**北京邮电大学课程设计报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **课程设计**  **名称** | **计算机网络课程设计** | | **学 院** | **计算机学院** | **指导教师** | **张雪松** |
| **班 级** | **班内序号** | **学 号** | | **学生姓名** | **成绩** | |
| **2022211311** | **25** | **2022211388** | | **陈韵涵** |  | |
| **2022211311** | **24** | **2022211387** | | **张卓依** |  | |
| **2022211311** | **20** | **2022211383** | | **田雨珩** |  | |
| **课**  **程**  **设**  **计**  **内**  **容** | **基本功能：**   * 设计了一个 DNS 服务器程序，读入“IP 地址-域名”对照表，当客户端查询域 名对应的 IP 地址时，用域名检索该对照表，有三种可能检索结果：   + 检索结果：IP 地址 0.0.0.0，则向客户端返回“域名不存在”的报错消息（不良网站拦截功能）     - 管理员用户一开始时将需要拦截的网站IP设置为“0.0.0.0”，储存在本地的域名-IP表中。在从本地表读出数据，且用户访问的域名对应IP为“0.0.0.0”时，dns 返回报文NXDOMAIN。   + 检索结果：普通 IP 地址，则向客户端返回该地址（服务器功能）     - 首先，DNS服务器通过读取cache查看是否在cache中存在此域名记录，若存在，则直接返回此域名对应的IP地址；否则继续在本地的 dnsrelay.dic 文件中查找是否有对应的记录，如果本地数据有记录是，服务器将直接将目的地IP地址提供给客户端，从而为用户提供有效的域名解析服务。   + 表中未检到该域名，则向因特网 DNS 服务器发出查询，并将结果返给客户端（中继功能）     - 考虑多个计算机上的客户端会同时查询，需要进行消息 ID 的转换。     - 考虑一个域名对应多个 IP。 * 多客户端并发：允许多个客户端（可能会位于不同的多个计算机）的并发查询，即：允许第一个查询尚未得到答案前就启动处理另外一个客户端查询请求。   **额外功能：**   * 输出调试信息和DNS报文内容。-d表示一级debug，-dd表示二级debug。 * “消费者/生产者”模型：采用多线程方式进行网络通信，提高中继服务器的处理速度。 * 用字典树存储本地IP-域名对照表，用双向链表实现了cache的存储，并且使用LRU算法对cache进行更新。 * 命令行参数解析。   **分工：**   * 陈韵涵：主函数、程序初始化、多线程相关、cache实现、字典树实现。 * 张卓依：DNS包解析、构建DNS包、处理客户端传来的包。 * 田雨珩：ID转换、处理从上游服务器传来的包。 | | | | | |
| **学生**  **课程设计**  **报告**  （附页） |  | | | | | |
| **课**  **程**  **设**  **计**  **成**  **绩**  **评**  **定** | **评语**:  **成绩**:  指导教师签名：  年 月 日 | | | | | |

注：评语要体现每个学生的工作情况，可以加页。

# DNS Relay实验报告

# 1 概述

## 1.1 程序功能简介

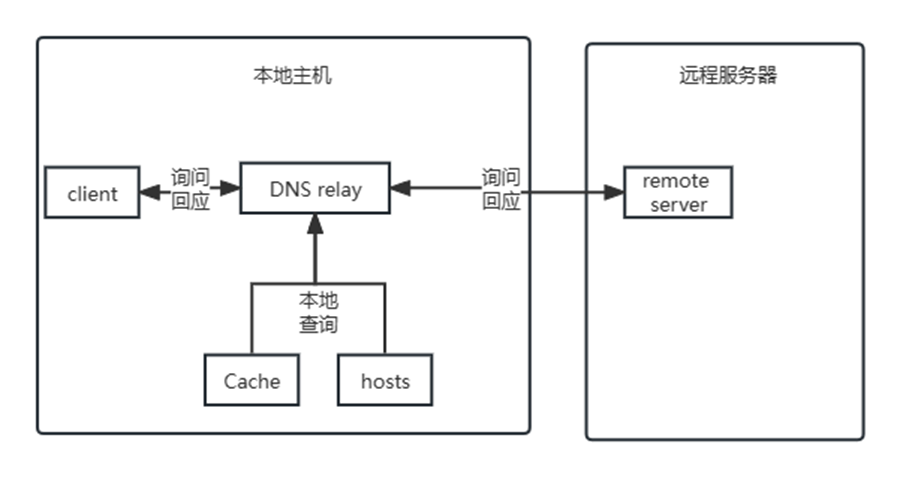
**基本功能：**

* 设计了一个 DNS 服务器程序，读入“IP 地址-域名”对照表，当客户端查询域 名对应的 IP 地址时，用域名检索该对照表，有三种可能检索结果：
  1. 检索结果：IP 地址 0.0.0.0，则向客户端返回“域名不存在”的报错消息（不良网站拦截功能）
     1. 管理员用户一开始时将需要拦截的网站IP设置为“0.0.0.0”，储存在本地的域名-IP表中。在从本地表读出数据，且用户访问的域名对应IP为“0.0.0.0”时，dns 返回报文NXDOMAIN。
  2. 检索结果：普通 IP 地址，则向客户端返回该地址（服务器功能）
     1. 首先，DNS服务器通过读取cache查看是否在cache中存在此域名记录，若存在，则直接返回此域名对应的IP地址；否则继续在本地的 dnsrelay.dic 文件中查找是否有对应的记录，如果本地数据有记录是，服务器将直接将目的地IP地址提供给客户端，从而为用户提供有效的域名解析服务。
  3. 表中未检到该域名，则向因特网 DNS 服务器发出查询，并将结果返给客户端（中继功能）
     1. 考虑多个计算机上的客户端会同时查询，需要进行消息 ID 的转换。
     2. 考虑一个域名对应多个 IP。
* 多客户端并发：允许多个客户端（可能会位于不同的多个计算机）的并发查询，即：允许第一个查询尚未得到答案前就启动处理另外一个客户端查询请求。

**额外功能：**

* 输出调试信息和DNS报文内容。-d表示一级debug，-dd表示二级debug。
* “消费者/生产者”模型：采用多线程方式进行网络通信，提高中继服务器的处理速度。
* 用字典树存储本地IP-域名对照表，用双向链表实现了cache的存储，并且使用LRU算法对cache进行更新。
* 命令行参数解析。

结合以上功能设计和需求分析，可以构建出以下的功能设计图：



## 1.2 程序模块划分

include

├── dns.h DNS包的数据结构

├── config.h 程序配置的数据结构及函数声明

├── handler.h 处理DNS包的数据结构及函数声明

├── relaylist.dic IP-域名对照表的数据结构及函数声明

├── resource.h 线程共享资源的数据结构及函数声明

├── thread.h 线程的数据结构及函数声明

└── trie.h 字典树的数据结构及函数声明

src

├── main.c 主函数，初始化程序

├── config.c 程序配置

├── handler.c 处理DNS包

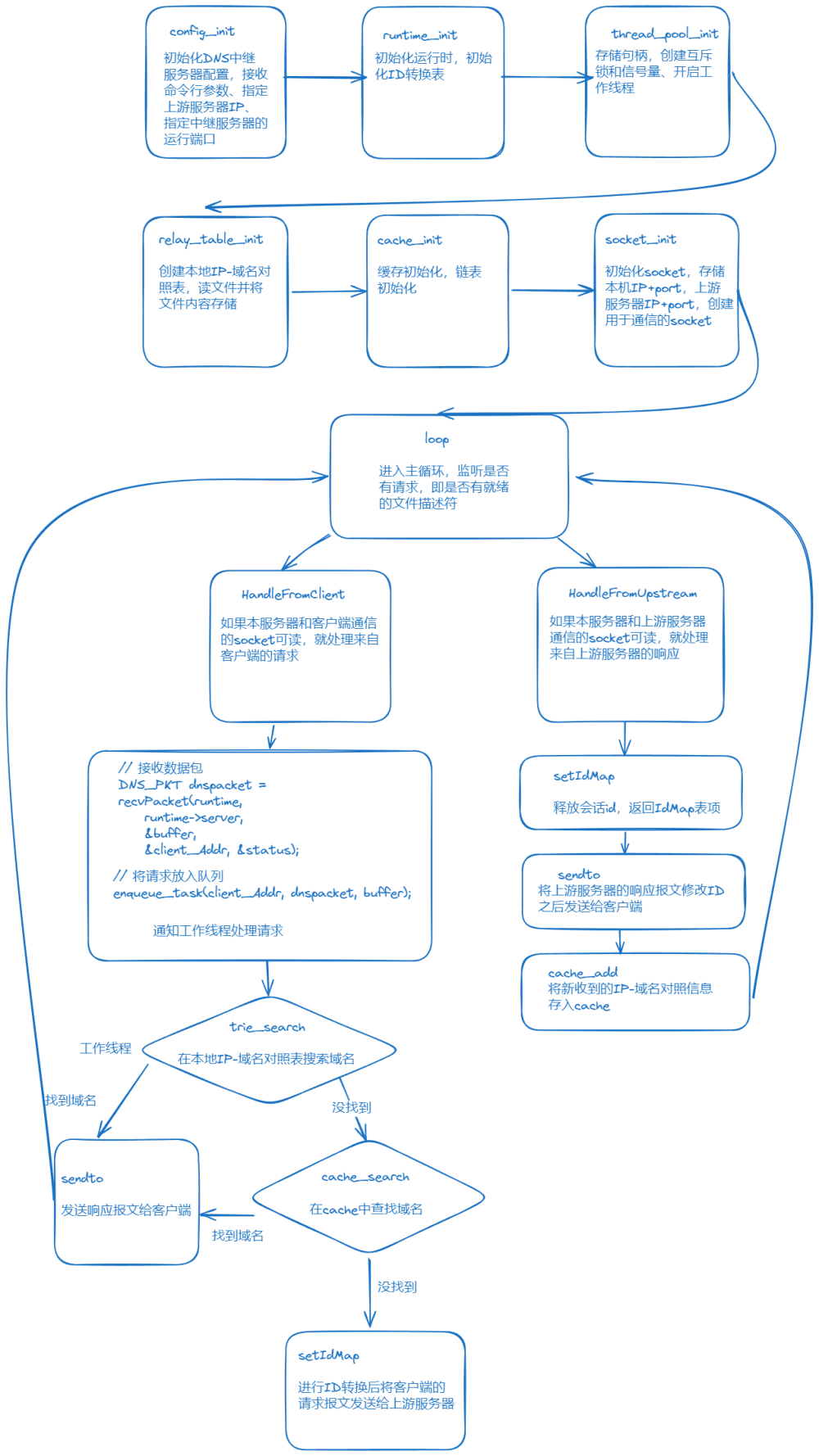
├── relaylist.dic 域名-ip对照表

├── resource.c 线程共享资源

├── thread.c 线程创建与管理

└── trie.c 字典树的初始化与管理

## 1.3 软件流程图



## 1.4 开发环境

* Windows11
* Visual Studio Code
* C语言

# 2 DNS理论基础

## 2.1 DNS 简介

DNS（Domain Name System，域名系统）是互联网上的一种服务，用于将域名翻译为与之对应的 IP 地址。DNS 是一个分布式的命名系统，它允许用户通过更易记的域名来访问互联网资源，而不必记住每个资源的复杂数字地址。

