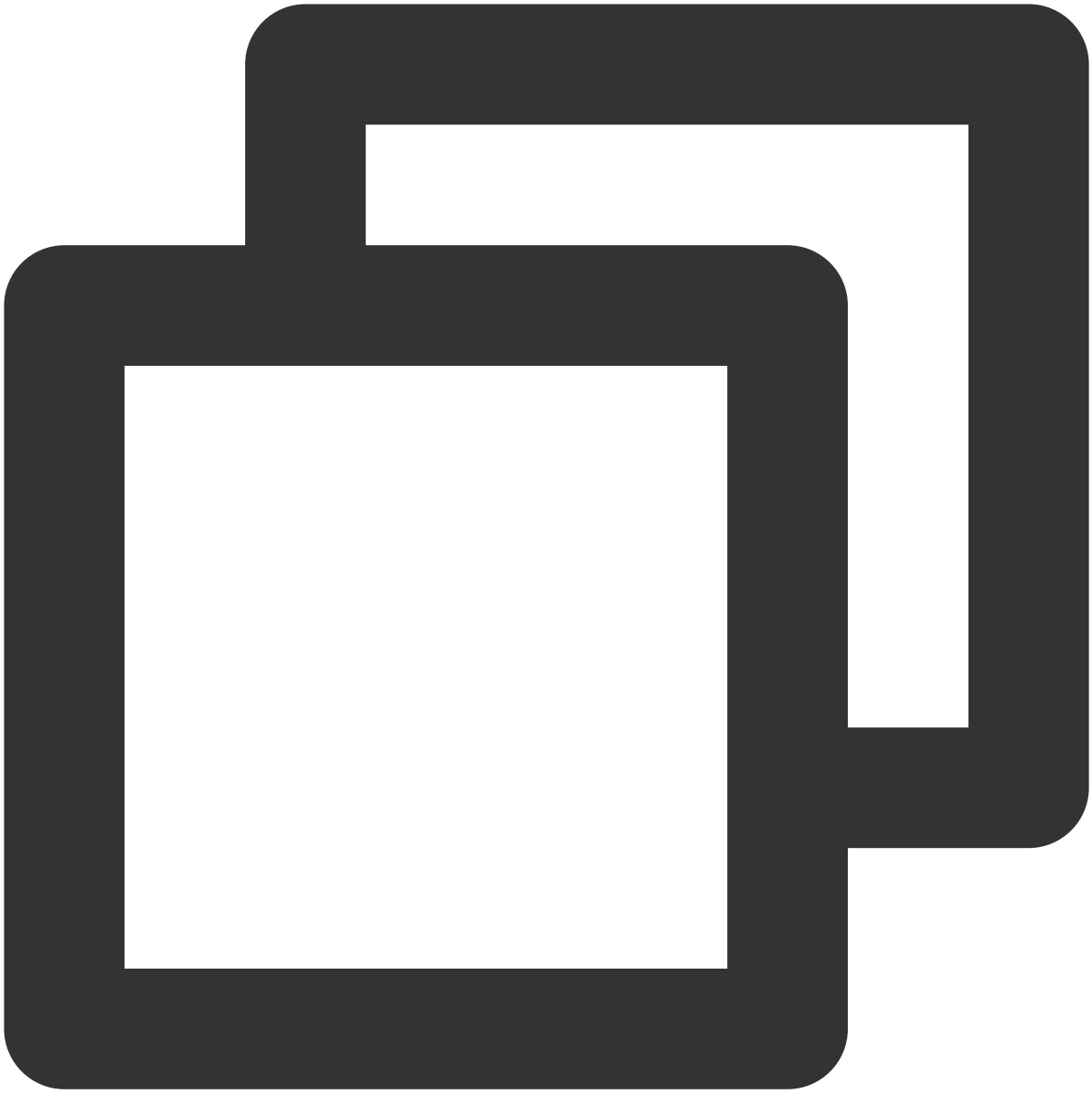
可以通过搭建 TCP 服务，然后使用另一台服务器模拟客户端请求进行验证。具体示例如下：

1. 可以使用 Python 在当前服务器上创建一个 HTTP 服务，来模拟 TCP 服务。



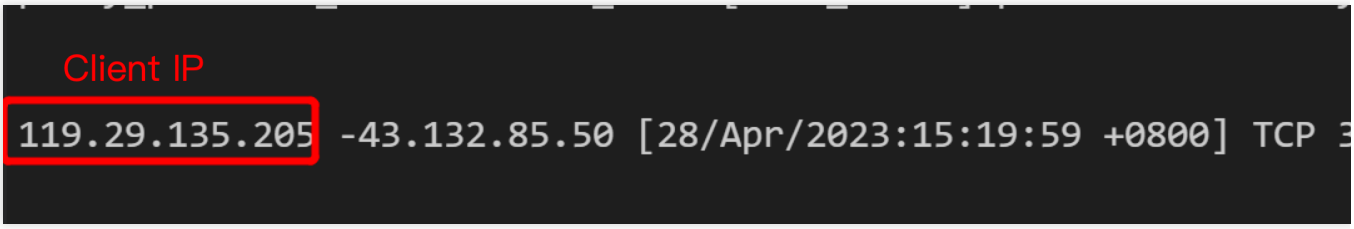
```
# 基于python2
python2 -m SimpleHTTPServer 8888

# 基于python3
python3 -m http.server 8888
```

2. 用另一台服务器充当客户端，构造客户端请求，以 Curl 请求来模拟 TCP 请求：

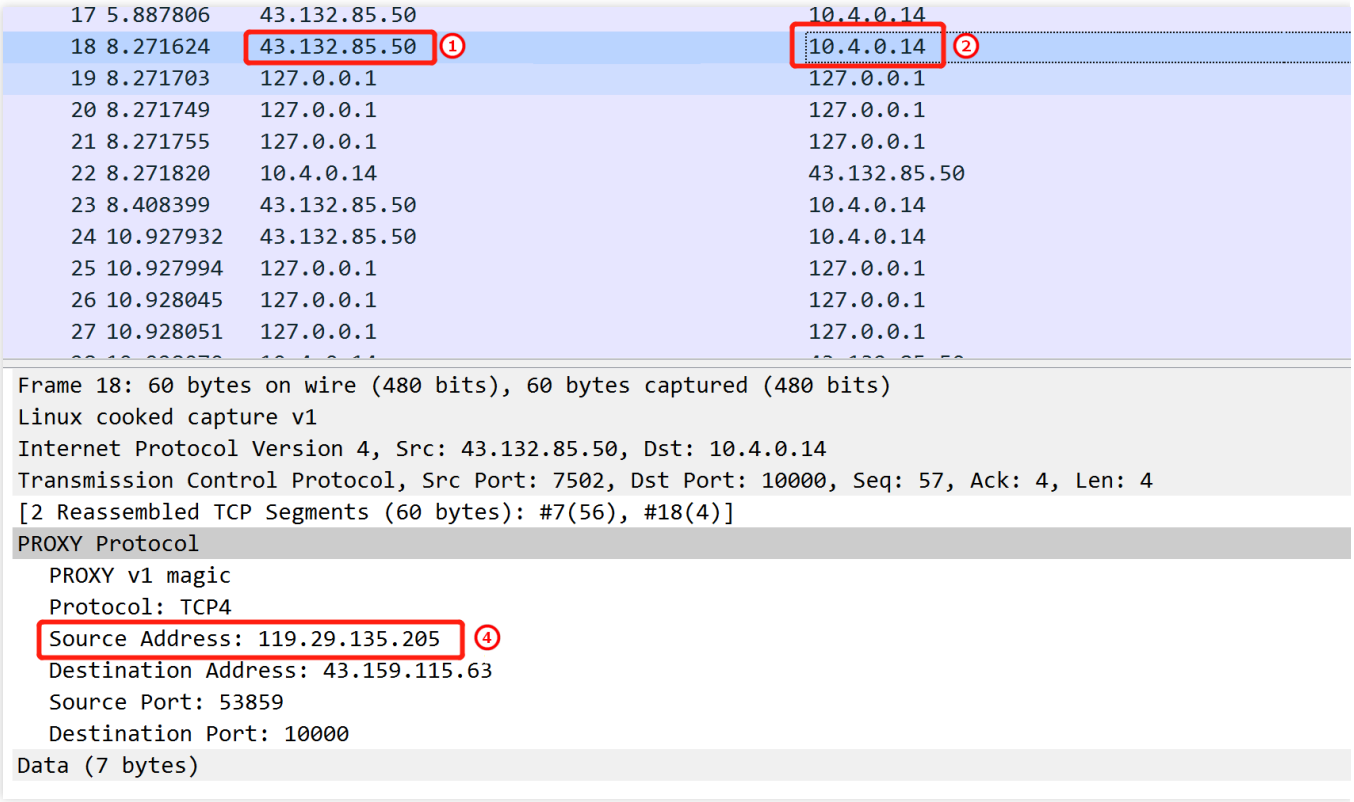


```
# 利用curl发起http请求，其中域名为四层代理域名，8888为四层代理转发端口
curl -i "http://d42f15b7a9b47488.davidjli.xyz.acc.edgeonedy1.com:8888/"
```

