**北京邮电大学《计算机网络》课程实验报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **实验**  **名称** | DNS中继服务器的实现 | | **学 院** | 计算机 | **指导教师** | 高占春 |
| **班 级** | **班内序号** | **学 号** | | **学生姓名** | **成绩** | |
| **2023211301** |  | **2023210884** | | **王玉鑫** |  | |
| **2023211301** |  | **2023210892** | | **穆文钦** |  | |
| **2023211301** |  | **2023210896** | | **王书翰** |  | |
| **实**  **验**  **内**  **容** | 设计一个DNS服务器程序，读入“域名-IP地址”对照表（一个文件），当客户端查询域名对应的IP地址时，用域名检索该对照表，得到三种检索结果：   1. 检索结果为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） 2. 检索结果为普通IP地址，则向客户返回这个地址（服务器功能） 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）   （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）  具体分工如下：王玉鑫负责代码中数据结构的建立以及多线程的实现  穆文钦负责socket通信以及收发套接字的实现  王书翰负责确定整个程序“接收—转发—响应”的逻辑、消息ID的转换，并对工程中的代码进行整合  将会对验收中出现的问题进行修正，修正结果将会在最终的实验报告中呈现 | | | | | |
| **学生实验报告** |  | | | | | |
| **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。





**计算机网络课程设计**

**学院：计算机学院（国家示范性软件学院）**

**专业： 计算机科学与技术**

**班级： 2023211301**

**小组： 第七组**

**成员： 王书翰 2023210896**

**王玉鑫 2023210884**

**穆文钦 2023210892**

**2025年6月8日**

**目录**

1 实验目的和要求

2 小组成员分工

3 实验环境描述

4 系统功能与设计

5 模块划分

6 软件流程图

7 测试用例以及运行结果

8 调试以及验收中出现的问题以及解决方案

9 总结以及心得体会

**1 实验目的和要求**

设计一个DNS服务器程序，读入“域名-IP地址”对照表（一个文件），当客户端查询域名对应的IP地址时，用域名检索该对照表，得到三种检索结果：

1、检索结果为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）

2、检索结果为普通IP地址，则向客户返回这个地址（服务器功能）

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）

（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 小组成员分工**

2023211301 2023210884 王玉鑫 负责代码中数据结构的建立以及多线程的实现

2023211301 2023210892 穆文钦 负责socket通信以及收发套接字的实现

2023211301 2023210896 王书翰 负责确定整个程序“接收—转发—响应”的逻辑、消息ID的转换，并对工程中的代码进行整合

**3 实验环境描述**

操作系统：Windows11 x64

编程语言：C

开发工具：Microsoft Visual Studio Community 2022 (64 位)

抓包工具：Wireshark Version 4.0.5

**4 系统功能设计**

**4.1、基本功能**

本项目旨在设计和实现一个DNS中继服务器程序，实现以下功能：

（1）域名-IP 地址映射查询：系统能够读取一个“域名-IP 地址”对照表，并根据客户端的域名查询请求，返回相应的 IP 地址或错误信息

（2）不良网站拦截功能：当客户端查询某个域名时，系统会先在本地对照表中检索。如果该域名对应的IP地址为0.0.0.0，则判定为不良网站，并向客户端返回"域名不存在"的错误，从而实现拦截功能。

（3）服务器功能：DNS中继服务器需监听UDP 53端口（标准DNS端口），接收客户端的DNS查询请求。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；否则继续在本地的文件中查找是否有对应的记录，如果本地文件有记录，则服务器直接将目的地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。

（4）中继功能：如果查询的域名不在本地对照表和cache存储器中，且未被拦截，则服务器需作为中继，将查询请求转发至上游DNS服务器（这里默认的外部DNS服务器地址是北邮的DNS服务器IP“10.3.9.6”），获取解析结果后返回给客户端，同时可缓存该结果以提高后续查询效率。

（5）ID转换功能：由于DNS协议使用16位ID字段匹配请求与响应，中继服务器在转发查询时需维护一个ID映射表，确保客户端请求的ID与上游服务器返回的响应ID正确匹配，避免因ID冲突导致响应错误。

（6）多客户端并发功能：多客户端并发：允许多个客户端（可能会位于不同的多台计算机）的并发查询，

（7）超时处理功能：在向上游DNS服务器转发查询时，需设置合理的超时机制。如果上游服务器未在指定时间内响应，则应向客户端返回"查询超时"错误，避免客户端长时间等待。