## 2.2 DNS 基本功能和原理

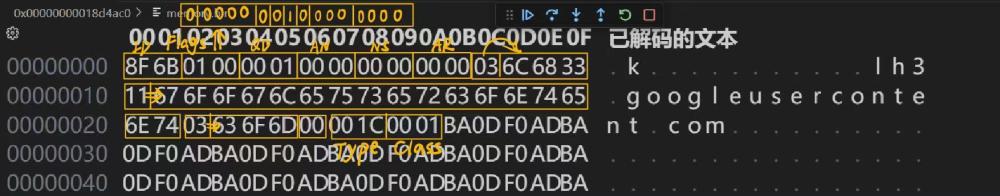
1. 域名解析：DNS 的主要功能是将人类可读的域名翻译成计算机可识别的 IP 地址。这个过程通常包括多个步骤，涉及到多个DNS服务器之间的协作。
2. 层次结构：DNS 使用层次结构来管理域名空间。域名从右向左依次增加层级，例如 www.example.com 中 com 是顶级域名，example 是二级域名，www 是主机名。
3. DNS服务器：DNS 服务由多种类型的服务器组成，包括根域名服务器、顶级域名服务器（如 .com, .net）、授权域名服务器和本地域名服务器（如 ISP 提供的 DNS 服务器或家庭路由器上的 DNS 缓存）。
4. DNS查询过程：
   1. 当用户输入一个域名时，操作系统的 DNS 客户端首先会向本地 DNS 服务器发起查询。
   2. 如果本地服务器未能找到域名的 IP 地址，它会向根域名服务器发出请求。
   3. 根服务器会指导本地服务器向负责顶级域名的服务器（如 .com 的服务器）发出进一步查询。
   4. 最终，负责该域名的授权服务器会返回域名对应的 IP 地址。
5. DNS缓存：为了加速查询过程，DNS 使用了缓存机制，本地服务器会将查询结果缓存一段时间，以便下次查询相同域名时能够直接返回结果，而无需再次向上级服务器查询。
6. 其他功能：除了将域名解析为 IP 地址外，DNS 还可以管理邮件服务器的 MX 记录、验证证书的 TLSA 记录、提供身份验证的 TXT 记录等

## 2.3 DNS报文格式

DNS报文具有统一的格式，均由12字节的首部和4个长度不固定的字段组成：



**程序调试过程中收到的请求报文**



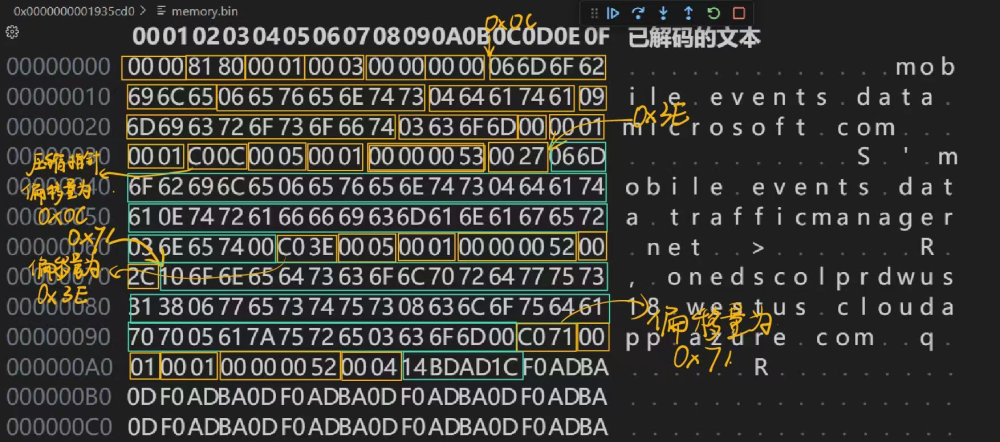
ID：0x8F6B

flag: RD:1 RA:1

问题数：1

回答资源数：0

**程序调试过程中收到的应答报文**



ID：0000

flag: QR:1 RD:1 RA:1

问题数：1

回答资源数：3

### 2.3.1头部：

* 事务ID(2字节)：DNS的会话标识。对于同一个会话的请求报文和应答报文使用同一个ID，可以通过ID号来区分DNS应答报文是哪一个请求的响应。
* 标志（2字节）：

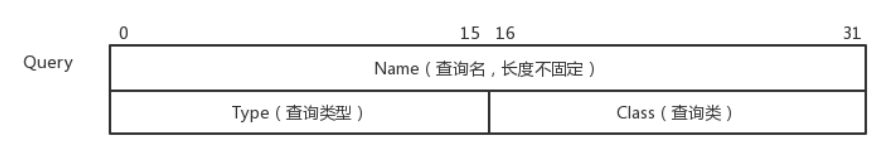


|  |  |
| --- | --- |
| QR（1bit） | 查询/响应的标志信息，查询请求报文该字段为0，响应报文为1 |
| opcode（4bit） | 操作码（0表示标准查询，1表示反向查询，2表示服务器状态请求） |
| AA（1bit） | 表示授权回答，在响应报文中有效（为1表示服务器是权威服务器） |
| TC（1bit） | 表示是否被截断（为1表示响应已经超过512字节，并且已被截断） |
| RD（1bit） | 表示期望递归，在查询报文中设置，在响应中返回。（为1时，使用递归查询，为0时使用迭代查询） |
| RA（1bit） | 表示可用递归，在响应报文中设置（为1时，代表服务器支持递归查询） |
| Z （3bit） | 保留字段，在所有的请求和应答报文中，它的值必须为 0 |
| rcode（4bit） | 表示返回码（0表示没有差错，1表示报文格式错误，2表示域名服务器错误，3表示名字错误，4表示查询类型不支持，5表示拒绝） |

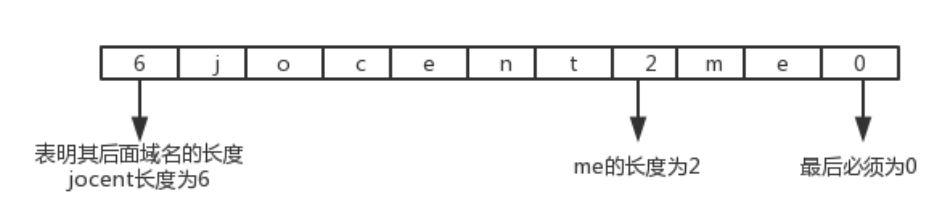
* 问题数（2字节）：DNS 查询请求的数目。
* 回答资源记录数（2字节）：DNS 响应的数目。
* 权威名称服务器计数（2字节）：权威名称服务器的数目。
* 附加资源记录数（2字节）：额外的记录数目（权威名称服务器对应 IP 地址的数目）。

### 2.3.2正文：

#### 问题字段：



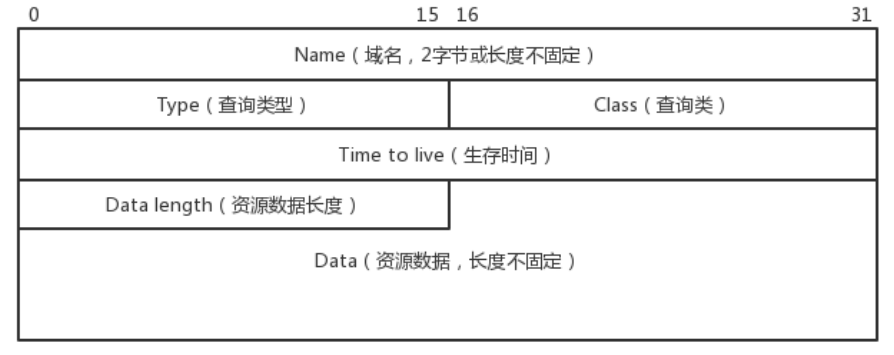
* **查询名（**长度不固定**）：**一般该字段表示的就是需要查询的域名（如果是反向查询，则为IP，反向查询即由IP地址反查域名），一般的格式如下图所示：



* **查询类型（2字节）：**DNS 查询请求的资源类型。通常查询类型为 A 类型，表示由域名获取对应的 IPv4 地址。
* **查询类（2字节）：**地址类型，通常为互联网地址，值为 1。

#### 资源记录字段：

资源记录字段包含回答、授权和附加三个区域。资源记录字段的格式如下图所示：



* 域名（2字节或者长度不固定）：当该域名是第一次出现在该报文中时，其存储方式和问题字段的域名存储方式相同。但若该域名在此报文中重复出现时，为节省DNS报文的长度，该字段使用压缩指针的方式进行存储。即选择存储一个指针的偏移量（通过判断该字段前两位是不是11，从而判断该字段选择压缩指针存储或者普通方式存储），使得指向该报文头部的指针经过偏移之后指向存储该域名的真实位置。
* 查询类型（2字节）：表明资源纪录的类型，与问题部分的查询类型相同。
* 查询类（2字节）：地址类型，与问题部分查询类相同。
* 生存时间（4字节）：以秒为单位，表示资源记录的生命周期，当生存时间耗尽时资源记录失效。
* 资源数据（长度不固定）：按照查询段要求返回的信息。

# 3 程序配置和运行时

## 3.1 数据结构

/\* DNS服务器的配置 \*/  
typedef struct DNS\_CONFIG  
{  
 boolean debug; // 是否输出debug信息  
 boolean debug\_2; // 二级调试信息  
 int port; // 监听端口号  
 char upstream\_server\_IP[16]; // 上游DNS服务器的IP  
} DNS\_CONFIG;  
  
/\* 程序运行时 \*/  
typedef struct DNS\_RUNTIME  
{  
 DNS\_CONFIG config; // 服务器配置  
 boolean quit; // 程序是否退出  
 SOCKET server; // 与客户端通信的socket  
 SOCKET client; // 与上游服务器通信的socket  
 IdMap \*idmap; // IdMap数组，用于会话id管理  
 uint16\_t maxId; // 上一次向上游服务器发送查询请求时所使用的ID号  
 struct sockaddr\_in listen\_addr; // 监听地址  
 struct sockaddr\_in upstream\_addr; // 上级DNS服务器地址  
} DNS\_RUNTIME;

## 3.2 初始化

/\*\*  
 \* 初始化DNS服务器配置  
 \*/  
DNS\_CONFIG config\_init(int argc, char \*argv[])  
{  
 DNS\_CONFIG config;  
 config.debug = FALSE;  
 config.port = 53;  
 // 默认上游服务器IP  
 strcpy(config.upstream\_server\_IP, "10.3.9.5");  
  
 for (int i = 0; i < argc; i++)  
 {  
 if (strcmp("-u", argv[i]) == 0)  
 {  
 // 设定上游服务器IP  
 strcpy(config.upstream\_server\_IP, argv[i + 1]);  
 }  
 else if (strcmp("-d", argv[i]) == 0)  
 {  
 // 开启debug模式  
 config.debug = TRUE;  
 }  
 else if (strcmp("-dd", argv[i]) == 0)  
 {  
 // 开启二级debug模式  
 config.debug = TRUE;  
 config.debug\_2 = TRUE;  
 }  
 }  
 return config;  
}  
  
/\*\*  
 \* 运行时初始化  
 \*/  
DNS\_RUNTIME runtime\_init(DNS\_CONFIG \*config)  
{  
 DNS\_RUNTIME runtime;  
 runtime.config = \*config;  
 runtime.quit = FALSE;  
 runtime.idmap = initIdMap();  
 runtime.maxId = 0;  
 return runtime;  
}

# 4 DNS报文的解析和构造

## 4.1 DNS报文和缓冲区的数据结构

typedef struct DNS\_PKT  
{  
 DNS\_HEADER \*header;  
 DNS\_QUESTION \*question;  
 DNS\_RECORD \*answer;  
 DNS\_RECORD \*authority;  
 DNS\_RECORD \*additional;  
} DNS\_PKT;  
  
/\* Header Section Format \*/  
typedef struct DNS\_HEADER  
{  
 uint16\_t ID; // 会话标识  
 uint8\_t RD : 1; // 表示期望递归  
 uint8\_t TC : 1; // 表示可截断的  
 uint8\_t AA : 1; // 表示授权回答  
 uint8\_t Opcode : 4; // 0表示标准查询，1表示反向查询，2表示服务器状态请求  
 uint8\_t QR : 1; // 查询、响应标识，0为查询，1为响应  
 uint8\_t Rcode : 4; // 应答码  
 uint8\_t Z : 3; // 保留值。在所有请求和应答报文中置为零  
 uint8\_t RA : 1; // 表示可用递归  
 uint16\_t QDCOUNT; // 问题数  
 uint16\_t ANCOUNT; // 回答资源记录数  
 uint16\_t NSCOUNT; // 授权资源记录数  
 uint16\_t ARCOUNT; // 附加资源记录数  
} DNS\_HEADER;  
  