3. 在 Nginx 服务器上查看 Nginx 日志，如下展示：



您可以在 Nginx 服务器上进行抓包，并通过 Wireshark 分析数据包。在 TCP 握手完成后，第一个业务数据包的前面会添加 Proxy Protocol 字段。下面是 Proxy Protocol V1 版本的示例：①四层代理出口 IP、②Nginx 服务器 IP、③协议版本、④真实客户端 IP 地址。



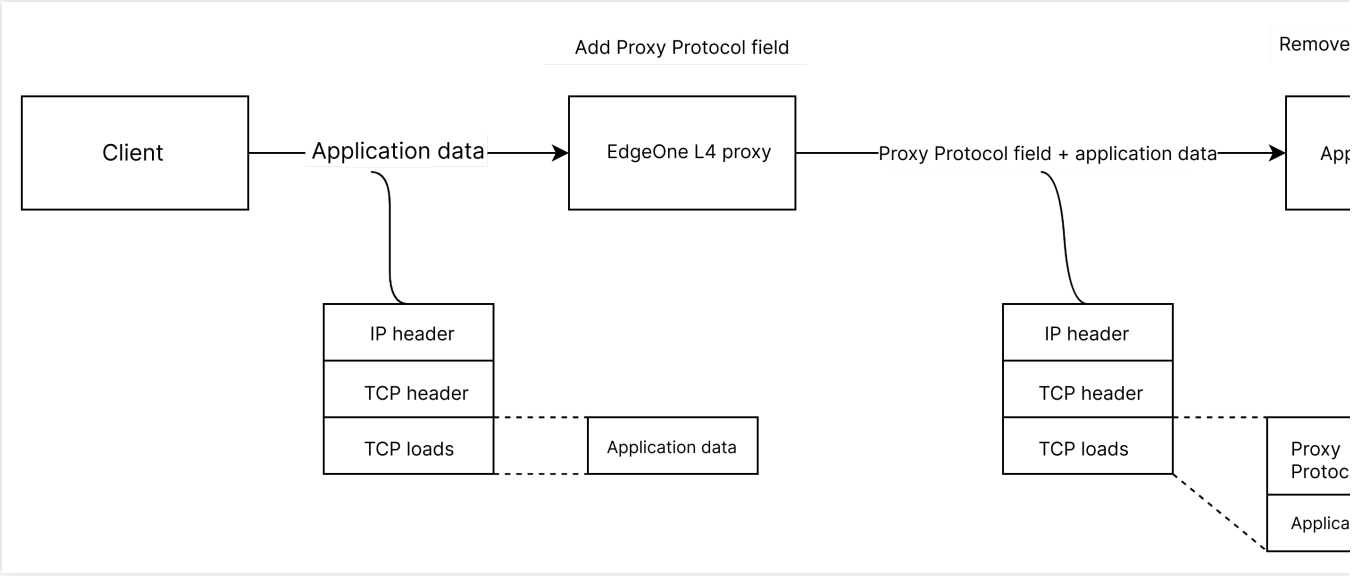
方式二：在业务服务器解析客户端真实 IP

最近更新时间：2023-09-11 17:43:42

使用场景

场景一：如果您的源站服务为 UDP 协议，仅 Proxy Protocol V2 支持 UDP 协议传递真实客户端 IP。但由于 Nginx 不支持对 Proxy Protocol V2 UDP 场景下协议的解析，因此需要在业务服务器上自行完成对 Proxy Protocol V2 协议的解析以获取客户端真实 IP。
场景二：如果您当前源站服务为 TCP 协议，但是需要在业务源站服务器内通过客户端真实 IP 进行业务判断时，您需要在业务服务器上自行完成对 Proxy Protocol V1/V2 协议的解析来获取客户端真实 IP。

部署框图

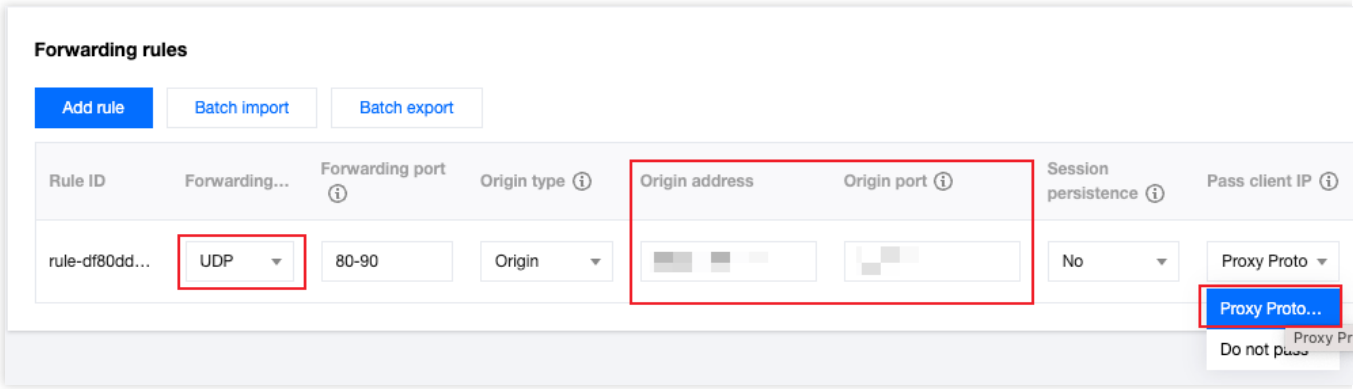


如上图所示，您可以通过 EdgeOne 四层代理模块，配置四层代理指向您的业务服务器，由 EdgeOne 四层代理服务在传输数据中添加 Proxy Protocol 字段，业务服务器进行解析。

操作步骤

步骤一：配置四层代理转发规则

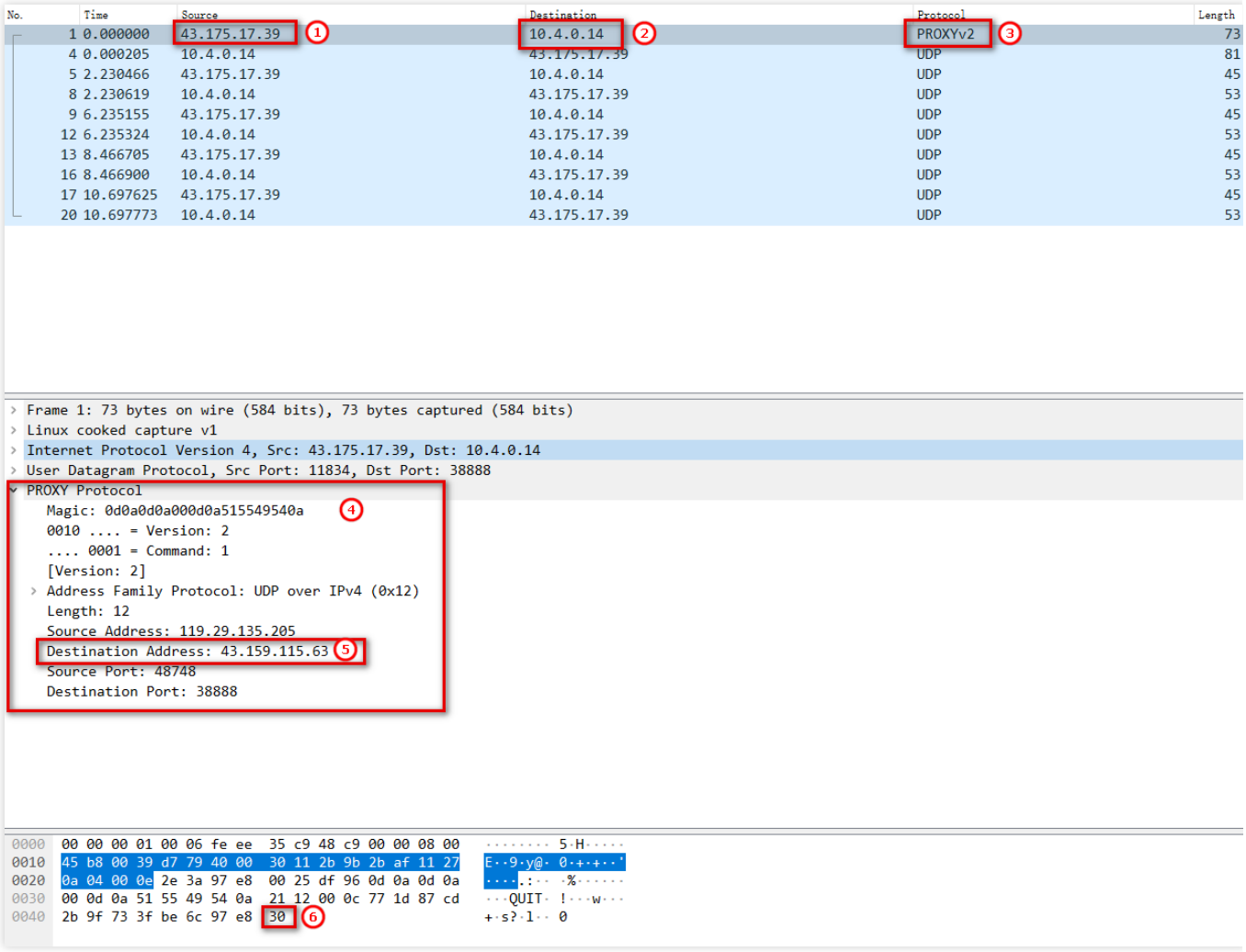
前往控制台内的四层代理服务，修改四层代理转发规则，填写对应的业务源站地址、源站端口，如果您当前的转发协议为 UDP，传递客户端 IP 选择为 Proxy Protocol V2。如果当前的转发协议为 TCP，则传递客户端真实 IP 选择为 Proxy Protocol V1/V2 均可。