（8）调试日志功能：提供打印日志的功能，便于分析与调试。

**4.1、进阶功能**

（1）cache高速缓存功能

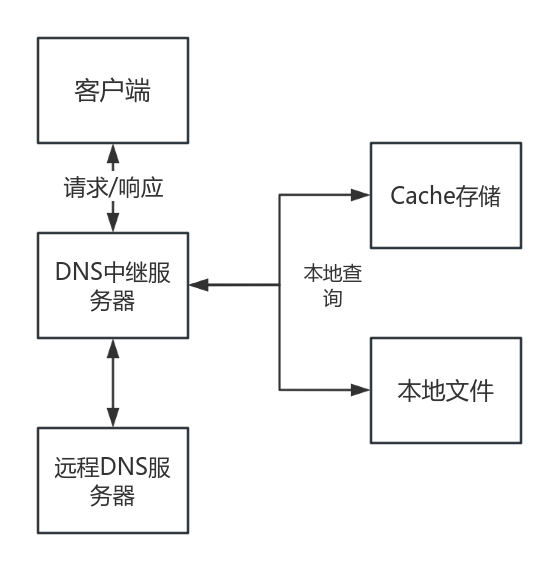
系统需要实现高效的DNS缓存机制，缓存近期查询过的域名解析结果。当收到客户端查询请求时，首先检查缓存中是否存在有效记录。如果命中缓存，将直接返回缓存结果，否则在本地的“域名-IP 地址”对照表中查询，

若命中结果则将结果缓存以供未来使用，否则将查询转发到远程 DNS 服务器

1. 查询算法的优化

为提高查询效率，实验中采用了Trie树来完成查找功能。

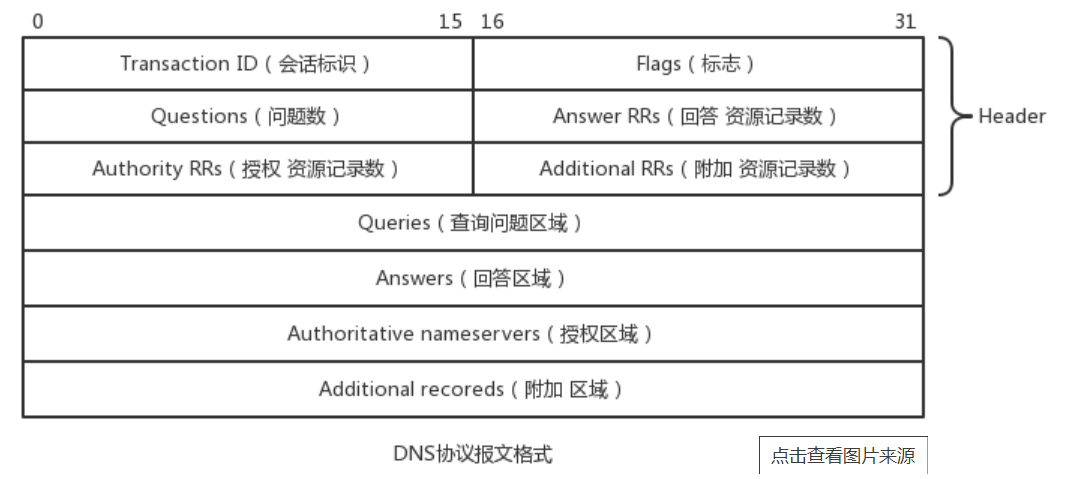
根据上述的设计需求，可以得到一个初步的功能设计图：



**5 模块划分**

**5.1DNS报文**

**5.1.1DNS报文结构**



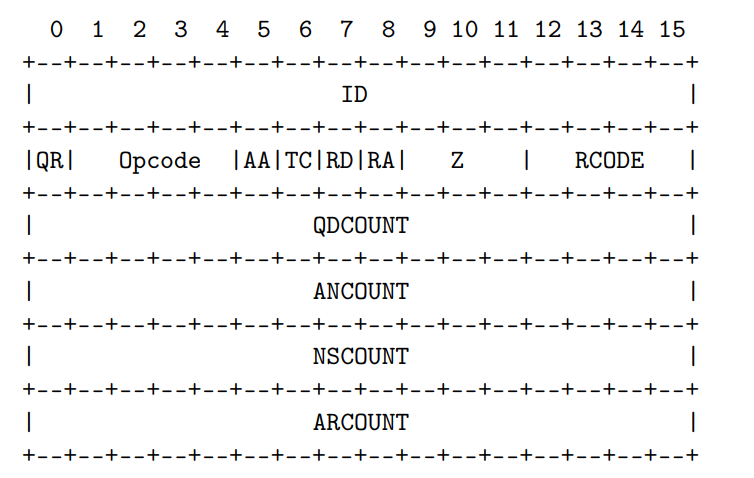
DNS报文可分为以下五个部分：

Header、Queries、Answers、Authoritative nameservers、Additional recoreds

接下来对这五个部分的结构进行详细阐述：

1、**Header（报文头）**

报文头（Header）是DNS报文的固定部分，长度为12字节。它用于标识和控制DNS报文的交互过程，包括报文的类型、状态、关联信息以及各部分的记录数量。通过报文头，DNS客户端和服务器可以快速了解报文的基本信息，从而正确处理查询和响应。具体格式如下：