/\* 查询区段 \*/  
typedef struct DNS\_QUESTION  
{  
 char name[NAME\_LEN]; // 域名  
 uint16\_t Qtype; // DNS请求的资源类型  
 uint16\_t Qclass; // DNS查询的地址类型，如IN  
} DNS\_QUESTION;  
  
/\* 资源记录 \*/  
typedef struct DNS\_RECORD  
{  
 char name[NAME\_LEN]; // 域名  
 uint16\_t type; // 资源记录的类型  
 uint16\_t addr\_class; // 资源记录的地址类型  
 uint32\_t TTL; // 有效时间  
 uint16\_t rdlength; // rdata的长度  
 char \*rdata; // 指向资源数据的指针  
} DNS\_RECORD;  
  
typedef struct Buffer  
{  
 uint8\_t \*data; // buffer首地址  
 uint32\_t length; // buffer长度  
} Buffer;

## 4.2 Decode（实现从字节流DNS数据包到DNS结构体的映射）

### 需求

程序通过recvfrom(SOCKET s, char \*buf, int len, int flags, struct sockaddr \*from, int \*fromlen)函数接收DNS包，并将DNS报存进缓冲区buf中。此时buf中存储的为网络上的字节流DNS协议报文，我们需要对其进行解析，将其转换为内存中的DNS数据结构,便于应用程序可以方便地访问和操作DNS信息。

### 实现思路

通过声明一个指针指向接收到的DNS数据报头部，根据各个字段分别占用的字节数来进行读取和指针的移动。

#### 定长字段

由于每次读取数据的长度和指针移动都固定，我们定义了一系列函数来分别进行单字节、双字节和四字节字段内容的读取和指针的移动，进行读取和指针移动的函数如下：（对于flag字段，我们先读取整个flag字段，再对读取字段进行解析）

uint8\_t \*\_read8(uint8\_t \*ptr, uint8\_t \*value)  
{  
 \*value = \*(uint8\_t \*)ptr;  
 return ptr + 1;  
}  
uint8\_t \*\_read16(uint8\_t \*ptr, uint16\_t \*value)  
{  
 \*value = ntohs(\*(uint16\_t \*)ptr);  
 return ptr + 2;  
}  
uint8\_t \*\_read32(uint8\_t \*ptr, uint32\_t \*value)  
{  
 \*value = ntohl(\*(uint32\_t \*)ptr);  
 return ptr + 4;  
}

#### 非定长字段

由于域名长度不固定、存在形式的转换且可能出现域名复用的情况，显然无法使用上述方法进行读取解析。因此我们定义了函数int toDot(char \*ptr, char \*start, char \*newStr)来进行解析。

##### 函数参数

* ptr：当前指针指向位置，即网络数据包中域名字段的起始位置。
* start：网络数据包的首地址，用于计算压缩指针的绝对偏移位置。
* newStr：用于存储解析后的域名字符串，函数将解析结果写入该数组

##### 初始化操作

* newStr[0] = '\0';：初始化 newStr 为空字符串，用于存储最终的解析结果。

##### 处理压缩指针

* if ((unsigned char)ptr[0] >= 0xC0)：判断当前域名字段是否是压缩指针形式。如果 ptr[0] 的最高两位为 11，则表示这是一个压缩指针。
* int offset = ((unsigned char)ptr[0] << 8 | (unsigned char)ptr[1]) & 0xfff ：计算出压缩指针的偏移量，通过将 ptr[0] 和 ptr[1] 的内容合并取低12位得到（将最高位的11去除）。0xfff 是一个掩码，用于确保偏移量在合理范围内。
* toDot(start + offset, start, newStr) ：递归调用 toDot 函数，以解析压缩指针指向的实际域名。
* 由于压缩指针的大小固定为2字节，所以返回值为2。

##### 解析普通域名

* 如果不是压缩指针形式，则进入解析普通域名的分支。
* 记录域名总长度len，初始化已经处理的域名的长度i，当i<len时需要重复执行下方操作。
* 根据2.3.2中提到的域名在字节流中存储的格式，我们使用bias记录每次要复制的字符长度以控制每段域名的长度，将当前端字符复制到newStr中。
* 复制完毕后更新bias的取值，将'.'加入nreStr中以形成点分隔的域名格式。
* 最后整个域名解析完毕，跳出循环，在newStr的末尾加上字符串结束符，并且返回读取数据的指针的偏移量，即当前函数处理的域名字段长度加 1，用于更新调用者的指针位置。

##### 代码实现

int toDot(char \*ptr, char \*start, char \*newStr) // 传入当前指针指向位置和buffer中data的首地址，newStr指向返回的域名  
{  
 newStr[0] = '\0';  
 if ((unsigned char)ptr[0] >= 0xC0)  
 { // 前两位是11，说明该域名字段是压缩指针的形式  
 int offset = ((unsigned char)ptr[0] << 8 | (unsigned char)ptr[1]) & 0xfff; // 压缩指针偏移量  
 toDot(start + offset, start, newStr); // 递归查询域名  
 return 2; // 压缩指针占两字节  
 }  
 else // 解析普通域名  
 {  
 int len = strlen(ptr); // 计算该域名的长度  
 int idx = 0; // 点分形式域名字符下标  
 int bias = ptr[0]; // 决定接下来复制字符数量  
 int i = 1; // 当前处理字节数  
 while (i < len)  
 {  
 for (int j = 0; j < bias; j++)  
 {  
 newStr[idx] = ptr[i];  
 idx++;  
 i++;  
 }  
 bias = ptr[i]; // 计算该域名下一段的长度  
 if (bias == 0 || i >= len)  
 break;  
 i++;  
 newStr[idx] = '.';  
 idx++;  
 }  
 newStr[idx] = '\0'; // 标志字符串结束  
 return len + 1; // 返回读取数据的指针的偏移量  
 }  
}

### 代码实现

void DNSPacket\_decode(Buffer \*buffer, DNS\_PKT \*packet)  
{  
 uint8\_t \*Rdata\_ptr = buffer->data;  
 uint8\_t tmp8;  
 // Transaction ID  
 Rdata\_ptr = \_read16(Rdata\_ptr, &packet->header->ID); // 一次读16位  
 // QR+OP+AA+TC+RD  
 Rdata\_ptr = \_read8(Rdata\_ptr, &tmp8); // 一次读8位  
 packet->header->QR = (DNSPacketQR)(tmp8 >> 7 & 0x01);  
 packet->header->Opcode = (DNSPacketOP)(tmp8 >> 3 & 0x0F);  
 packet->header->AA = tmp8 >> 2 & 0x01;  
 packet->header->TC = tmp8 >> 1 & 0x01;  
 packet->header->RD = tmp8 >> 0 & 0x01;  
 // RA+padding(3)+RCODE  
 Rdata\_ptr = \_read8(Rdata\_ptr, &tmp8);  
 packet->header->RA = tmp8 >> 7 & 0x01;  
 packet->header->Rcode = (DNSPacketRC)(tmp8 & 0xF);  
 // Counts  
 Rdata\_ptr = \_read16(Rdata\_ptr, &packet->header->QDCOUNT);  
 Rdata\_ptr = \_read16(Rdata\_ptr, &packet->header->ANCOUNT);  
 Rdata\_ptr = \_read16(Rdata\_ptr, &packet->header->NSCOUNT);  
 Rdata\_ptr = \_read16(Rdata\_ptr, &packet->header->ARCOUNT);  
 if (Rdata\_ptr > buffer->data + buffer->length)  
 {  
 buffer->length = 0;  
 return packet;  
 }  
 // Questions  
 if (packet->header->QDCOUNT > 0)  
 {  
 packet->question = (DNS\_QUESTION \*)malloc(sizeof(DNS\_QUESTION) \* packet->header->QDCOUNT);  
 if (packet->header->QDCOUNT != 1) // 问题数量大于1  
 {  
 buffer->length = 0;  
 \*packet = init\_DNSpacket();  
 return;  
 }  
 packet->question[0].name[0] = (char \*)malloc((strlen(Rdata\_ptr)) \* sizeof(char));  
 Rdata\_ptr += toDot((char \*)Rdata\_ptr, (char \*)buffer->data, packet->question[0].name);  
 packet->question[0].Qtype = (uint16\_t)(Rdata\_ptr[0] << 8) + Rdata\_ptr[1];  
 packet->question[0].Qclass = (uint16\_t)(Rdata\_ptr[2] << 8) + Rdata\_ptr[3];  
 Rdata\_ptr += 4;  
 }  
 // Answers  
 if (packet->header->ANCOUNT > 0)  
 {  
 packet->answer = (DNS\_RECORD \*)malloc(sizeof(DNS\_RECORD) \* packet->header->ANCOUNT); // 根据头部记录answer的数量来malloc指定空间  
 // Name  
 for (int i = 0; i < packet->header->ANCOUNT; i++)  
 {  
  
 Rdata\_ptr += toDot((char \*)Rdata\_ptr, (char \*)buffer->data, packet->answer[i].name);  
 // Type  
 uint16\_t tmp;  
 Rdata\_ptr = \_read16(Rdata\_ptr, &tmp);  
 packet->answer[i].type = (DNSQType)tmp;  
 // Class  
 Rdata\_ptr = \_read16(Rdata\_ptr, &tmp);  
 packet->answer[i].addr\_class = (uint16\_t)tmp;  
 // Time to live  
 Rdata\_ptr = \_read32(Rdata\_ptr, &packet->answer[i].TTL);  
 // Data length  
 Rdata\_ptr = \_read16(Rdata\_ptr, &packet->answer[i].rdlength);  
 // data  
 packet->answer[i].rdata = (char \*)malloc(sizeof(char) \* packet->answer[i].rdlength);  
 memcpy(packet->answer[i].rdata, Rdata\_ptr, packet->answer[i].rdlength);  
 Rdata\_ptr += packet->answer[i].rdlength;  
  
 if (Rdata\_ptr > buffer->data + buffer->length + 1) // 指针越界  
 {  
 buffer->length = 0;  
 \*packet = init\_DNSpacket();  
 return; // 返回一个空包  
 }  
 }  
 }  
 else  
 {  
 packet->answer = NULL;  
 }  
 // Authority和Additional与answer字段类型省略（在完整代码中给出）  
   
}

## 4.3 Encode（实现从DNS结构体到字节流DNS数据包的映射）

### 4.3.1 需求

当程序对DNS结构体进行修改或者构建另一个新的DNS包发送给客户端或者上游服务器时，需要将DNS结构体再次转化为字节流DNS数据包的格式，通过sendto(SOCKET s, const char \*buf, int len, int flags, const struct sockaddr \*to, int tolen)函数将包通过指定端口发送到目标地址。

### 4.3.2 实现思路

通过声明一个指针指向声明的DNS数据报头部，根据各个字段分别占用的字节数来进行写入和指针的移动。

#### 定长字段

由于每次写入数据的长度和指针移动都固定，我们定义了一系列函数来分别进行单字节、双字节和四字节字段内容的写入和指针的移动，进行写入和指针移动的函数如下：

uint8\_t \_write8(uint8\_t \*ptr, uint8\_t value)  
{  
 \*(uint8\_t \*)ptr = value;  
 return 1;  
}  
uint8\_t \_write16(uint8\_t \*ptr, uint16\_t value)  
{  
 \*(uint16\_t \*)ptr = htons(value);  
 return 2;  
}  
uint8\_t \_write32(uint8\_t \*ptr, uint32\_t value)  
{  
 \*(uint32\_t \*)ptr = htonl(value);  
 return 4;  
}