步骤二：在业务服务器解析 Proxy Protocol 字段获取真实客户端 IP

您需要参考 Proxy Protocol 协议 内的 sample code 开发解析 Proxy Protocol 字段，获取的客户端 IP 格式可参考：Proxy Protocol V1/V2 获取的客户端真实 IP 格式。

在 UDP 传输场景中，使用 Proxy Protocol V2 版本时，会将 Proxy Protocol 字段添加到第一个 UDP 数据报文上。其中①四层代理出口 IP、②源站地址、③协议版本、④Proxy Protocol 字段、⑤真实客户端地址、⑥业务数据。

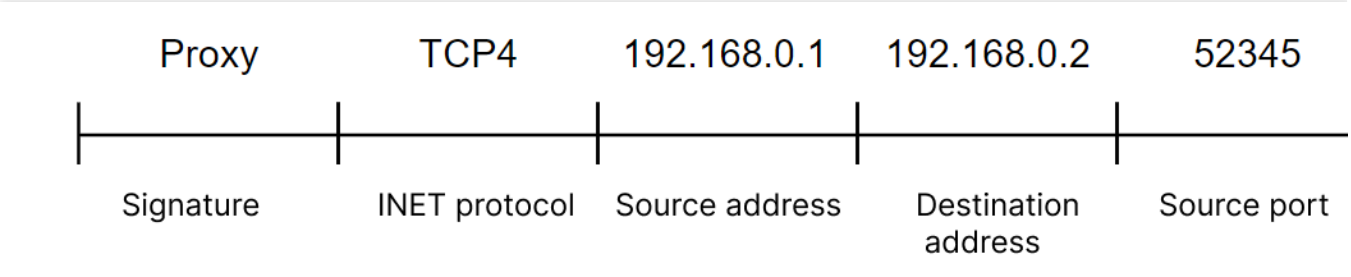

No. Time Source Destination Protocol Length
1 0.000000 43.175.17.39 10.4.0.14 PROXYv2 73
4 0.000205 10.4.0.14 43.175.17.39 UDP 81
5 2.230466 43.175.17.39 10.4.0.14 UDP 45
8 2.230619 10.4.0.14 43.175.17.39 UDP 53
9 6.235155 43.175.17.39 10.4.0.14 UDP 45
12 6.235324 10.4.0.14 43.175.17.39 UDP 53
13 8.466705 43.175.17.39 10.4.0.14 UDP 45
16 8.466900 10.4.0.14 43.175.17.39 UDP 53
17 10.697625 43.175.17.39 10.4.0.14 UDP 45
20 10.697773 10.4.0.14 43.175.17.39 UDP 53
1
2
3
Frame 1: 73 bytes on wire (584 bits), 73 bytes captured (584 bits)
Linux cooked capture v1
Internet Protocol Version 4, Src: 43.175.17.39, Dst: 10.4.0.14
User Datagram Protocol, Src Port: 11834, Dst Port: 38888
PROXY Protocol
Magic: 0d0a0d0a000d0a515549540a
4
0010 .... = Version: 2
.... 0001 = Command: 1
[Version: 2]
Address Family Protocol: UDP over IPv4 (0x12)
Length: 12
Source Address: 119.29.135.205
Destination Address: 43.159.115.63
5
Source Port: 48748
Destination Port: 38888
0000 00 00 00 01 00 06 fe ee 35 c9 48 c9 00 00 08 00
0010 45 b8 00 39 d7 79 40 00 30 11 2b 9b 2b af 11 27
0020 0a 04 00 0e 2e 3a 97 e8 00 25 df 96 0d 0a 0d 0a
0030 00 0d 0a 51 55 49 54 0a 21 12 00 0c 77 1d 87 cd
0040 2b 9f 73 3f be 6c 97 e8 30
6