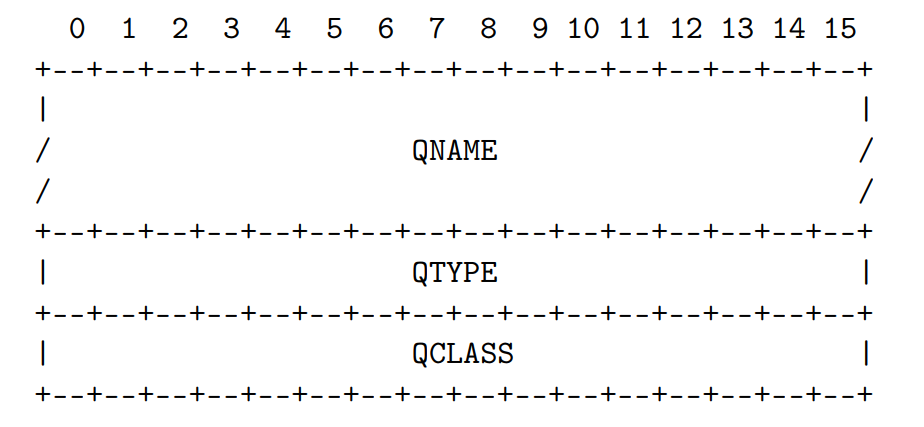


* ID（2字节）  
  16位的报文ID，用于标识DNS查询和响应的关联。客户端在发送查询时设置一个唯一的ID，服务器在响应时使用相同的ID，以便客户端匹配查询和响应。
* Flags（2字节）  
  包含多个标志位，用于控制DNS报文的类型和状态：
  + QR（1位）：查询为0，响应为1。
  + Opcode（4位）：定义报文的类型，如标准查询（0）、反向查询（1）等。
  + AA（1位）：表示响应是否来自权威DNS服务器。
  + TC（1位）：表示报文是否被截断。
  + RD（1位）：客户端请求递归查询时设置为1。
  + RA（1位）：服务器支持递归查询时设置为1。
  + Z（3位）：保留未用。
  + RCode（4位）：响应码，表示查询的状态，如无错误（0）、格式错误（1）、服务器失败（2）等。
* QDCount（2字节）  
  问题部分的记录数，通常为1。
* ANCount（2字节）  
  回答部分的记录数。
* NSCount（2字节）  
  权威部分的记录数。
* ARCount（2字节）  
  附加部分的记录数。

**2、Queries（查询问题区域）**

Question部分用于描述DNS查询的具体内容。它告诉DNS服务器客户端需要查询的域名、查询类型和类别。通过Question部分，DNS服务器可以明确客户端的需求，并提供相应的响应。

具体格式如下：

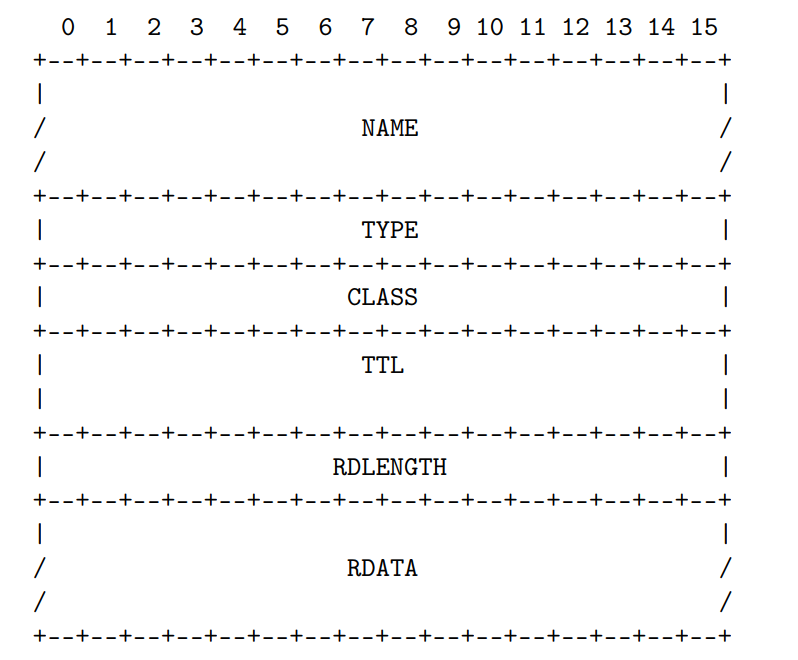


* QNAME（可变长度）  
  查询的域名，以一系列长度字节和标签组成，例如www.example.com表示为3www6example3com0。域名以点分隔的标签形式表示，每个标签前有一个长度字节，最后一个标签后有一个空字节（0x00）表示结束。
* QTYPE（2字节）  
  查询的资源记录类型，如A（IPv4地址）、AAAA（IPv6地址）、MX（邮件交换记录）、CNAME（别名记录）等。
* QCLASS（2字节）  
  查询的类别，通常为IN（Internet）。