#### 非定长字段

由于域名长度不固定、存在形式的转换且可能出现域名复用的情况，显然无法使用上述方法进行写入解析。对于重复域名，为节省DNS字节流的长度，采用压缩指针的方式进行存储。首先我们定义了一个函数来判断当前域名是否在报文中已经出现过，并且返回指针的偏移量（uint16\_t类型）。如果当前域名是重复的则将这个偏移量的前两位置为1，并将其写入待发送数据包中。否则，就作为普通域名进行处理。

#define DNS\_COMPRESSION\_POINTER(offset) (0xC000 | (offset))  
uint16\_t new\_offset = isFind\_repeatDomain((char \*)packet.question, (char \*)packet.answer[i].name, (char \*)question\_ptr, (char \*)buffer.data);  
if (offset >= 0)  
{  
 int16\_t compressed\_pointer = DNS\_COMPRESSION\_POINTER(new\_offset);  
 offset = \_write16(data, compressed\_pointer);  
}  
else  
{  
 offset = toQname(packet.answer[i].name, (char \*)data);  
}

判断域名是否重复的函数 isFind\_repeatDomain（）将指向当前域名、问题字段的域名以及报文头的指针作为参数传入，如果重复则返回与当前域名相同的域名在报文中的位置（即压缩指针的偏移量），不重复则返回-1，供调用者进行判断。

int isFind\_repeatDomain(char \*question\_name, char \*answer\_name, char \*question, char \*start)  
{  
 if (strcmp(answer\_name, question\_name) == 0)  
 {  
 return question - start;  
 }  
 else  
 return -1;  
}

对于普通域名，我们定义了函数int toQname(char \*name, char \*data)来进行解析。

##### 函数参数

* name：待解析域名字符串，例如 "[www.example.com](http://www.example.com/)"。
* data：用于存储转换后的字节流格式数据的缓冲区。

##### 初始化操作

* int i, j = 0, length = 0;定义循环变量 i，以及用于追踪当前段长度的和写入位置的length和j。

##### 域名处理循环

* for (i = 0; i < strlen(name); i++) 遍历带解析的域名字符串。

##### 处理域名中的每个字符

* if (name[i] == '.') 如果当前字符是 '.'，表示一个域名段的结束。
  1. data[j] = length; 将当前段的长度写入当前域名段的起始位置。
  2. length = 0; 重置 length 以开始下一个域名段的计数。
  3. j = i + 1; 更新 j 到下一个段的起始位置。
* else：处理非 '.' 的字符。
  1. if (name[i] >= 'A' && name[i] <= 'Z') 如果字符是大写字母，则将其转换为小写字母。
  2. length++; 递增当前段的长度计数。
  3. data[i + 1] = name[i]; 将字符写入 data 数组中，索引为 i + 1。

##### 处理最后一个段和结束标志

* data[j] = length; 将域名最后一个段的长度写入该段的起始位置。
* data[i + 1] = 0; 在 data 数组的最后一个位置写入结束标志 0，表示域名结束。

##### 返回值

* return strlen(name) + 2;返回处理后的字节流格式数据的总长度。这里加 2 是因为域名结束标志和指示当前域名段的长度占用了额外的空间。

##### 代码实现

uint8\_t toQname(char \*name, char \*data)  
{  
 int i, j = 0, length = 0;  
 for (i = 0; i < strlen(name); i++)  
 {  
 if (name[i] == '.')  
 {  
 data[j] = length;  
 length = 0;  
 j = i + 1;  
 }  
 else  
 {  
 if (name[i] >= 'A' && name[i] <= 'Z') // 将域名中大写转小写  
 name[i] = name[i] + 'a' - 'A';  
 length++;  
 data[i + 1] = name[i];  
 }  
 }  
 data[j] = length;  
 data[i + 1] = 0;  
 return strlen(name) + 2;  
}

### 4.3.3 代码实现

Buffer DNSPacket\_encode(DNS\_PKT packet)  
{  
 Buffer buffer = makeBuffer(DNS\_PACKET\_SIZE);  
 uint8\_t \*data = buffer.data;  
 uint8\_t offset = 0;  
 // Header  
 offset = \_write16(data, packet.header->ID);  
 data += offset;  
 // QR+OP+AA+TC+RD  
 offset = \_write8(data, packet.header->QR << 7 |  
 packet.header->Opcode << 3 |  
 packet.header->AA << 2 |  
 packet.header->TC << 1 |  
 packet.header->RD << 0);  
 data += offset;  
 // RA+padding(3)+RCODE  
 offset = \_write8(data, packet.header->RA << 7 | packet.header->Rcode);  
 data += offset;  
 // Counts  
 offset = \_write16(data, packet.header->QDCOUNT);  
 data += offset;  
 offset = \_write16(data, packet.header->ANCOUNT);  
 data += offset;  
 offset = \_write16(data, packet.header->NSCOUNT);  
 data += offset;  
 offset = \_write16(data, packet.header->ARCOUNT);  
 data += offset;  
 uint8\_t \*question\_ptr = data;  
 // Questions  
 offset = toQname(packet.question[0].name, (char \*)data);  
 data += offset;  
 offset = \_write16(data, packet.question[0].Qtype);  
 data += offset;  
 offset = \_write16(data, packet.question[0].Qclass);  
 data += offset;  
 // Answers  
 for (int i = 0; i < packet.header->ANCOUNT; i++)  
 {  
 // 考虑压缩指针问题  
 uint16\_t new\_offset = isFind\_repeatDomain((char \*)packet.question, (char \*)packet.answer[i].name, (char \*)question\_ptr, (char \*)buffer.data);  
 if (offset >= 0)  
 {  
 int16\_t compressed\_pointer = DNS\_COMPRESSION\_POINTER(new\_offset);  
 offset = \_write16(data, compressed\_pointer);  
 }  
 else  
 {  
 offset = toQname(packet.answer[i].name, (char \*)data);  
 }  
 data += offset;  
 offset = \_write16(data, packet.answer[i].type);  
 data += offset;  
 offset = \_write16(data, packet.answer[i].addr\_class);  
 data += offset;  
 offset = \_write32(data, packet.answer[i].TTL);  
 data += offset;  
 offset = \_write16(data, packet.answer[i].rdlength);  
 data += offset;  
 memcpy(data, packet.answer[i].rdata, packet.answer[i].rdlength);  
 data += packet.answer[i].rdlength;  
 }  
 // Authorities和Additional字段各answer字段类型相同，故省略（会在完整代码中给出）  
 buffer.length = (uint32\_t)(data - buffer.data);  
 return buffer;  
}

# 5 DNS Cache

当客户端发来一个查询IP的DNS报文时，DNS中继服务器会首先查找本地的IP-域名对照表，如果在表中也没有找到对应的域名，就查找在cache中查找。并且当本地IP-域名对照表和cache里面都查不到该域名时，DNS中继服务器会给设置好的上游DNS服务器发送查询报文，并且在收到上游DNS服务器发回的响应报文时，将该IP-域名对照信息存入cache。

Cache使用一个双向链表来存储，Cache表头包括了Cache表项的长度和互斥锁，Cache表项包括了域名和IP，超时时间、计数器、包含的IP个数。其中，IP用了一个数组来存储，考虑了一个域名有多个IP的情况

## 5.1 Cache的数据结构及函数声明

#define MAX\_CACHE\_LEN 1024  
#define MAX\_IP\_COUNT 10

/\*\*  
 \* cache列表头  
 \*/  
typedef struct CACHE\_LIST  
{  
 struct list\_head list;  
 int list\_size;  
 HANDLE lock; // 互斥量句柄  
} CACHE\_LIST;  
  
/\*\*  
 \* cache条目  
 \*/  
typedef struct CACHE\_ENTRY  
{  
 struct list\_head list; // 包含了这个节点的两个指针  
 // 数据部分  
 char name[NAME\_LEN];  
 uint32\_t ip\_list[MAX\_IP\_COUNT]; // 当前域名包含的ip  
 time\_t expireTime; // 超时时间  
 uint32\_t count; // LRU算法的计数器  
 uint32\_t ip\_count; // 当前域名包含的ip个数  
} CACHE\_ENTRY;

/\* 初始化cache \*/  
void cache\_init();  
  
/\* 添加cache \*/  
void cache\_add(char \*name, uint32\_t \*ip, uint32\_t ttl, int ip\_cnt);  
  
/\* 查找cache \*/  
bool cache\_search(char \*name, uint32\_t \*\*ip\_list, int \*actual\_ip\_cnt);  
  
/\* cache列表，使用双向链表存储 \*/  
CACHE\_LIST cache\_list;

## 5.2 Cache的更新

使用LRU算法更新Cache列表。

**加入一条Cache表项：**

* 如果当前链表的长度没有超出设置好的MAX\_CACHE\_LEN，就构建一个新的CACHE\_ENTRY，并将其加入Cache列表。
  1. 在加入链表的过程中，将所有没命中的Cache表项的计数器都+1，并且删除所有超时的链表项。
* 如果当前链表的长度已经等于MAX\_CACHE\_LEN
  1. 首先查看是否有表项超时，如果有表项超时，就将其删除。当有超时报文被删除时，说明此时Cache有空闲位置，就不用删除别的表项了，直接将新表项加入，并且所有没命中的Cache表项的计数器都+1。
  2. 如果没有链表超时，就删除计数器最大的那个链表，并且所有没命中的Cache表项的计数器都+1。

**搜索Cache表项：**

* 当Cache命中时，将所有未命中并且计数器比命中Cache表项的Cache的计数器+1，并且将Cache表项里，该域名对应的所有IP（可能有多个）存入传进来的参数指向的内存区域，函数返回True。
* 当Cache未命中时，函数返回False

## 5.3 核心代码

/\*\*  
 \* 添加ip\_cnt条cache  
 \*/  
void cache\_add(char \*name, uint32\_t \*ip, uint32\_t ttl, int ip\_cnt)  
{  
 CACHE\_ENTRY \*cache\_entry = (CACHE\_ENTRY \*)malloc(sizeof(CACHE\_ENTRY));  
  
 // 将ip加入ip数组  
 for (int i = 0; i < ip\_cnt; i++)  
 {  
 cache\_entry->ip\_list[i] = ip[i];  
 }  
 // 存储域名  
 strcpy(cache\_entry->name, name);  
  
 cache\_entry->ip\_count = ip\_cnt;  
 cache\_entry->count = 0;  
 cache\_entry->expireTime = time(NULL) + ttl;  
  
 // 操作cache\_list，需要获取互斥锁  
  
 // 等待获取互斥量的控制权  
 DWORD dwWaitResult = WaitForSingleObject(cache\_list.lock, INFINITE);  
  
 switch (dwWaitResult)  
 {  
 case WAIT\_OBJECT\_0:  
 // 成功获取互斥量，可以访问共享资源  
 // 链表满了，优先删除expire time到了的节点，然后考虑使用LRU删除一个节点  
 if (cache\_list.list\_size == MAX\_CACHE\_LEN)  
 {  
 struct list\_head \*pos, \*n;  
 CACHE\_ENTRY \*entry; // 当前遍历到的节点  
 CACHE\_ENTRY \*entry\_to\_del = NULL; // 要删除的节点  
 bool has\_find\_expired\_one = false;  
 uint32\_t max\_lru\_cnt = 0;  
  
 list\_for\_each\_safe(pos, n, &cache\_list.list)  
 {  
 entry = list\_entry(pos, CACHE\_ENTRY, list);  
 // 删除所有超时的链表  
 if (entry->expireTime < time(NULL))  
 {  
 has\_find\_expired\_one = true;  
 list\_del(&entry->list);  
 free(entry);  
 cache\_list.list\_size--;  
 }  
 // 找出LRU最大的链表，如果已经删除了超时的就不用删除这一条了  
 if (entry->count > max\_lru\_cnt && !has\_find\_expired\_one)  
 {  
 max\_lru\_cnt = entry->count;  
 entry\_to\_del = entry;  
 }  
 }  
  