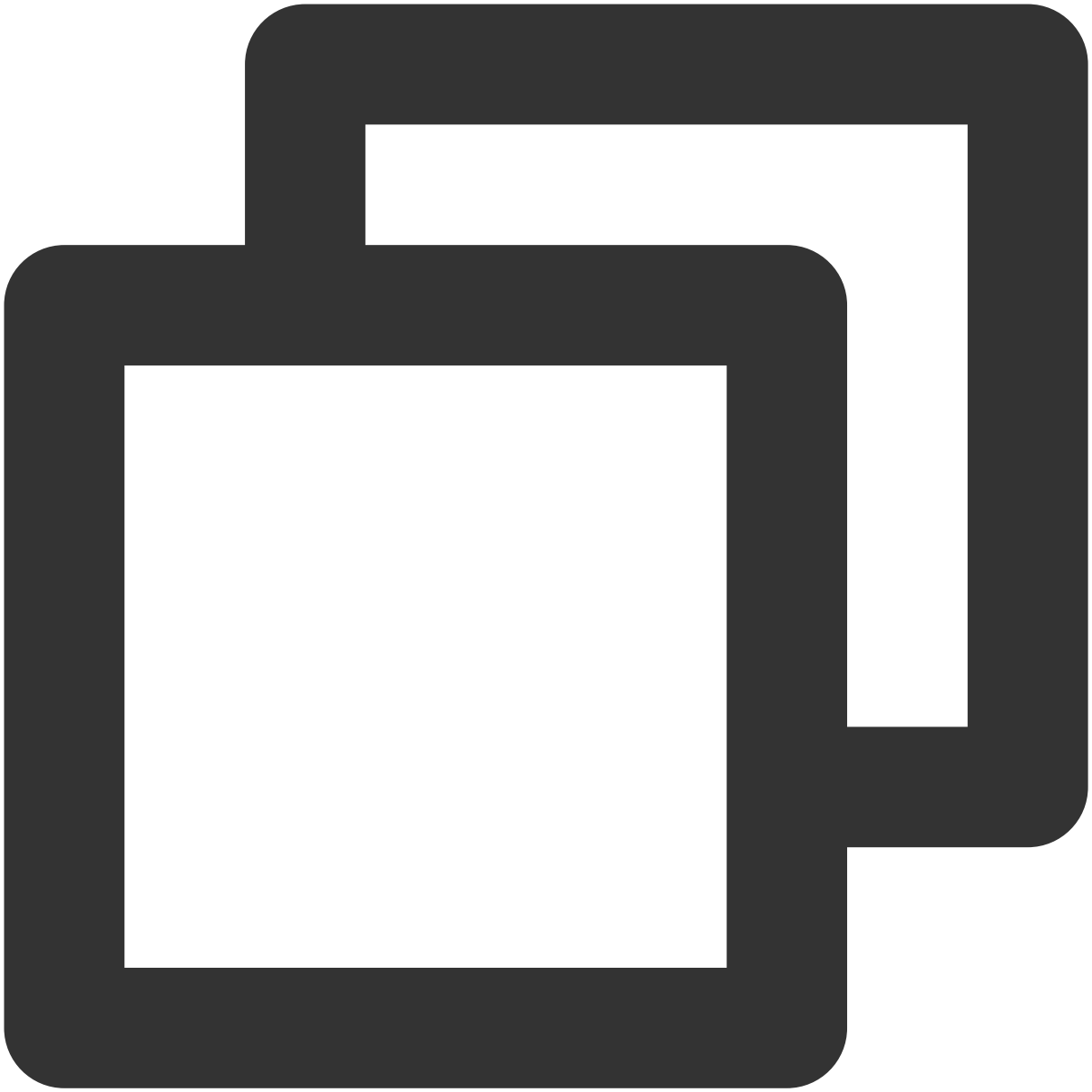
Proxy Protocol V1/V2 获取的客户端真实 IP 格式

最近更新时间：2023-06-29 15:37:55

Proxy Protocol V1

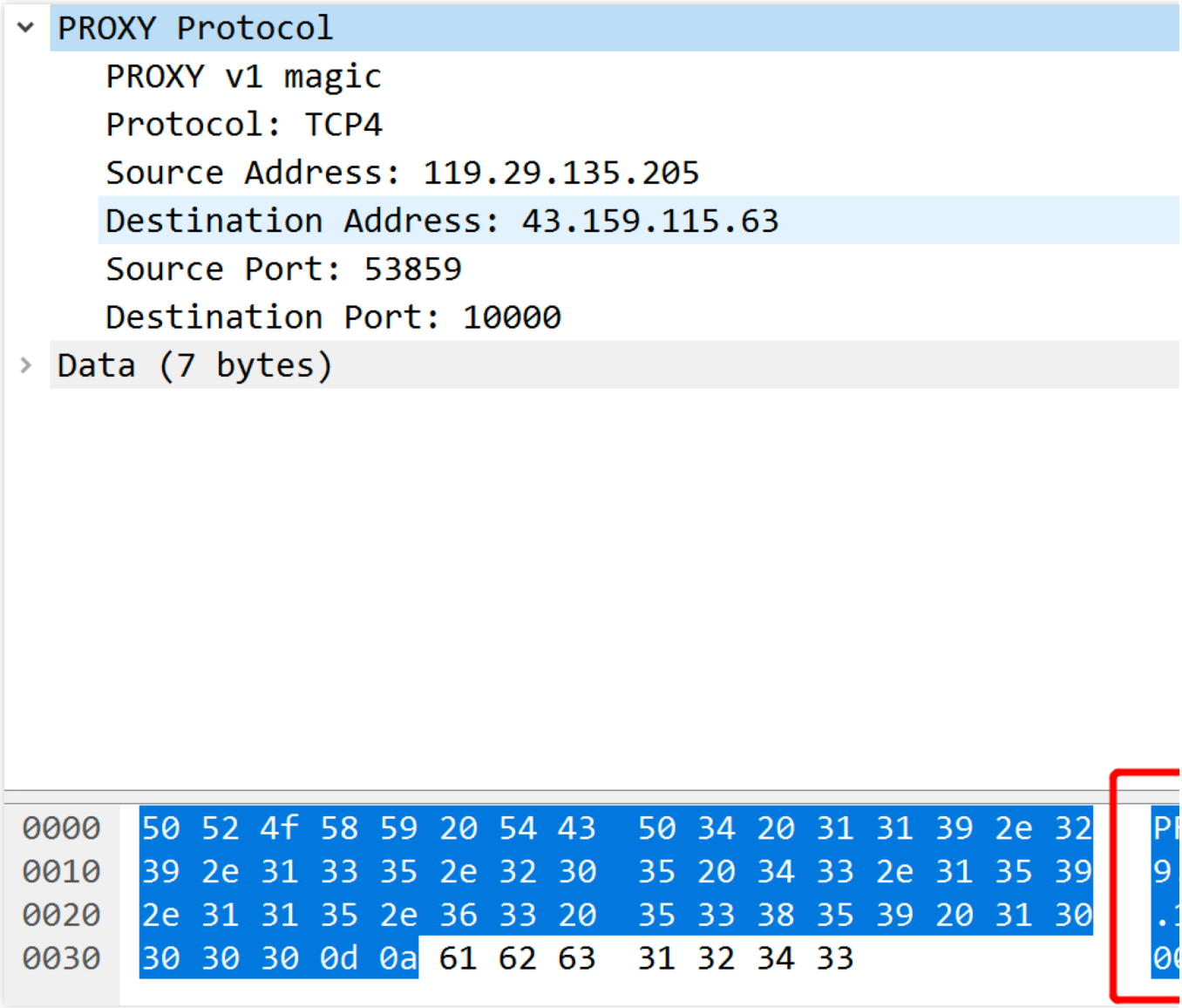
Proxy Protocol V1 协议仅支持 TCPv4、TCPv6 协议，并采用字符串格式。其格式如下：





```
PROXY TCP4 192.168.0.1 192.168.0.11 56324 443\\r\\n
```