**3、Answers（回答区域）**

Answer部分包含DNS服务器对查询的响应。它提供了查询域名对应的资源记录，包括记录类型、类别、生存时间（TTL）和具体数据。通过Answer部分，客户端可以获取到所需的DNS解析结果。

具体格式如下：



* NAME（可变长度）  
  原始查询的域名，通常是一个压缩的域名表示。DNS协议允许域名的压缩，通过指针（以0xC0开头的两个字节）指向报文中的其他位置，避免重复传输相同的域名。
* TYPE（2字节）  
  资源记录的类型，如A（IPv4地址）、AAAA（IPv6地址）、MX（邮件交换记录）等。
* CLASS（2字节）  
  资源记录的类别，通常为IN（Internet）。
* TTL（4字节）  
  资源记录的有效期，单位为秒，表示该记录可以被缓存的时间。TTL值越高，记录在客户端缓存中的存活时间越长。
* RDLength（2字节）  
  资源记录数据部分的长度。
* RData（可变长度）  
  资源记录的实际数据，例如IP地址、邮件交换服务器等。

**4、Authoritative nameservers（授权区域）**

Authority部分包含对查询域名的权威DNS服务器信息。它告诉客户端哪些DNS服务器是该域名的权威服务器，从而帮助客户端找到正确的服务器进行进一步查询。Authority部分通常用于递归查询或非权威响应。

格式与Answers部分格式相同，不再重复阐述。

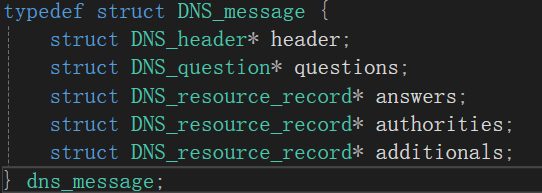
**5、Additional recoreds（附加区域）**

Additional部分提供额外的辅助信息，通常用于优化DNS查询过程。它包含权威部分中提到的DNS服务器的IP地址等信息，避免客户端需要额外查询这些信息，从而提高查询效率。

格式与Answers部分格式相同，不再重复阐述。

**5.1.2程序中定义的DNS报文结构**

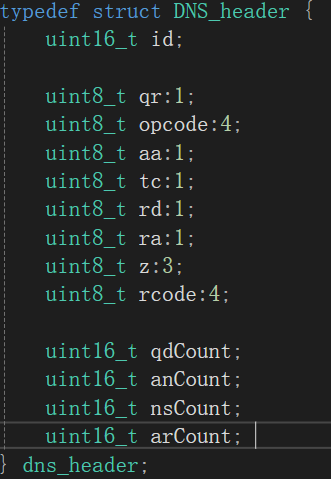
根据上面所述，我们进行DSN报文结构体的定义：



各部分与DNS报文结构相对应。

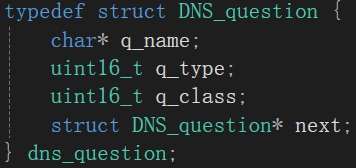
由于Answers、Authoritative、Additional部分结构相同，我们统一定义该结构为resource\_record（RR）。