 // 如果没有超时的，就删除计数器最大的  
 if (!has\_find\_expired\_one && entry\_to\_del)  
 {  
 list\_del(&entry\_to\_del->list);  
 free(entry\_to\_del);  
 cache\_list.list\_size--;  
 }  
 }  
  
 // LRU：除新加入的节点外，其余未命中节点计数器+1  
 struct list\_head \*pos;  
 CACHE\_ENTRY \*entry; // 当前遍历到的节点  
 list\_for\_each(pos, &cache\_list.list)  
 {  
 entry = list\_entry(pos, CACHE\_ENTRY, list);  
 entry->count++;  
 }  
  
 // 添加新节点到链表头部  
 list\_add(&cache\_entry->list, &cache\_list.list);  
 cache\_list.list\_size++;  
 ReleaseMutex(cache\_list.lock); // 释放互斥量  
 break;  
 case WAIT\_ABANDONED:  
 // 互斥量已被放弃  
 free(cache\_entry);  
 break;  
 default:  
 // 获取互斥量失败  
 perror("Error waiting for mutex.");  
 free(cache\_entry);  
 break;  
 }  
}  
  
/\*\*  
 \* 查找一条cache并返回ip数组，若找到，其余未命中cache的count++  
 \*/  
bool cache\_search(char \*name, uint32\_t \*\*ip\_list, int \*actual\_ip\_cnt)  
{  
 bool ret = false;  
 uint32\_t hit\_cnt = 0; // 命中节点的计数器值  
 struct list\_head \*pos; // 当前遍历到的链表节点  
 CACHE\_ENTRY \*entry; // 当前遍历到的节点  
 int i = 0;  
  
 // 等待获取互斥量的控制权  
 DWORD dwWaitResult = WaitForSingleObject(cache\_list.lock, INFINITE);  
  
 switch (dwWaitResult)  
 {  
 case WAIT\_OBJECT\_0:  
 // 成功获取互斥量，可以访问共享资源  
 list\_for\_each(pos, &cache\_list.list)  
 {  
 entry = list\_entry(pos, CACHE\_ENTRY, list);  
 // cache命中  
 if (strcmp(entry->name, name) == 0)  
 {  
 // 命中了一个超时的cache，将其删除  
 if (entry->expireTime < time(NULL))  
 {  
 list\_del(&entry->list);  
 continue;  
 }  
 ret = true;  
 hit\_cnt = entry->count;  
 \*ip\_list = entry->ip\_list;  
 \*actual\_ip\_cnt = entry->ip\_count;  
 entry->count = 0;  
 }  
 }  
 if (ret == true)  
 {  
 printf("[CACHE] Hit cache\n");  
 list\_for\_each(pos, &cache\_list.list)  
 {  
 entry = list\_entry(pos, CACHE\_ENTRY, list);  
 // 未命中节点计数器+1  
 if (entry->count > hit\_cnt)  
 {  
 entry->count++;  
 }  
 }  
 }  
 ReleaseMutex(cache\_list.lock); // 释放互斥量  
  
 break;  
 case WAIT\_ABANDONED:  
 // 互斥量已被放弃  
 ret = false;  
 break;  
 default:  
 // 获取互斥量失败  
 perror("Error waiting for mutex.");  
 ret = false;  
 break;  
 }  
 return ret;  
}

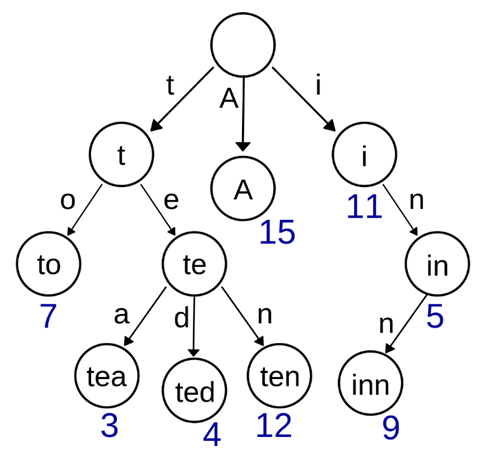
# 6 DNS 本地IP-域名对照表

在程序中，本地IP-域名对照表是relaylist.dic，里面存储了IP和域名的对照，是由程序用户在程序运行前就添加的。

* IP显示为0.0.0.0，表示此域名是不良网站的域名，应该拦截。
* IP显示正常，表示此域名是正常域名，发送响应包时，IP字段应是该IP-域名对照表里面显示的IP。

## 6.1 RELAY\_TABLE的数据结构及函数声明

为了提高查找效率，在读取本地DNS记录时，将本地DNS 记录存储在一个 Trie树中。Trie树又叫字典树或前缀树，是一种多叉树结构。如下图：



Trie树的基本性质：

* 根节点不包含字符，除根节点外的每一个子节点都包含一个字符。
* 从根节点到某一个节点，路径上经过的字符连接起来，为该节点对应的字符串。
* 每个节点的所有子节点包含的字符互不相同。

Trie树的关键字一般都是字符串，而且Trie树把每个关键字保存在一条路径上，而不是一个结点中。另外，两个有公共前缀的关键字，在Trie树中前缀部分的路径相同，所以Trie树又叫做前缀树。而本地IP-域名对照表较长，并且不会在里面进行增删，需要查找的效率较高，因此采用Trie树来存储。

/\* 域名-ip对照表，使用字典树存储 \*/  
#define block\_table trie  
  
/\*\*  
 \* 域名-ip对照表  
 \*/  
typedef struct RELAY\_TABLE  
{  
 uint32\_t ipv4; // IP  
 char \*name; // 域名  
} RELAY\_TABLE;  
  
/\* 初始化拦截表 \*/  
void relay\_table\_init();

/\*\*  
 \* 字典树叶子的数据  
 \*/  
typedef struct Trie\_Leaf  
{  
 uint32\_t ip;  
 time\_t expireTime;  
} Trie\_Leaf;  
  
/\*\*  
 \* 字典树节点  
 \*/  
typedef struct Trie  
{  
 struct Trie \*children[37]; // 包括26个字母、0~9、.符号  
 Trie\_Leaf \*leaf; // 叶子节点信息  
 bool is\_end; // 标识是否为一个完整的域名结尾  
} Trie;  
  
Trie \*trie;  
  
char ALPHABET[37] = "abcdefghijklmnopqrstuvwxyz0123456789.";  
  
/\* 初始化字典树 \*/  
void trie\_init();  
  
/\* 在字典树里插入信息 \*/  
void trie\_insert(char \*domain, uint32\_t ip);  
  
/\* 获取字符在字母表里的索引 \*/  
int get\_char\_index(char c);  
  
/\* 根据域名查找字典树叶子节点 \*/  
bool trie\_search(char \*domain, uint32\_t \*ip);

## 6.2 RELAY\_TABLE的更新

**RELAY\_TABLE的加入：**

* 通过循环遍历域名的每个字符，找到该字符在字符表的索引，逐级在字典树中插入节点。
* 每个节点使用一个 Trie 结构体表示，包含指向子节点的指针数组 children。
* 最终的叶子节点（表示域名结束的节点）包含一个 Trie\_Leaf 结构体，存储对应的IP地址。

**RELAY\_TABLE的查找：**

* 使用循环遍历域名 domain 的每个字符。
* get\_char\_index(domain[i]) 根据当前字符获取其在字典树节点数组中的索引 index。
* 如果当前节点 t 为空，或者当前节点的 children[index] 为空，说明在字典树中找不到对应的路径，直接返回 false 表示未找到。

## 6.3 核心代码

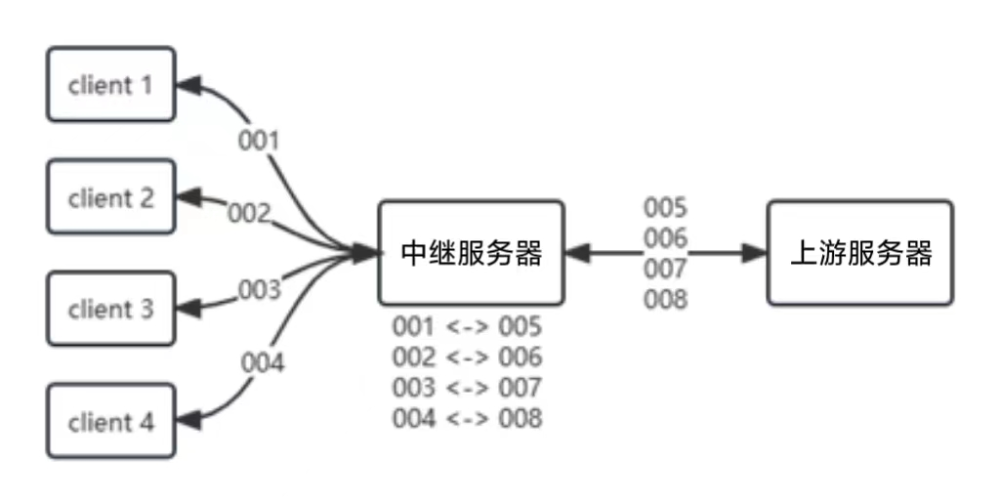
/\*\*  
 \* 获取字符在ALPHABET的索引  
 \*/  
int get\_char\_index(char c)  
{  
 if (c >= 'A' && c <= 'Z')  
 c = c + 'a' - 'A';  
 if (c >= 'a' && c <= 'z')  
 return c - 'a';  
 else if (c >= '0' && c <= '9')  
 return c - '0' + 26;  
 else if (c == '.')  
 return 36;  
 return -1; // 错误情况，应该处理错误输入  
}

/\*\*  
 \* 字典树插入节点  
 \* ip为0.0.0.0表示域名不存在（拦截），ip正常表示域名ip都正常返回  
 \*/  
void trie\_insert(char \*domain, uint32\_t ip)  
{  
 Trie \*t = trie;  
 for (int i = 0; i < strlen(domain); i++)  
 {  
 int index = get\_char\_index(domain[i]);  
 if (t->children[index] == NULL)  
 {  
 t->children[index] = (Trie \*)malloc(sizeof(Trie));  
 memset(t->children[index], 0, sizeof(Trie));  
 t->children[index]->leaf = NULL;  
 t->children[index]->is\_end = false;  
 }  
 t = t->children[index];  
 }  
 // 最后一个字符的节点标记为域名结尾  
 t->is\_end = true;  
 if (t->leaf == NULL)  
 {  
 t->leaf = (Trie\_Leaf \*)malloc(sizeof(Trie\_Leaf));  
 }  
 t->leaf->ip = ip;  
}

/\*\*  
 \* 根据域名查找字典树节点  
 \*/  
bool trie\_search(char \*domain, uint32\_t \*ip)  
{  
 Trie \*t = trie;  
 for (int i = 0; i < strlen(domain); i++)  
 {  
 int index = get\_char\_index(domain[i]);  
 if (t == NULL || t->children[index] == NULL)  
 return false; // 没找到  
 t = t->children[index];  
 }  
 // 到达域名末尾，需要检查是否是完整的域名  
 if (t->is\_end && t->leaf != NULL)  
 {  
 \*ip = t->leaf->ip;  
 return true;  
 }  
 return false;  
}

# 7 DNS报文会话id管理

DNS中继服务器在工作中，需要同时和客户端以及上游服务器进行通信，因此需要对通信时的会话id进行管理，避免id重复。原理如下：



## 7.1 IdMap数据结构

typedef struct idMap {  
 time\_t time; //过期时间  
 uint16\_t originalId; //请求方ID  
 struct sockaddr\_in addr; //请求方IP+端口  
} IdMap;