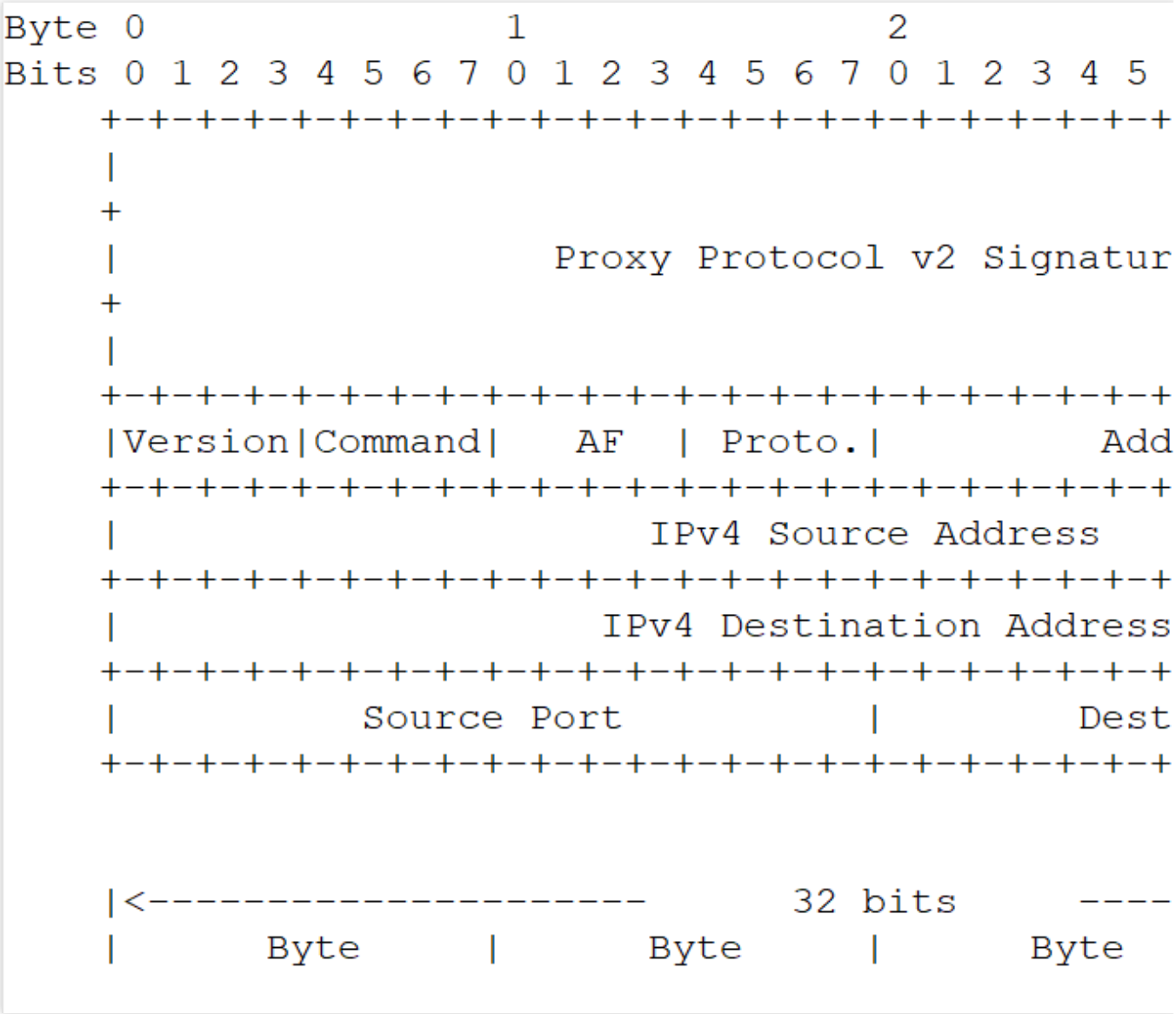
通过使用 Wireshark 抓包工具，可以查看到以下信息：



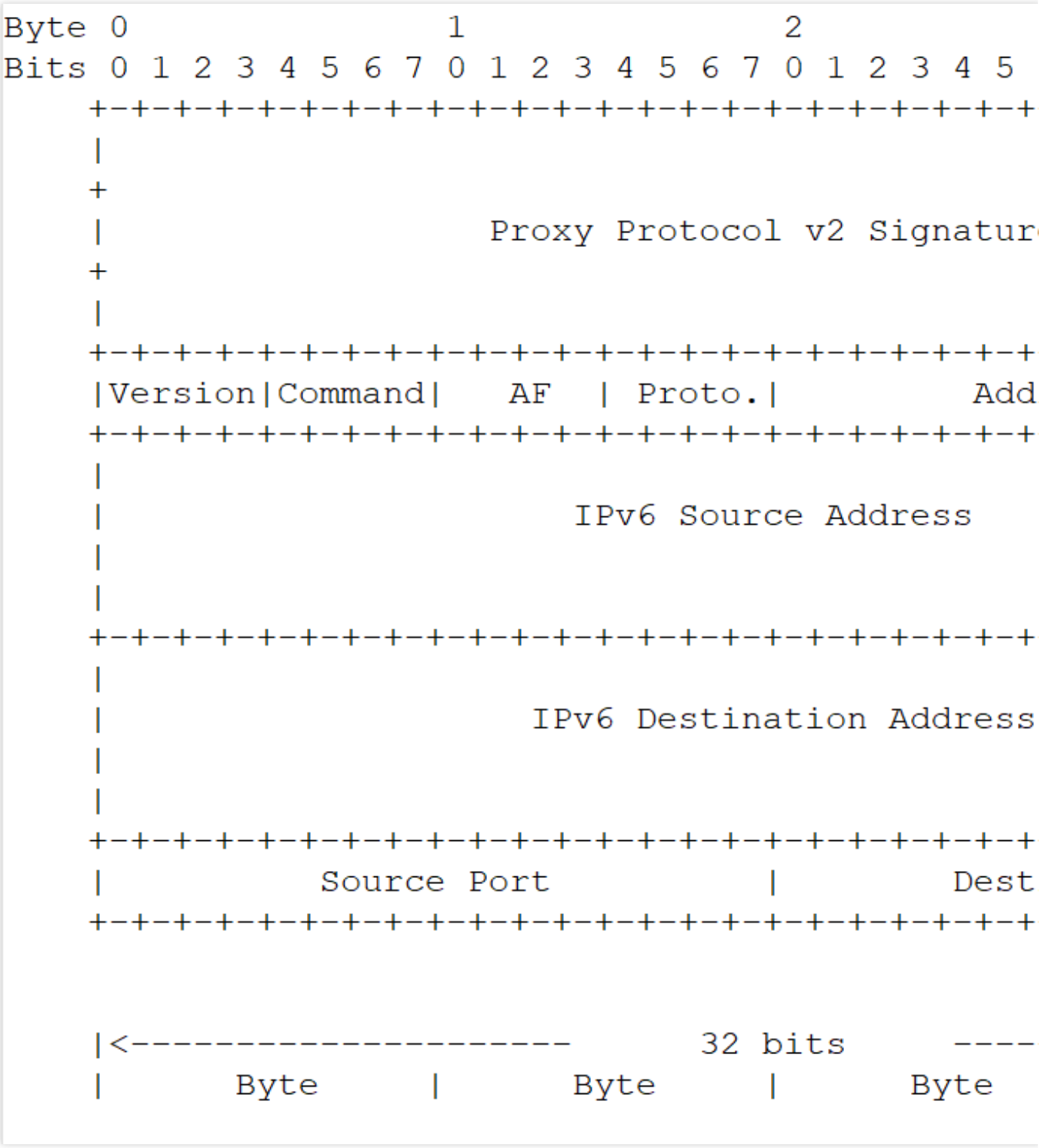
Proxy Protocol V2

Proxy Protocol V2 协议采用二进制格式，支持 TCPv4、TCPv6、UDPv4、UDPv6 协议，其格式如下：

IPv4格式


Byte 0 1 2
Bits 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5
Proxy Protocol v2 Signatur
Version
Command
AF
Proto.
Add
IPv4 Source Address
IPv4 Destination Address
Source Port
Dest
32 bits
Byte
Byte
Byte

IPv6格式


Byte 0 1 2
Bits 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5
Proxy Protocol v2 Signatur
Version
Command
AF
Proto.
Add
IPv6 Source Address
IPv6 Destination Address
Source Port
Dest
32 bits
Byte
Byte
Byte

通过 SPP 协议传递客户端真实 IP

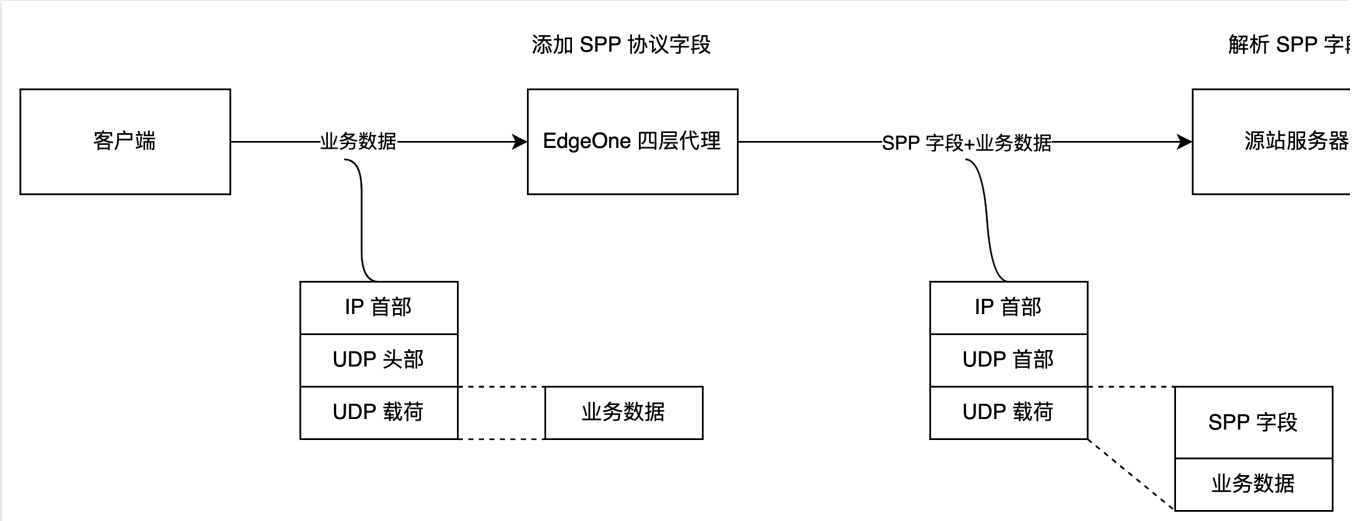
最近更新时间：2024-07-30 16:21:58

使用场景

SPP （Simple Proxy Protocol Header，以下简称 SPP）协议是一种自定义的协议头格式，用于代理服务器将真实客户端 IP 和其他相关信息传递给后端服务器，用于记录日志、实现访问控制、负载均衡或者故障排除等场景。SPP 协议头固定长度为38字节，相比 Proxy Protocol V2 协议更为简单。
如果您当前现有的后端业务服务为 UDP 服务，已经支持了 SPP 协议或者希望使用更简单的解析方式，您可以使用 SPP 协议来传递客户端真实 IP。EdgeOne 的四层代理支持根据 SPP 协议标准传递真实客户端 IP 至业务服务器，您可以在服务端自行对该协议解析来获取真实客户端 IP 和 Port。

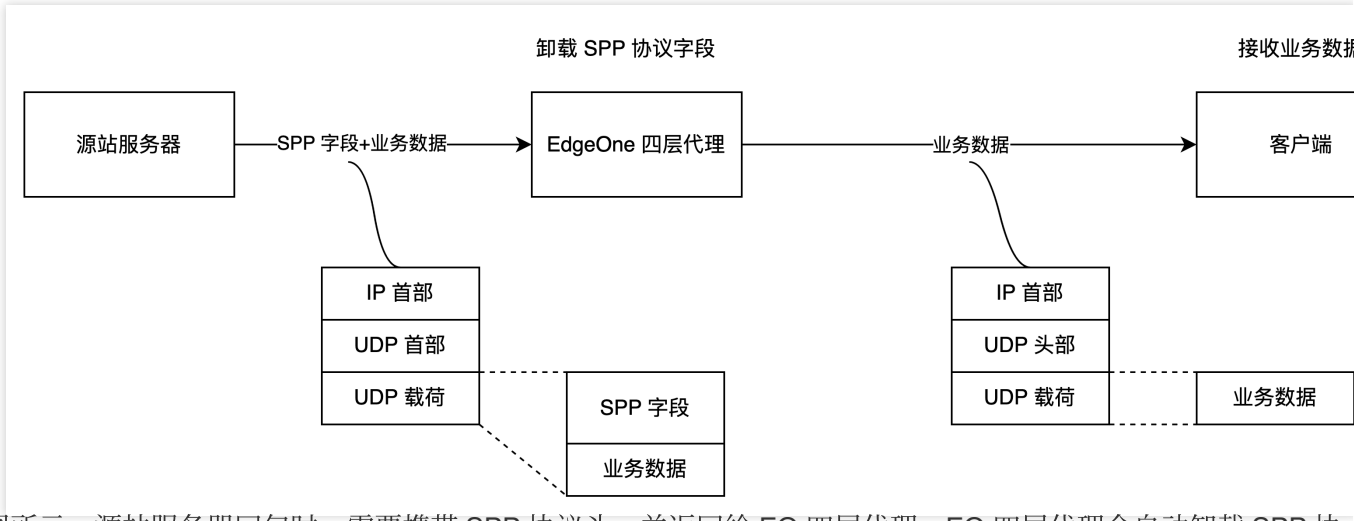
EdgeOne 对 SPP 协议处理流程

请求访问



如上图所示，当您使用 SPP 协议传递客户端 IP 和 Port 时，EdgeOne 的四层代理会自动将客户端的真实 IP 和 Port 以固定 38 字节长度，按照 SPP 协议头格式添加到每个有效载荷之前，您需要在源站服务器解析 SPP 头部字段才能获取客户端的真实 IP 和 Port。