下面是header部分结构体定义：



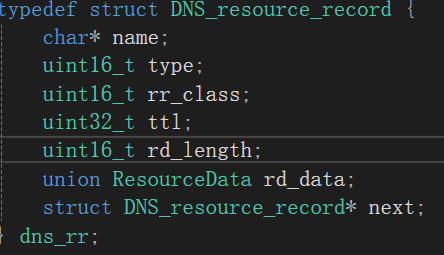
各部分与DNS报文header部分结构相对应。

下面是question部分结构体定义：



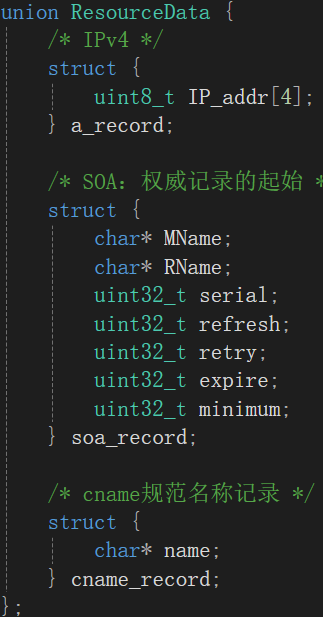
各部分与DNS报文question部分结构相对应。

下面是resource\_record部分结构体定义：



各部分与DNS报文answers、authoritative、additional部分结构相对应。

最后是resource\_data部分结构体定义：



包括三个部分：a\_record、SOA、cname

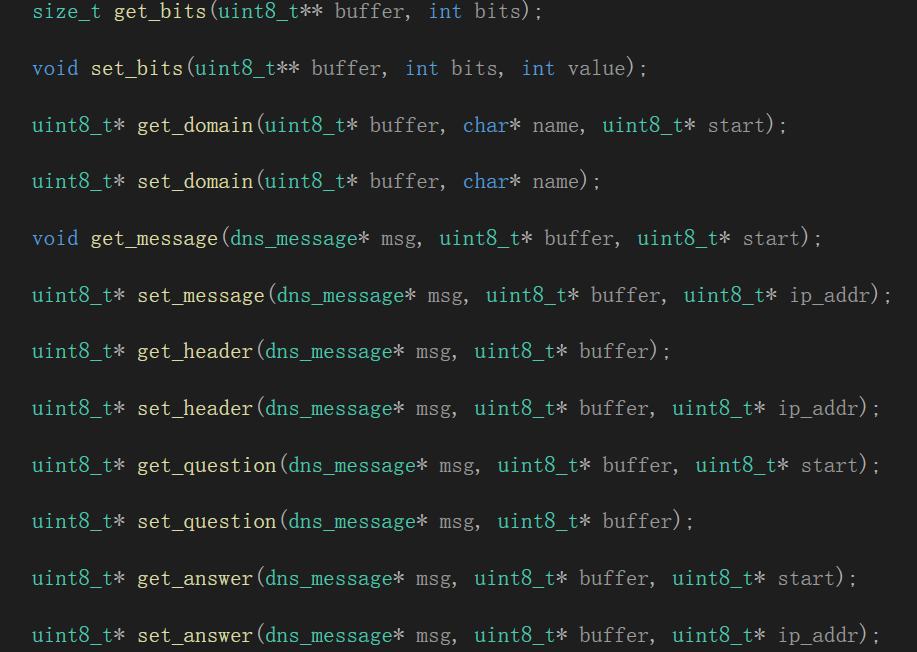
其中a\_record记录IPV4类型的ip地址（本程序不支持IPV6）。

SOA记录 DNS 区域的权威信息，包括主域名服务器、负责人邮箱、序列号、刷新时间、重试时间、过期时间和最小 TTL。

Cname记录一个域名（别名）所指向的目标域名。

**5.1.3 DNS报文的读取与组装**

由于从socket获取的信息均为字节流的形式，所以我们需要进行字节流与结构体之间的转换。以下是涉及的函数：



对于上述函数，以get打头的函数用来将字节流转换成结构体格式，主要作用在获取信息；而set打头的函数用来将结构体格式转换成字节流格式，主要作用在发送信息。

其中get\_bits函数用来获取指定数目的字节，而set\_bits函数用来写指定数目的字节。（因为我们要按照字节数将完整信息拆分为各个部分，所以需要利用这两个函数来获取/写指定数目的字节）

对于get\_message函数，其内部依次调用了get\_header、get\_question、get\_answer，这三个函数内部又分别调用了get\_bits函数。

同理对于set\_message函数，其内部依次调用了set\_header、set\_question、set\_answer，这三个函数内部又分别调用了set\_bits函数。

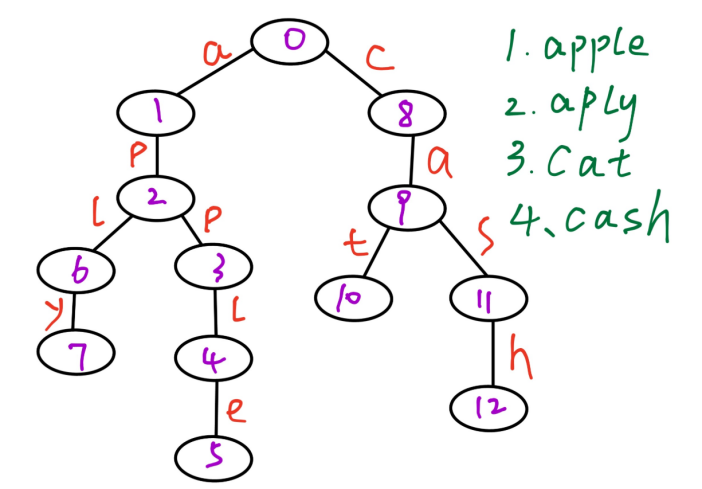
上述函数，能够实现字节流与结构体格式的转换。程序运行时，从socket读入字节流，然后将其转换成结构体的格式，方便程序处理。处理完成后，再将其转换成字节流格式，发送给socket进行传输。

**5.2本地读取与高速缓存**

**5.2.1 Trie树（本地信息存储）**

程序初始化时，要先将本地文件存储的“ip-域名”关系读入程序。此时就涉及了信息的存储问题。为了提高查找的效率，本程序使用了Trie树算法来实现存储和查找，下面将对此进行详细介绍：

**a)算法详解**

****

Trie树是一种树形结构，其中每个节点代表字符串中的一个字符。从根节点到某个节点的路径表示一个字符串前缀。Trie树的主要特点如下：

 **根节点**：通常表示空字符串。

 **内部节点**：每个节点包含多个子节点，每个子节点对应一个字符。

 **叶子节点**：表示一个完整的字符串，通常会标记为结束节点，并可能存储与该字符串相关的信息（如单词的频率、定义等）。

插入操作：

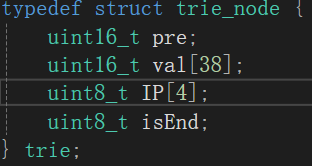
插入一个字符串到Trie树中，需要从根节点开始，逐个字符地遍历字符串。如果某个字符对应的子节点不存在，则创建一个新的子节点。到达字符串的末尾时，将最后一个节点标记为结束节点。