## 7.2 相关函数

### 7.2.1 初始化一个IdMap类型的数组

IdMap \*initIdMap(){  
 IdMap \*idmap = (IdMap \*)malloc(sizeof(IdMap) \* (MAXID + 1)); // 为0-65535共65536个id的IdMap分配空间  
 for(int i=0; i <= MAXID; i++){  
 idmap[i].time = 0; // 把每一个id的过期时间初始化为0  
 }  
 return idmap;  
}

### 7.2.2 寻找并分配空闲id号

uint16\_t setIdMap(IdMap \*idMap, IdMap item, uint16\_t curMaxId)  
{  
 uint16\_t originId = curMaxId; // 暂存上次向上级发出查询请求时的会话id  
 time\_t t = time(NULL); // 将t设为当前时间  
 while (idMap[curMaxId].time >= t) // 从上次的会话id开始，寻找空闲id，若过期时间大于当前时间说明id正在被占用  
 {   
 curMaxId++; // 若当前id正在被占用，则id++，查看下一个id是否可用  
 curMaxId %= (MAXID + 1); // 防止id号超过65535  
 if (curMaxId == originId)  
 { // 如果找了一整圈，回到起始的id，说明所有id都被占用，无可用的id号  
 return -1; // id分配失败  
 }  
 }  
 idMap[curMaxId % (MAXID + 1)] = item; // 将runtime中的idMap数组的信息更新  
 return curMaxId % (MAXID + 1); // 将当前空闲id设为本次向上游服务器发出请求的id  
}

### 7.2.3 释放id号并返回该id对应的IdMap信息

IdMap getIdMap(IdMap \*idMap, uint16\_t i)  
{  
 idMap[i].time = 0; // 归还原来的会话id，把过期时间还原为0  
 return idMap[i]; // 返回会话id对应的idMap项  
}

## 7.3 函数应用

### 7.3.1 初始化

在初始化runtime时，同时初始化一个IdMap类型的数组：

DNS\_RUNTIME runtime\_init(DNS\_CONFIG \*config) {  
 // 其他初始化项省略  
 runtime.idmap = initIdMap();  
 runtime.maxId = 0; // 上一次向上游服务器发送查询请求时所使用的id号  
 return runtime;  
}

### 7.3.2 向上游服务器转发DNS查询报文

在处理客户端发来的查询请求时，若在relaylist中未找到并且cache也未命中，则需向上游服务发送查询请求。此时要重新分配一个空闲的会话id号，用于与上游沟通：

IdMap mapItem; // 声明一个新的IdMap项，用于存储新会话的信息  
mapItem.addr = request->client\_addr; // 记录请求方的地址  
mapItem.originalId = dnspacket.header->ID; // 记录请求方的查询报文的id，用于后续将响应报文发回给客户端时维持会话id一致性  
mapItem.time = time(NULL) + IDMAP\_TIMEOUT; // 设置该记录的过期时间  
runtime->maxId = setIdMap(runtime->idmap, mapItem, runtime->maxId); // 分配一个空闲id号，并记录在runtime中  
dnspacket.header->ID = runtime->maxId; // 将新分配的id号设置为向上游查询包的id

### 7.3.3 向客户端转发DNS响应报文

在接收到上游服务器发来的响应报文后，需要将其转发给客户端，此时需要将报文的id设置为原来客户端发送查询请求时的报文id：

IdMap client = getIdMap(runtime->idmap, packet.header->ID); // 释放本次会话id并将id对应的IdMap项返回，用于找到原来客户端发来的查询报文的id号  
\_write16(buffer.data, client.originalId); // 还原id

# 8 接收并处理DNS报文

## 8.1 接包函数

### 8.1.1 核心思路

每当socket接到包时，就通过接包函数recvPacket来存储包的内容以及将包转换为DNS\_PKT形式。

### 8.1.2 核心代码

DNS\_PKT recvPacket(DNS\_RUNTIME \*runtime, SOCKET socket, Buffer \*buffer, struct sockaddr\_in \*client\_Addr, int \*error)  
{  
 int client\_AddrLength = sizeof(\*client\_Addr);  
 int recvBytes = recvfrom(socket, (char \*)buffer->data, buffer->length, 0, (struct sockaddr \*)client\_Addr, &client\_AddrLength);  
 DNS\_PKT packet = init\_DNSpacket(); // 初始化一个DNS包  
 if (recvBytes == SOCKET\_ERROR)  
 { // 若接收包过程出现异常  
 printf("recvfrom failed:%d\n", WSAGetLastError());  
 \*error = -1; // 指示DNS包接收状态为故障  
 }  
 else  
 { // 正常接受包  
 buffer->length = recvBytes; // 更新buffer长度字段  
 \*error = recvBytes; // 给函数调用者的标识  
 DNSPacket\_decode(buffer, &packet); // 将字节流形式的包转换为DNS\_PKT形式  
 if (buffer->length == 0)  
 {  
 \*error = -2; // 指示接收包为空包  
 packet = init\_DNSpacket(); // 将DNS\_PKT形式的包设为空包  
 }  
 }  
 return packet;  
}

## 8.2 接收并处理客户端报文

### 8.2.1 核心思路

每当接到客户端发来的包时，便会进入HandleFromClient函数。在buffer中存储该DNS包的字节流形式，并将接该包转换为DNS\_PKT形式，存储在dnspacket中。然后将该包放入任务队列，等待工作线程处理。

工作线程从任务队列中取出请求，开始对包进行处理。处理逻辑为：

1. 先在relaylist中寻找域名
2. 若找到了且该ip为0.0.0.0，则拦截，给客户端发送域名不存在的响应报文
3. 若找到了且该ip为正常ip，则给客户端发送正常响应报文，在回答区段的rdata中写入ip
4. 若在relaylist中未找到，则在cache中寻找
5. 若cache命中，则客户端发送正常响应报文，在回答区段的rdata中写入ip
6. 若cache未命中，则向上游服务器发送查询请求

### 8.2.2 核心代码

void HandleFromClient(DNS\_RUNTIME \*runtime)  
{  
 Buffer buffer = makeBuffer(DNS\_PACKET\_SIZE); // 创建缓冲区  
 struct sockaddr\_in client\_Addr; // 存储客户端的地址信息(IP + port)  
 int status = 0; // 存储接收数据包的状态  
  
 DNS\_PKT dnspacket = recvPacket(runtime, runtime->server, &buffer, &client\_Addr, &status); // 接收数据包  
 if (status <= 0)  
 {  
 free(buffer.data);  
 return;  
 }  
  
 enqueue\_task(client\_Addr, dnspacket, buffer); // 将请求放入队列  
}

unsigned \_\_stdcall worker\_thread(void \*arg)  
{  
 DNS\_RUNTIME \*runtime = (DNS\_RUNTIME \*)arg; // 获取运行时  
 while (1)  
 {  
 // 等待任务或者关闭事件  
 HANDLE events[] = {thread\_pool.cond, thread\_pool.shutdown\_event};  
 DWORD wait\_result = WaitForMultipleObjects(2, events, FALSE, INFINITE);  
  
 if (wait\_result == WAIT\_OBJECT\_0 + 1)  
 {  
 return 0;  
 }  
  
 Request \*request = dequeue\_task(&thread\_pool.request\_queue); // 获取请求  
 if (request)  
 {  
 struct sockaddr\_in client\_Addr = request->client\_addr; // 获取客户端地址  
 Buffer buffer = request->buffer; // 获取数据缓冲区  
 DNS\_PKT dnspacket = request->dns\_packet; // 获取接收到的dns包  
  
 // 处理请求的逻辑  
 if (dnspacket.header->QR != QRQUERY || dnspacket.header->QDCOUNT != 1)  
 { // 若dnspacket不是一个查询类型的 DNS 报文或者其问题字段不是一个问题  
 DNSPacket\_destroy(dnspacket); // 销毁packet，解除内存占用  
 return 0;  
 }  
 if (dnspacket.question->Qtype == 1)  
 { // 只有当请求的资源类型为ipv4时，服务器做出回应  
 uint32\_t found\_ip[1]; // 用于存储找到的IP  
 // 先在本地relaylist中查找  
 if (trie\_search(dnspacket.question->name, &found\_ip[0])) // 若在relayList中找到了  
 {  
 prepare\_answerPacket(found\_ip, &dnspacket, 1);  
 if (found\_ip[0] == 0)  
 {  
 buffer = DNSPacket\_encode(dnspacket); // 将DNS包转换为buffer，方便发送  
 DNSPacket\_destroy(dnspacket);  
 int sendBytes = sendto(runtime->server, (char \*)buffer.data, buffer.length, 0, (struct sockaddr \*)&client\_Addr, sizeof(client\_Addr)); // 由服务器发给客户端找到的ip信息  
 free(buffer.data);  
 if (sendBytes == SOCKET\_ERROR)  
 {  
 printf("sendto failed: %d\n", WSAGetLastError());  
 WSACleanup();  
 }  
 }  
 else  
 {  
 // 若在relaylist中不存在，则再在cache中搜索  
 int actual\_ip\_cnt = 0;  
 uint32\_t \*target\_ip;  
 bool find\_result = cache\_search(dnspacket.question->name, &target\_ip, &actual\_ip\_cnt);  
 if (find\_result)  
 {  
 // 若在cache中查询到了结果  
 prepare\_answerPacket(target\_ip, &dnspacket, actual\_ip\_cnt);  
 dnspacket.header->RA = 1;  
 buffer = DNSPacket\_encode(dnspacket); // 将DNS包转换为buffer，方便发送  
 int sendBytes = sendto(runtime->server, (char \*)buffer.data, buffer.length, 0, (struct sockaddr \*)&client\_Addr, sizeof(client\_Addr)); // 由服务器发给客户端找到的ip信息  
 free(buffer.data);  
 if (sendBytes == SOCKET\_ERROR)  
 {  
 printf("sendto failed: %d\n", WSAGetLastError());  
 WSACleanup();  
 }  
 }  
 // 若cache未命中，则需要向上级发送包进一步查询  
 IdMap mapItem; // 声明一个ID转换表  
 mapItem.addr = request->client\_addr; // 请求方的地址  
 mapItem.originalId = dnspacket.header->ID; // 请求方的ID  
 mapItem.time = time(NULL) + IDMAP\_TIMEOUT; // 设置该记录的过期时间  
 runtime->maxId = setIdMap(runtime->idmap, mapItem, runtime->maxId);  
 dnspacket.header->ID = runtime->maxId;  
 // 发走  
 buffer = DNSPacket\_encode(dnspacket);  
 DNSPacket\_destroy(dnspacket);  
 int status = sendto(runtime->client, (char \*)buffer.data, buffer.length, 0, (struct sockaddr \*)&runtime->upstream\_addr, sizeof(runtime->upstream\_addr));  
 free(buffer.data);  
 free(request);  
 }  
 }  
 }  
 }  
 return 0;  
}

## 8.3 接收并处理上游服务器的响应报文

### 8.3.1 核心思路

每当接到上游服务器发来的包时，便会进入HandleFromUpStream函数，对包进行处理并发回客户端。此 外，还需将上游响应的域名-ip信息存储在cache中。

### 8.3.2 核心代码

void HandleFromUpstream(DNS\_RUNTIME \*runtime)  
{  
 Buffer buffer = makeBuffer(DNS\_PACKET\_SIZE); // 创建一个缓冲区，用于存放收到的包的数据  
 int status = 0;  
 DNS\_PKT packet = recvPacket(runtime, runtime->client, &buffer, &runtime->upstream\_addr, &status); // 接包，包的二进制数据存储在buffer中  
  
 if (status <= 0)  
 {  
 // 接收失败  
 free(buffer.data);  
 return;  
 }  
 IdMap client = getIdMap(runtime->idmap, packet.header->ID);  
 \_write16(buffer.data, client.originalId); // 还原id  
  