源站响应



如上图所示，源站服务器回包时，需要携带 SPP 协议头一并返回给 EO 四层代理，EO 四层代理会自动卸载 SPP 协议头。

注意：

如果源站服务器没有返回 SPP 协议头，则会导致 EO 四层代理截断有效载荷的业务数据。

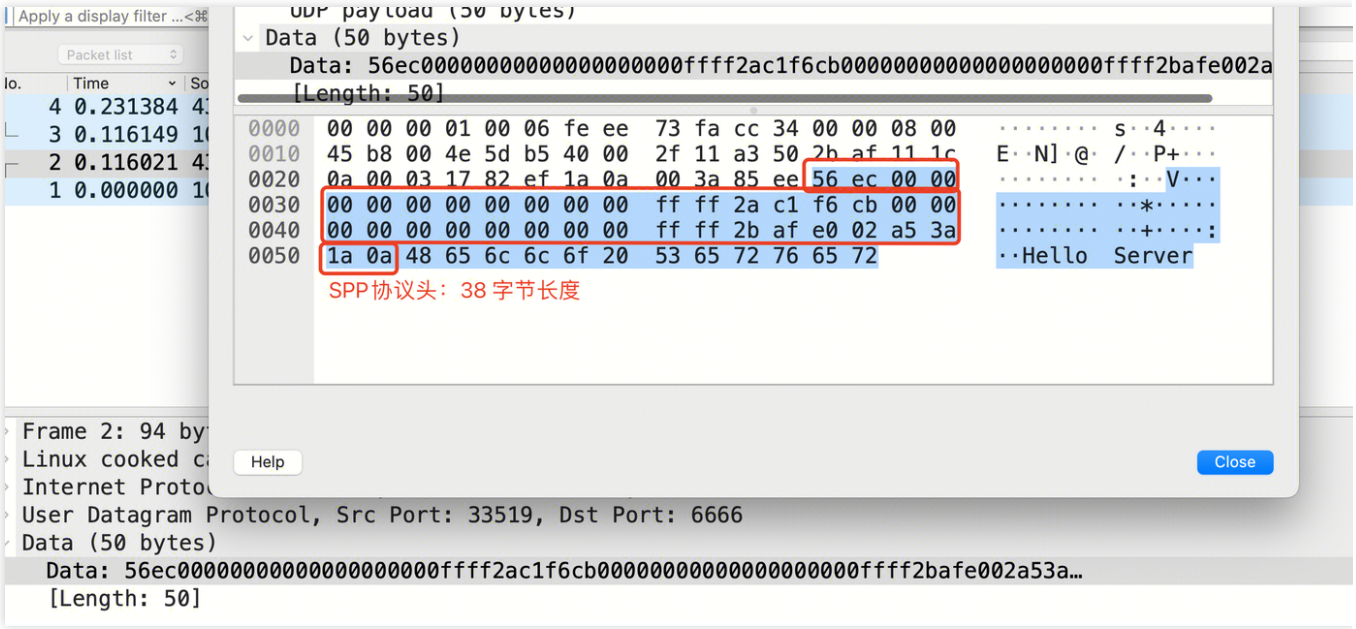
操作步骤

步骤1：配置四层代理转发规则

1. 登录 边缘安全加速平台 EO 控制台，在左侧菜单栏中，单击**站点列表**，在站点列表内单击需配置的**站点**。
2. 在站点详情页面，单击**四层代理**。
3. 在四层代理页面，选择需要修改的四层代理实例，单击**配置**。
4. 选择需要传递客户端真实 IP 的四层代理规则，单击**编辑**。
5. 填写对应的业务源站地址、源站端口，转发协议选择 UDP，传递客户端 IP 选择 Simple Proxy Protocol，单击**保存**。