搜索操作：

搜索一个字符串是否存在于Trie树中，同样从根节点开始，逐个字符地遍历字符串。如果某个字符对应的子节点不存在，则说明字符串不在Trie树中。如果遍历完字符串，且最后一个节点被标记为结束节点，则说明字符串存在。

**b)代码实现**

定义Tried树结构体如下：



其中pre为父节点编号

Val为子节点编号

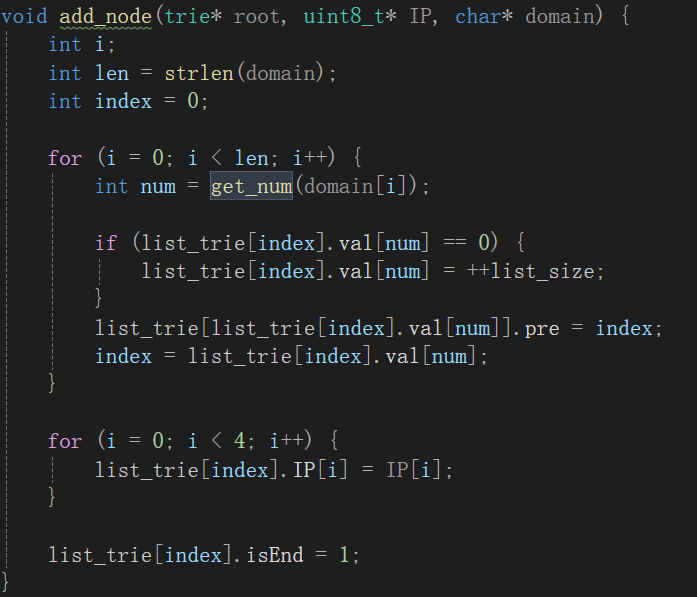
IP为IP地址

isEnd用来标记叶子节点（域名的结束）

定义增加、查询字典树结点函数如下：



其中add\_node函数如下：



传入的参数：

root为指向Tire树根节点的指针

IP为一个指向IPV4地址的指针，由四个字节构成

domain为指向域名的指针，表示要添加到Trie树中的域名

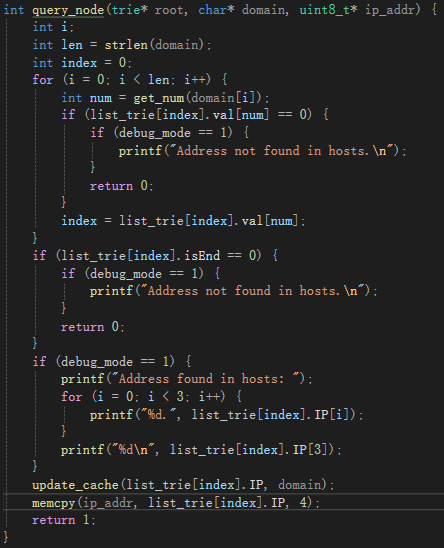
算法过程：

首先，它计算域名字符串的长度，并初始化一个索引变量。接着，代码遍历域名中的每一个字符，并将每个字符通过get\_num函数转换为一个数字索引，这个索引用来在Trie树中找到对应的节点位置。（因为域名格式规定，只会出现0-9，a-z，‘-’，‘.’这38个符号，所以我们将这38个符号映射为0-37）。

如果某个字符对应的节点不存在，算法就会在Trie树中创建一个新的节点，并更新节点的前一个节点索引以及当前节点的索引。完成域名的遍历后，算法会将传入的IP地址的四个字节依次存储到Trie树中对应节点的IP数组中。

最后，算法将当前节点标记为一个结束节点，表示这是一个完整的域名。通过这个过程，域名和它的IP地址就被成功地添加到了Trie树中，使得后续可以通过域名快速检索到对应的IP地址。