 // 将接收到的上游应答 发送回客户端  
 status = sendto(runtime->server, (char \*)buffer.data, buffer.length, 0, (struct sockaddr \*)&client.addr, sizeof(client.addr));  
 if (status == SOCKET\_ERROR)  
 {  
 printf("sendto failed: %d\n", WSAGetLastError());  
 WSACleanup();  
 }  
 if (status < buffer.length)  
 {  
 printf("Error sendto: %d\n", WSAGetLastError());  
 }  
 // 判断是否应该缓存  
 int shouldCache = 1;  
 if (packet.header->Rcode != OK || packet.question->Qtype != A || packet.header->ANCOUNT < 1)  
 {  
 shouldCache = 0; // 若在查询中指定域名不存在，或Qtype不是IPv4，或上游服务器应答中answer数量<1，则不缓存  
 }  
 if (shouldCache)  
 {  
 uint32\_t \*ip\_Array = (uint32\_t \*)malloc(sizeof(uint32\_t) \* packet.header->ANCOUNT); // 创建一个IP数组，用于存储该域名的所有IP  
 for (int i = 0; i < packet.header->ANCOUNT; i++)  
 {  
 \_read32((uint8\_t \*)packet.answer[i].rdata, &ip\_Array[i]); // 将rdata中的IP地址写入ip\_Arrray  
 }  
 cache\_add(packet.answer[0].name, ip\_Array, packet.answer[0].TTL, packet.header->ANCOUNT); // 该域名的所有IP地址以及其他信息存入cache中  
 }  
 // 用完销毁  
 free(buffer.data);  
 DNSPacket\_destroy(packet);  
}

# 9 多线程

为了实现更高效的接收查询报文和发送响应报文，我们的程序引入了多线程，采用了“生产者/消费者”模式。因为接受请求比处理请求（包括向上游服务器查询）更快，因此我们设置了一个“生产者”线程为主线程，负责接收客户端发到DNS中继服务器的查询报文，并且将这个查询报文以及里面的信息解析出来，存入到一个线程共享的“任务队列中”；设置了多个线程作为“消费者”线程，负责将任务从任务队列里面取出，并处理它，最终向客户端发送DNS响应报文。

该队列是一个结构体，里面包含一个互斥锁和一个信号量，这都是为了避免CPU忙等的结构。当“生产者”线程收到一个请求时，会获取这个队列的互斥锁，在成功往里面添加任务后，会释放互斥锁并且使用SetEvent函数通知其他“消费者”线程任务队列里面有新东西了，而“消费者”线程收到这个通知之后，会尝试获取该任务的互斥锁，如果没有获取到，线程会主动转让CPU使用权给获取到了这个锁的线程，让它先处理任务，从而避免了CPU忙等。

## 9.1 线程池

线程池负责存储所有线程的句柄和任务队列，以便在程序启动时为每个线程分配任务以及在程序结束时等待所有线程结束并关闭线程，从而使程序安全初始化以及退出。

### 9.1.1 线程池的定义及函数声明

/\*\*  
 \* 定义线程池  
 \*/  
typedef struct ThreadPool  
{  
 HANDLE threads[THREAD\_COUNT]; // 存放线程的句柄  
 int num\_threads;  
 // int max\_threads;  
 RequestQueue request\_queue;  
 HANDLE mutex;  
 HANDLE cond;  
 HANDLE shutdown\_event;  
} ThreadPool;  
  
/\* 线程池变量 \*/  
ThreadPool thread\_pool;  
  
/\* 初始化线程池 \*/  
void init\_thread\_pool();  
  
/\* 销毁线程池 \*/  
void destroy\_thread\_pool();

### 9.2.2 线程池的核心代码

根据之前存储的线程的句柄，等待线程结束并关闭线程，安全退出程序。

/\*\*  
 \* 初始化线程池  
 \*/  
void init\_thread\_pool()  
{  
 thread\_pool.num\_threads = THREAD\_COUNT; // 初始线程为最少线程量  
 thread\_pool.mutex = CreateMutex(NULL, FALSE, NULL); // 创建互斥锁  
 thread\_pool.cond = CreateEvent(NULL, FALSE, FALSE, NULL); // 创建条件变量  
 thread\_pool.shutdown\_event = CreateEvent(NULL, FALSE, FALSE, NULL); // 线程池关闭事件  
 init\_request\_queue(&thread\_pool.request\_queue);  
  
 // 创建初始线程  
 for (int i = 0; i < thread\_pool.num\_threads; i++)  
 {  
 // 保存线程句柄  
 thread\_pool.threads[i] = (HANDLE)\_beginthreadex(NULL, 0, worker\_thread, &runtime, 0, NULL);  
 }  
}  
  
/\*\*  
 \* 销毁线程池  
 \*/  
void destroy\_thread\_pool()  
{  
 // 设置关闭事件  
 SetEvent(thread\_pool.shutdown\_event);  
  
 // 等待所有线程退出  
 WaitForMultipleObjects(thread\_pool.num\_threads, thread\_pool.threads, TRUE, INFINITE);  
  
 for (int i = 0; i < thread\_pool.num\_threads; i++)  
 {  
 CloseHandle(thread\_pool.threads[i]);  
 }  
 destroy\_request\_queue(&thread\_pool.request\_queue);  
 CloseHandle(thread\_pool.mutex);  
 CloseHandle(thread\_pool.cond);  
 CloseHandle(thread\_pool.shutdown\_event);  
}

## 9.2 任务队列

任务队列是程序很关键的一个数据结构，并且一直处于被多个线程访问和修改的状态，因此任务队列必须加入互斥锁和信号量来保证线程安全。任务队列里面是否有新的任务也将决定“消费者线程是否被唤醒”。

### 9.2.1 任务队列的数据结构及函数声明

/\*\*  
 \* 定义任务结构  
 \*/  
typedef struct Request  
{  
 struct list\_head list; // 链表节点  
 struct sockaddr\_in client\_addr; // 客户端地址信息  
 DNS\_PKT dns\_packet; // 收到的包  
 Buffer buffer; // 数据缓冲区  
} Request;  
  
/\*\*  
 \* 定义任务链表  
 \*/  
typedef struct  
{  
 struct list\_head head; // 链表头  
 int queue\_len; // 任务队列长度  
 HANDLE mutex; // 互斥锁  
 HANDLE cond; // 条件变量  
} RequestQueue;  
  
/\* 初始化任务队列 \*/  
void init\_request\_queue(RequestQueue \*queue);  
  
/\* 往请求队列添加请求 \*/  
void enqueue\_request(RequestQueue \*queue, struct sockaddr\_in client\_addr, DNS\_PKT pkt, Buffer buffer);  
  
/\* 往任务队列添加任务 \*/  
void enqueue\_task(struct sockaddr\_in client\_addr, DNS\_PKT pkt, Buffer buffer);  
  
/\* 从请求队列取出请求 \*/  
Request \*dequeue\_request(RequestQueue \*queue);  
  
/\* 从任务队列取出任务 \*/  
Request \*dequeue\_task(Request \*request);  
  
/\* 销毁任务队列 \*/  
void destroy\_request\_queue(RequestQueue \*queue);

### 9.2.2 任务队列的核心代码

/\*\*  
 \* 向任务队列添加任务  
 \*/  
void enqueue\_request(RequestQueue \*queue, struct sockaddr\_in client\_addr, DNS\_PKT pkt, Buffer buffer)  
{  
 Request \*request = (Request \*)malloc(sizeof(Request)); // 分配新的请求节点  
 request->client\_addr = client\_addr; // 设置客户端地址  
 request->buffer = buffer; // 设置数据缓冲区  
 request->dns\_packet = pkt; // 设置接收到的dns包  
 INIT\_LIST\_HEAD(&request->list); // 初始化链表节点  
  
 WaitForSingleObject(queue->mutex, INFINITE); // 获取锁  
 list\_add\_tail(&request->list, &queue->head); // 添加到链表尾部  
 SetEvent(thread\_pool.cond); // 通知线程  
 queue->queue\_len++; // 队列长度++  
 ReleaseMutex(queue->mutex); // 释放锁  
}

首先获取互斥锁，操作完任务队列后释放互斥锁并通知“消费者”线程，将其唤醒。

/\*\*  
 \* 从任务队列获取任务  
 \*/  
Request \*dequeue\_request(RequestQueue \*queue)  
{  
 WaitForSingleObject(queue->mutex, INFINITE); // 加锁  
 if (list\_empty(&queue->head))  
 { // 如果队列为空，解锁并返回NULL  
 ReleaseMutex(queue->mutex);  
 return NULL;  
 }  
  
 struct list\_head \*pos = queue->head.next; // 获取队列头部的节点  
 list\_del(pos); // 从队列中删除  
 ReleaseMutex(queue->mutex); // 解锁  
  
 return list\_entry(pos, Request, list); // 返回请求节点  
}

首先获取互斥锁，操作完成后释放锁并且返回一个请求。

## 9.3 工作线程

即之前所说的“消费者线程”。

### 9.3.1 工作线程的定义及函数声明

/\* 工作线程 \*/  
unsigned \_\_stdcall worker\_thread(void \*arg);

\_\_stdcall 是一种调用约定（calling convention），它影响了函数在被调用时参数如何传递、函数如何处理堆栈以及如何返回值给调用者。在Windows平台上，\_\_stdcall 是一种常见的调用约定，特别适合于Windows API函数和线程函数。

它接收一个 void \* 类型的参数 arg，用于传递线程函数所需的参数或者上下文信息，在我们的程序里，这个信息就是RUNTIME，里面包括了接收请求和与上游服务器连接的两个SOCKET，以及本机地址和上游服务器地址，因此在这个线程里面可以用这些信息来完成请求的接收和发送。

### 9.3.2 工作线程的核心代码

/\*\*  
 \* “消费者”函数  
 \* 任务是从请求队列里面取出一个请求并处理这个请求  
 \* 解码DNS包  
 \* 查看cache是否命中，未命中向上游服务器发出请求  
 \*/  
unsigned \_\_stdcall worker\_thread(void \*arg)  
{  
 DNS\_RUNTIME \*runtime = (DNS\_RUNTIME \*)arg; // 获取运行时  
 while (1)  
 {  
 // 等待任务或者关闭事件  
 HANDLE events[] = {thread\_pool.cond, thread\_pool.shutdown\_event};  
 DWORD wait\_result = WaitForMultipleObjects(2, events, FALSE, INFINITE);  
  
 if (wait\_result == WAIT\_OBJECT\_0 + 1)  
 {  
 return 0;  
 }  
 if (config.debug)  
 printf("[THREAD] Worker\_thread start\n");  
  
 Request \*request = dequeue\_task(&thread\_pool.request\_queue); // 获取请求  
 if (request)  
 {  
 if (config.debug)  
 printf("[RECEIVE] Request from client received\n");  
 ...  
 // 请求处理逻辑  
 ...  
 }  
 }  
}

在这里，线程以阻塞的状态等待任务队列新增事件或者线程池关闭事件的发生，如果有事件发生，就进行处理，没事件发生的时候也不会一直忙等。

# 10 主函数main

## 10.1 主函数的作用

主函数结合了以上所有的函数，简洁地初始化了程序的配置信息、运行时、socket、本地IP-域名对照表以及cache，并且通过loop开启了主线程，通过初始化线程池开启了工作线程。同时设置了按下Ctrl+C后程序释放空间、关闭线程并安全退出。为整个程序划分出了最主要的框架和处理逻辑。