步骤2：在源站服务器解析 SPP 字段获取客户端真实 IP

您可以参考 SPP 协议头格式和 示例代码，在源站服务器上解析 SPP 字段，使用 SPP 协议传输真实客户端 IP 时，服务端获取的业务包数据格式如下：

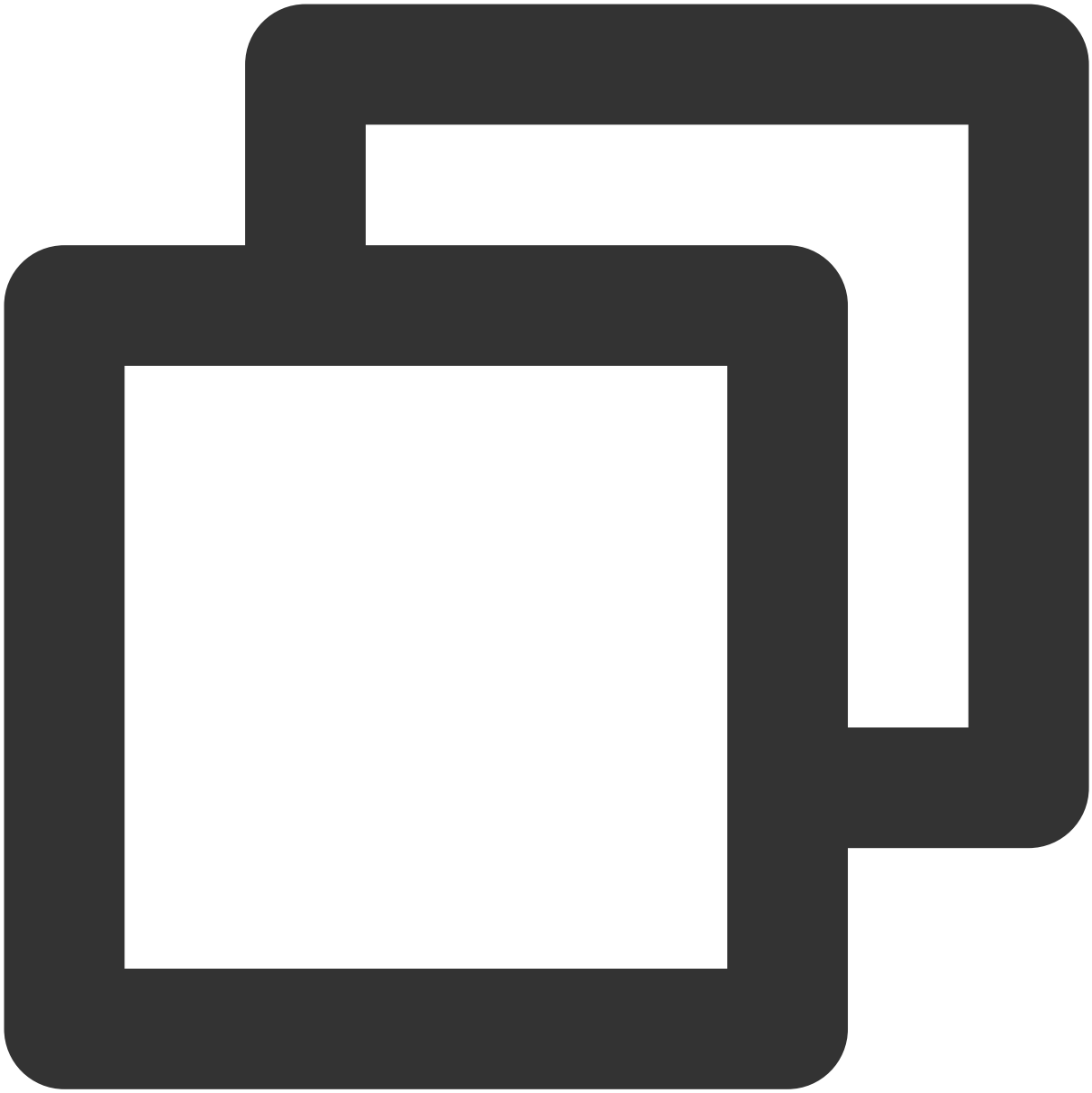


您可以

参考以下示例代码来对业务数据解析获取到真实客户端 IP。

Go

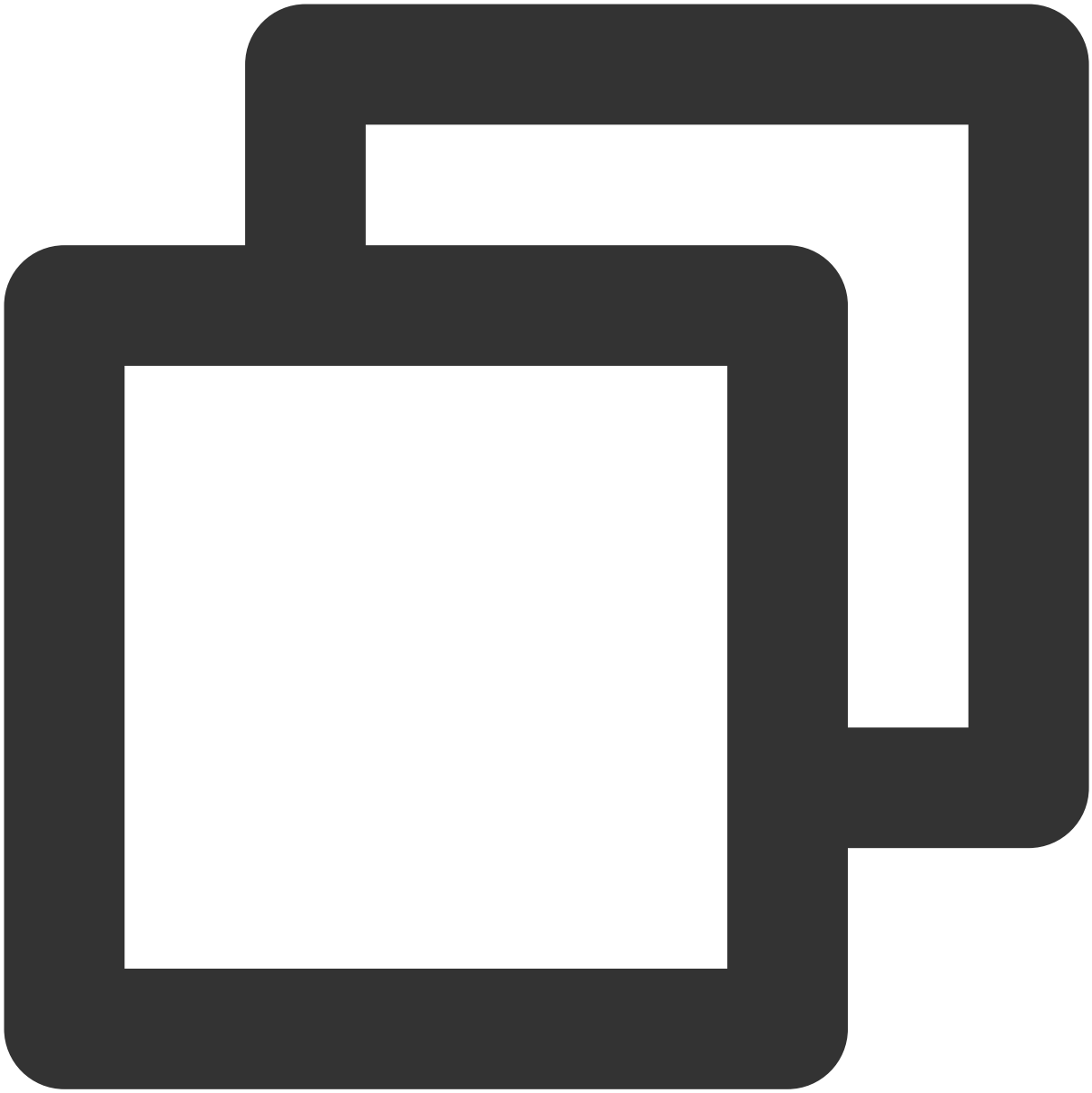
C



```
package main

import (
        "encoding/binary"
        "fmt"
        "net"
)

type NetworkConnection struct {
        Magic      uint16
        ClientAddr net.IP
```

```
        ProxyAddr  net.IP
        ClientPort uint16
        ProxyPort  uint16
}

func handleConn(conn *net.UDPConn) {
        buf := make([]byte, 1024)              // 创建缓冲区
        n, addr, err := conn.ReadFromUDP(buf) // 从连接读取数据包

        if err != nil {
                fmt.Println("Error reading from UDP connection:", err)
                return
        }

        // 将接收到的字节转换为NetworkConnection结构体
        nc := NetworkConnection{
                Magic:      binary.BigEndian.Uint16(buf[0:2]),
                ClientAddr: make(net.IP, net.IPv6len),
                ProxyAddr:  make(net.IP, net.IPv6len),
        }
        if nc.Magic == 0x56EC {
                copy(nc.ClientAddr, buf[2:18])
                copy(nc.ProxyAddr, buf[18:34])
                nc.ClientPort = binary.BigEndian.Uint16(buf[34:36])
                nc.ProxyPort = binary.BigEndian.Uint16(buf[36:38])

                // 打印 spp 头信息，包含 magic、客户端真实 ip 和 port、代理的 ip 和 port
                fmt.Printf("Received packet:\\n")
                fmt.Printf("\\tmagic: %x\\n", nc.Magic)
                fmt.Printf("\\tclient address: %s\\n", nc.ClientAddr.String())
                fmt.Printf("\\tproxy address: %s\\n", nc.ProxyAddr.String())
                fmt.Printf("\\tclient port: %d\\n", nc.ClientPort)
                fmt.Printf("\\tproxy port: %d\\n", nc.ProxyPort)
                // 打印真实有效的载荷
                fmt.Printf("\\tdata: %v\\n\\tcount: %v\\n", string(buf[38:n]), n)
        } else {
                // 打印真实有效的载荷
                fmt.Printf("\\tdata: %v\\n\\tcount: %v\\n", string(buf[0:n]), n)
        }

        // 回包，注意：需要将 SPP 38字节长度原封不动地返回
        response := make([]byte, n)
        copy(response, buf[0:n])
        _, err = conn.WriteToUDP(response, addr) // 发送数据
        if err != nil {
                fmt.Println("Write to udp failed, err: ", err)
        }
```

```
}

func main() {
        localAddr, _ := net.ResolveUDPAddr("udp", ":6666") // 使用本地地址和端口创建 UD
        conn, err := net.ListenUDP("udp", localAddr)        // 创建监听器
        if err != nil {
                panic("Failed to listen for UDP connections:" + err.Error())
        }

        defer conn.Close() // 结束时关闭连接
        for {
                handleConn(conn) // 处理传入的连接
        }
}
```



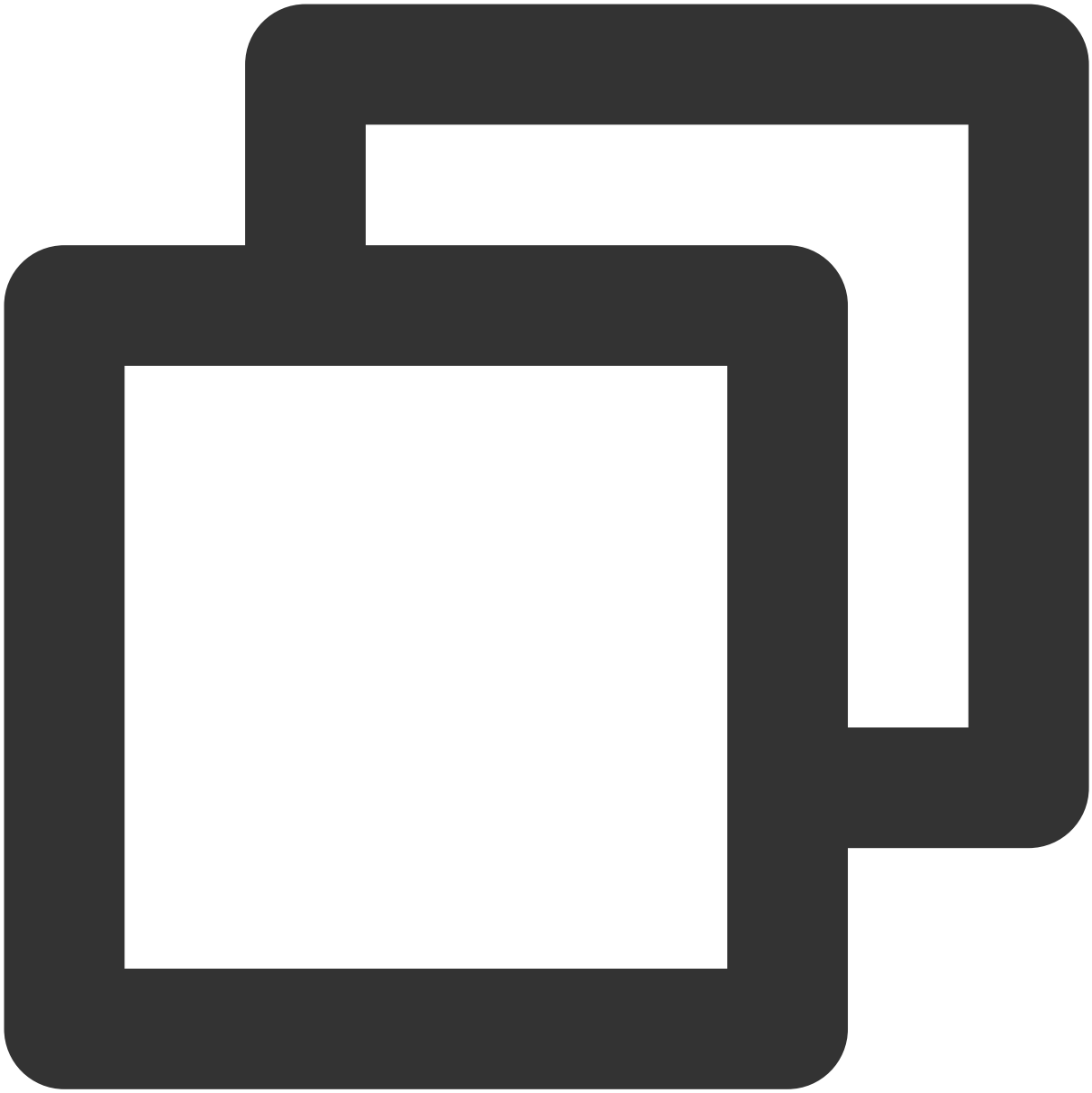
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/socket.h>
#define BUF_SIZE 1024
struct NetworkConnection {
    uint16_t magic;
    struct in6_addr clientAddr;
    struct in6_addr proxyAddr;
```