其中，query\_node函数如下：



传入的参数：

root为指向Tire树根节点的指针

domain为指向域名的指针，表示要查询的域名

ip\_addr为IPV4地址的指针，用来存储查询到的ip地址

算法过程：

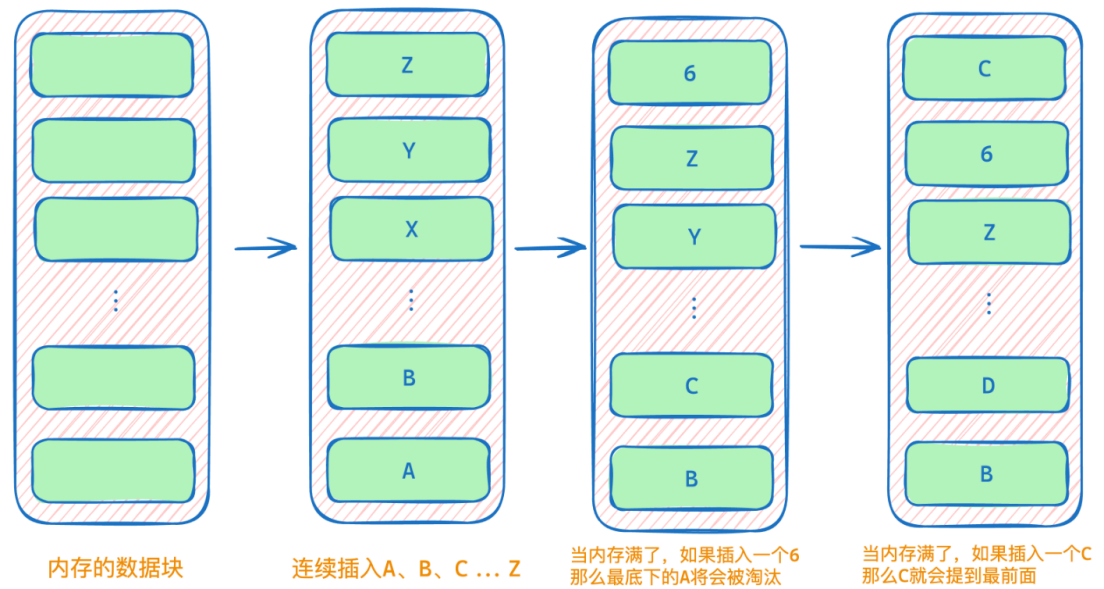
首先获取域名字符串的长度，并初始化一个索引变量index用于追踪当前节点的位置。函数通过遍历域名的每个字符，并使用get\_num函数将每个字符转换为数字索引，以便在Trie树中找到相应的节点。

如果在遍历过程中发现某个字符对应的节点不存在，则返回0表示查询失败。如果所有字符都成功匹配，函数会检查最后一个节点是否标记为结束节点，如果不是，同样返回0。如果查询成功，则输出该节点存储的IP地址。随后，函数会更新缓存并使用memcpy函数将IP地址从Trie树节点复制到提供的ip\_addr数组中，最后返回1表示查询成功。整个过程中，函数利用Trie树的结构特点，高效地实现了域名到IP地址的映射查询。

**5.2.2 LRU算法（高速缓存）**

在query\_node函数中，我们会发现一个update\_cache函数。其功能为更新缓存。接下来，我们将对缓存部分进行详细介绍：

**a)算法详解**

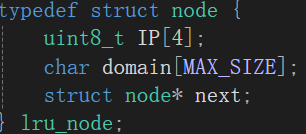
****

LRU（Least Recently Used）算法，即最近最少使用算法，是一种常用的页面置换算法，也广泛应用于缓存管理中。它的核心思想是：如果一个数据在最近一段时间内未被访问，那么在将来它被访问的可能性也较小。因此，当缓存满时，LRU算法会选择最近最少使用的数据进行淘汰，为新数据腾出空间。