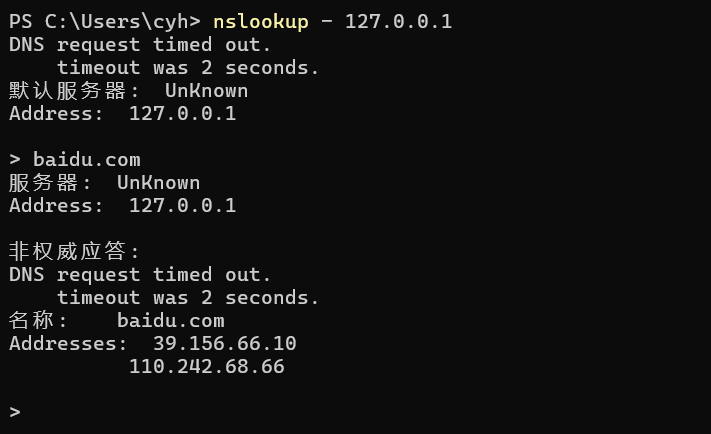
## 10.2 主函数的核心代码

int main(int argc, char \*\*argv)  
{  
  
 /\* 初始化Windows套接字 \*/  
 WSADATA wsaData;  
 if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)  
 {  
 printf("WSAStartup failed: %d\n", WSAGetLastError());  
 return 1;  
 }  
  
 /\* 初始化DNS服务器配置 \*/  
 config = config\_init(argc, argv);  
  
 /\* 初始化DNS服务器运行时 \*/  
 runtime = runtime\_init(&config);  
  
 /\* 线程池初始化，初始化请求队列，创建工作线程 \*/  
 init\_thread\_pool();  
  
 /\* 初始化拦截列表 \*/  
 relay\_table\_init();  
  
 /\* 初始化cache \*/  
 cache\_init();  
  
 /\* 初始化socket \*/  
 socket\_init(&runtime, &config);  
  
 /\* 设置Ctrl+C(SIGINT)时的友好退出 \*/  
 SetConsoleCtrlHandler(console\_handler, TRUE);  
  
 /\* 处理客户端请求 \*/  
 loop(&runtime);  
  
 /\* 程序退出 \*/  
 WSACleanup();  
 return 0;  
}

# 11 程序运行效果

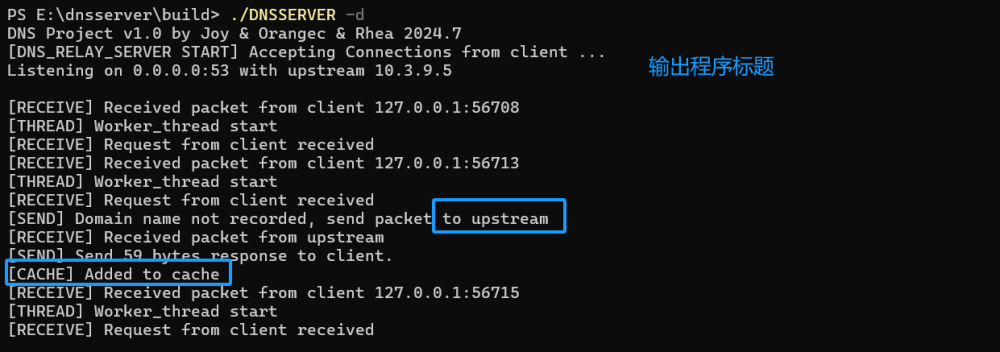
## 11.1 以-d模式运行

客户端：查询baidu.com，这是一个不在cache也不再本地IP-域名对照表里的域名



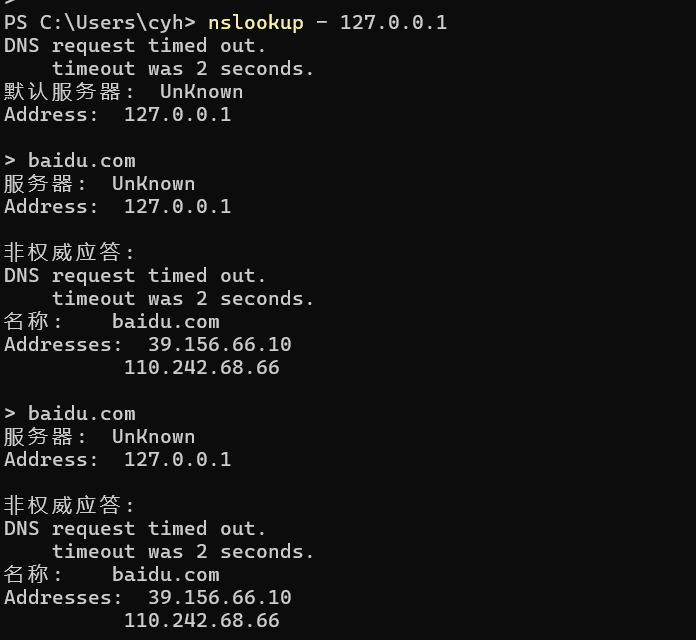
程序输出：

表示已经把查询报文发送给了上游服务器，并且在收到上游服务器发回来的包后已经将其加入了cache



再次查询baidu.com，此时该域名已经被存入cache

客户端：

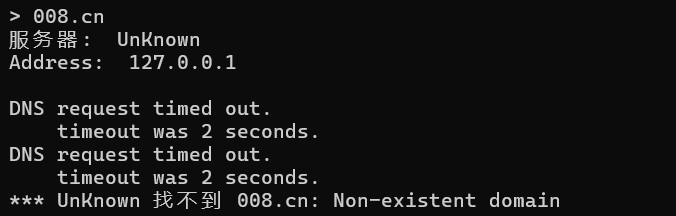


程序输出：

查询一个在本地IP-域名对照表里面的域名：

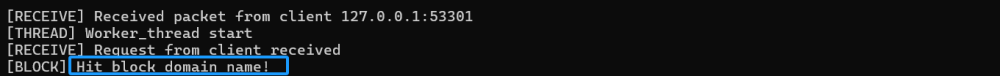


收到了域名不存在的报文



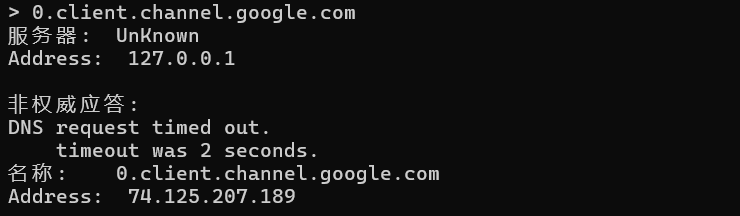
程序输出：

该域名被拦截了



查询一个在本地IP-域名对照表里面的合法域名：





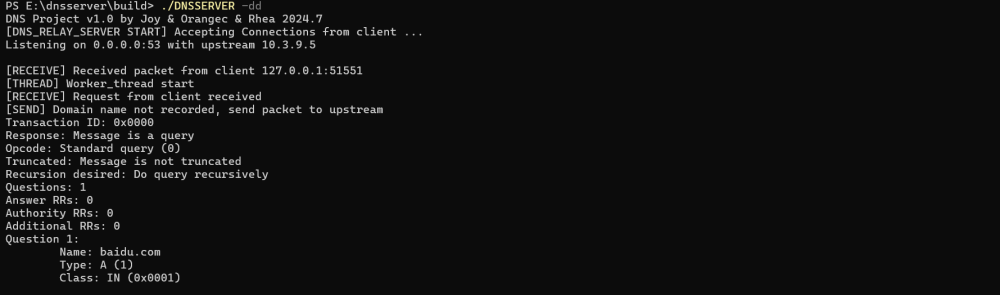
程序输出：

表示在本地IP-域名对照表找到了域名，并发送回去了



## 11.2 以-dd模式运行

接到查询报文，程序输出解析出的报文的详细信息



收到上游服务器发送回来的响应报文，程序输出响应报文的详细信息



# 12 调试中遇到并解决的问题

## 12.1 域名对应的ip数量不唯一

一开始没考虑到一个域名可能会对应多个ip的情况，导致解析DNS报文出错，并且cache中存储条目的数据结构也有误。后来观察到有的DNS响应报文中ANCOUNT>1，意识到可能返回多个ip，于是修改了cache条目的数据结构，将ip设为数组，从而存储多个ip。

## 12.2 重复free指针和忘记给指针分配空间

在释放无用指针的过程中，没有系统地考虑应当何时释放每个指针，有一搭没一搭地写，导致出现了好几处重复释放指针的错误。

而在给指针赋值时，经常忘记要先给指针分配空间，尤其是结构体内的指针，导致许多次debug监视变量时发现莫名其妙地数据就失踪了。因此，每次使用指针时，一定要先初始化指针，为其分配空间，再去考虑其他的各种操作。

## 12.3 线程释放错误

一开始在本地IP-域名对照表里面找到拦截的域名时，就在工作线程那里return 0了，导致工作线程结束，后续所有的客户端请求都超时。在debug过程中，一开始误以为是返回Rcode=3的报文时，报文构建错误，抓包看了好几遍buffer里面的原始二进制数据感觉没有问题，后来才发现不是因为发的包错了，而是线程被错误结束了。因此写多线程的程序时，一定要仔细缕清线程的运行逻辑，避免犯此处理起来类耗时耗力的错误。

## 12.4 压缩指针问题

最开始没有考虑到压缩指针问题，构造响应报文的时候，以为answer字段的name和question字段的name完全一致，导致这样发出去的包没有办法被客户端解析，因此出现超时的情况。后来发现，当这两个name是相同的时候，answer字段的name应该用压缩指针的方式来存储，该指针指向question字段的name，该指针的值也就是name相对于报文头部的偏移量，并且把该压缩指针的高两位置为1。这个问题解决之后，我们的客户端就可以正常解析报文了。

# 13 总结与心得体会

**总结**

在完成了基本的DNS中继服务器的功能的基础上，我们还完成了几个扩展功能：字典树的查找优化、LRU机制的cache。但是，我们还有一些功能需要完善：

* 我们的程序只能处理IPv4的报文，对于想要查询IPv6地址的报文没有办法回应。
* 程序响应较慢，报文一般会超时2s才发送到客户端，目前原因还不是很清楚，需要更加深入的学习和研究。
* 工作线程的数量是设定的一个常量，实际上可以根据请求的数量，来动态调整工作线程的数量，加速对请求的处理，并且在请求较少时，不浪费线程。

**心得**

当我们开始着手开发DNS中继服务器程序时，我们没想到这个项目会带给我如此深刻的收获和感受。这个过程不仅让我深入理解了DNS的工作原理，还让我们在编程技巧、问题解决能力以及对技术的热爱上有了新的认识。

开发过程中，我们遇到了很多意想不到的挑战。最初，我们对DNS协议的细节了解不深，需要阅读大量的文档和资料。这让我认识到，技术的发展需要不断的学习和探索，只有深入理解技术背后的原理，才能真正解决问题。

在编写代码的过程中，调试和测试是最耗费时间的部分。每当程序出现错误时，我们都会仔细检查代码，寻找问题的根源，用抓包的方式一个字节一个字节检查我们构建的DNS包以及解析的DNS报文。有时候，一个小小的错误可能会让我花费数小时甚至数天的时间去修复。但每当我们成功解决一个问题时，那种成就感是无与伦比的。这个过程让我们学会了耐心和细致，同时也提升了我们的程序调试能力。

**收获与成长**

通过这次项目，我们在编程技巧上有了很大的提升。特别是在网络编程方面，我们学会了如何处理网络请求和响应，如何管理多个并发连接，以及如何优化程序性能。这些技能不仅对这次项目有帮助，对我们未来的编程工作也将大有裨益。

此外，这个项目还让我们对团队协作有了更深的理解。在遇到困难时，我们会和队友一起解决，一起在debug时逐个检查每一个变量的变化。通过讨论和交流，我们学到了很多新的思路和解决问题的方法。技术不仅仅是一个人的事情，团队的力量可以帮助我们更快地解决问题，共同进步。

**未来展望**

这次DNS中继服务器的开发经历，让我们对技术充满了新的热情和信心。我们意识到，只要肯努力学习，勇于面对挑战，就能够不断提升自己，实现更高的目标。未来，我们希望能够继续深入研究网络技术，开发出更多有用的程序，为互联网的发展贡献自己的力量。