```
    uint16_t clientPort;
    uint16_t proxyPort;
};
void handleConn(int sockfd) {
    struct sockaddr_in clientAddr;
    socklen_t addrLen = sizeof(clientAddr);
    unsigned char buf[BUF_SIZE];
    ssize_t n = recvfrom(sockfd, buf, BUF_SIZE, 0, (struct sockaddr *)&clientAddr,
    if (n < 0) {
        perror("Error reading from UDP connection");
        return;
    }
    // 将接收到的字节转换为 NetworkConnection 结构体
    struct NetworkConnection nc;
    nc.magic = ntohs(*(uint16_t *)buf);
    if (nc.magic == 0x56EC) { // Magic 为 0x56EC 表示 SPP 头
        memcpy(&nc.clientAddr, buf + 2, 16);
        memcpy(&nc.proxyAddr, buf + 18, 16);
        nc.clientPort = ntohs(*(uint16_t *)(buf + 34));
        nc.proxyPort = ntohs(*(uint16_t *)(buf + 36));
        printf("Received packet:\\n");
        printf("\\tmagic: %x\\n", nc.magic);
        char clientIp[INET6_ADDRSTRLEN];
        char proxyIp[INET6_ADDRSTRLEN];
        inet_ntop(AF_INET6, &nc.clientAddr, clientIp, INET6_ADDRSTRLEN);
        inet_ntop(AF_INET6, &nc.proxyAddr, proxyIp, INET6_ADDRSTRLEN);

        // 打印 spp 头信息，包含 magic、客户端真实 ip 和 port、代理的 ip 和 port
        printf("\\tclient address: %s\\n", clientIp);
        printf("\\tproxy address: %s\\n", proxyIp);
        printf("\\tclient port: %d\\n", nc.clientPort);
        printf("\\tproxy port: %d\\n", nc.proxyPort);
        // 打印真实有效的载荷
        printf("\\tdata: %.*s\\n\\tcount: %zd\\n", (int)(n - 38), buf + 38, n);
    } else {
        printf("\\tdata: %.*s\\n\\tcount: %zd\\n", (int)n, buf, n);
    }
    // 回包，注意：需要将 SPP 38字节长度原封不动的返回
    sendto(sockfd, buf, n, 0, (struct sockaddr *)&clientAddr, addrLen);
}
int main() {
    int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd < 0) {
        perror("Failed to create socket");
        exit(EXIT_FAILURE);
    }
    // 使用本地地址和端口创建 UDP 地址
```

```
    struct sockaddr_in serverAddr;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_addr.s_addr = INADDR_ANY;
    serverAddr.sin_port = htons(6666);
    if (bind(sockfd, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) < 0) {
        perror("Failed to bind");
        exit(EXIT_FAILURE);
    }
    while (1) {
        handleConn(sockfd);
    }
}
```

步骤3：测试验证

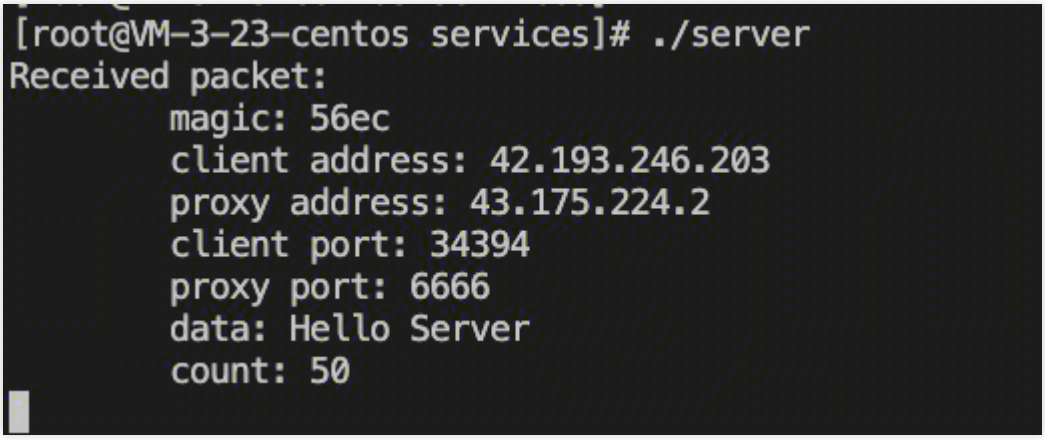
您可以找一台服务器充当客户端，构造客户端请求，以 nc 命令来模拟 UDP 请求，命令详情如下：



```
echo "Hello Server" | nc -w 1 -u <IP/DOMAIN> <PORT>
```

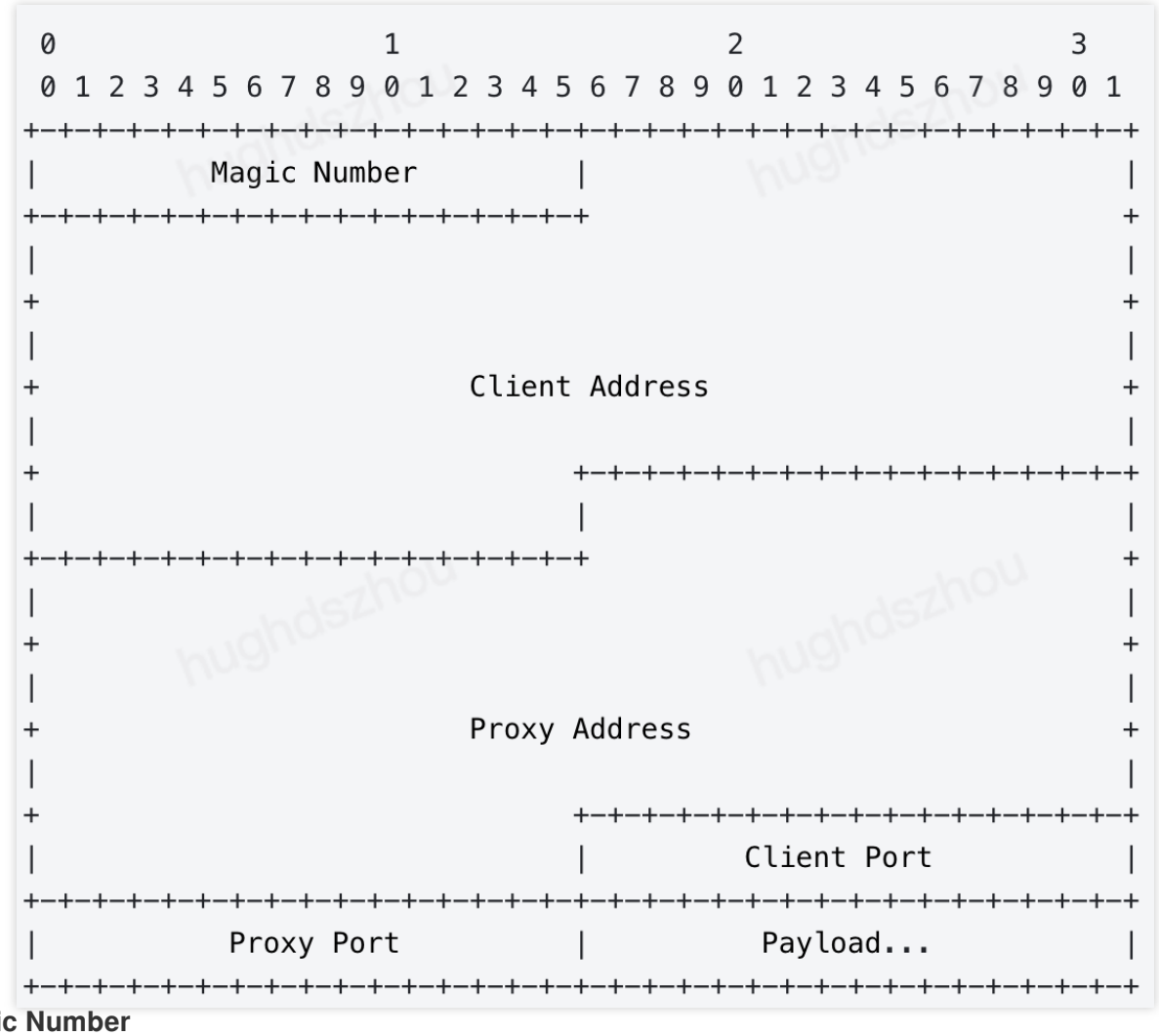
其中，IP/Domain 即为您的四层代理实例接入 IP 或者域名，您可以在 EdgeOne 控制台内查看对应的四层代理实例信息。Port 即为您在 步骤1 内为该规则所配置的转发端口。

服务端收到请求并解析客户端 IP 地址如下：



相关参考

SPP 协议头格式



Magic Number

在 SPP 协议格式中，Magic Number 为 16 位，且固定值为 0x56EC，主要用于识别 SPP 协议，并定义了 SPP 协议头是固定 38 字节长度。

Client Address

客户端发起请求的 IP 地址，长度为 128 位，如果是 IPV4 客户端发起，则该值表示 IPV4；如果是 IPV6 客户端发起，则该值表示 IPV6。

Proxy Address

代理服务器的 IP 地址，长度为 128 位，可以和 Client Address 相同的解析方式。

Client Port

客户端发送 UDP 数据包的端口，长度为 16 位。

Proxy Port

代理服务器接收 UDP 数据包的端口，长度为 16 位。

payload

有效载荷，数据包携带的标头后面的数据。