在此，我们维护一个链表来实现LRU算法。

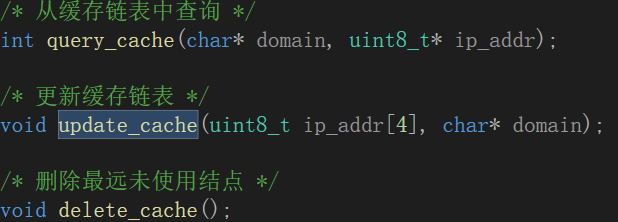
**b)代码实现**

定义链表结构体如下：

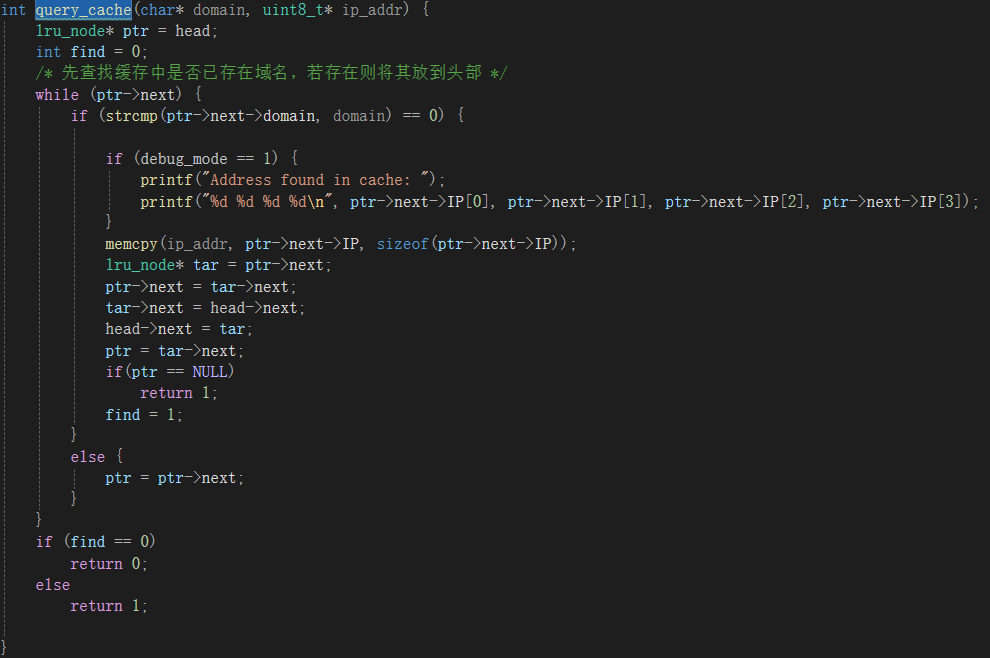


如图所示，链表的每个节点包含IP地址和域名。

定义查询、更新、删除函数如下：



其中query\_cache函数如下：



传入的参数：

domain：用户要查询的域名

ip\_addr：IPV4地址，用来存储找到的ip地址

算法过程：

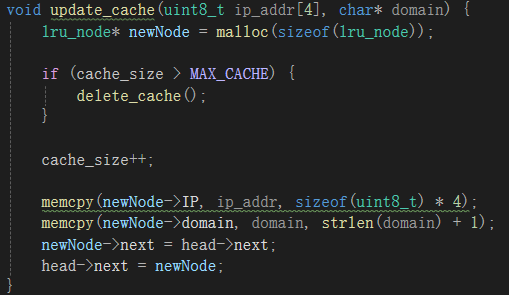
函数开始时，定义了一个指向缓存头部的指针ptr和一个标志变量find，初始化为0。然后，函数进入一个while循环，遍历缓存中的每个节点，通过比较域名来查找是否存在目标域名。

如果找到了匹配的域名，函数会检查是否处于调试模式，如果是，则打印出在缓存中找到了地址的消息以及对应的IP地址。接着，使用memcpy函数将找到的IP地址复制到提供的ip\_addr数组中。如果遍历完整个缓存都没有找到目标域名，函数将返回0。如果找到了，find标志会被设置为1，并在函数最后返回1，表示查询成功。

为了将找到的节点移动到缓存的头部，函数创建了一个临时指针tar来暂存当前节点的下一个节点，然后调整指针，将当前找到的节点移动到头部。如果移动后的节点是NULL，表示该节点原本就是头部节点，此时直接返回1。

整个过程中，函数通过移动最近访问的节点到缓存头部，确保了缓存中的数据遵循LRU策略，即最近访问的数据总是位于缓存的最前面。

其中updata\_cache函数如下：



传入的参数：

ip\_addr:用户传入的ip地址

domain:用户传入的域名

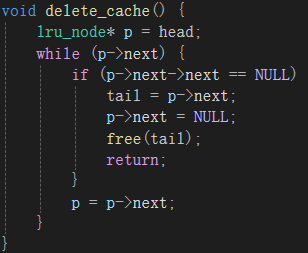
算法过程：

函数开始时，首先为新节点分配内存，然后检查当前缓存的大小是否已经超过了最大限制MAX\_CACHE。

如果缓存已满，它会调用delete\_cache函数来删除最久未使用的缓存项，以腾出空间。接着，函数将新节点的IP地址和域名从输入参数中复制到新节点中，并将新节点插入到缓存的头部，这样最近使用的缓存项总是位于缓存的最前面。如果缓存未满，就直接将新节点添加到头部。

最后，更新缓存大小cache\_size，以反映缓存中当前的节点数量。通过这种方式，update\_cache函数确保了缓存中始终保留最近使用的域名和IP地址映射，同时淘汰最久未使用的映射，以优化缓存的效率和性能。

其中delete\_cache函数如下：



算法过程：

函数开始时，定义了一个指针p指向缓存的头部。然后，它进入一个while循环，遍历缓存中的每个节点。在循环中，函数检查当前节点的下一个节点是否是缓存的尾部，即它的下一个节点的下一个节点是否为NULL。如果是，说明找到了最久未使用的节点，函数将其从缓存中移除：将当前节点的next指针设置为NULL，断开与下一个节点的连接，然后释放该节点的内存，并返回。如果不是尾部节点，函数继续遍历，将p指针移动到下一个节点。通过这种方式，delete\_cache函数确保了缓存中总是保留最近使用的节点，同时淘汰最久未使用的节点，从而优化了缓存的使用效率。

综上所述，以上函数通过查询、更新、删除等方式，实现了将最新的消息保存在最先访问的地方。

**5.3 读取程序命令参数**

void get\_config(int argc, char\* argv[]) {

int index;

argc--;

print\_help\_info();//打印帮助信息

if (argc == 0)

{

printf("默认模式\n");

return;

}

for (index = 1; index <= argc; index++) {

if (strcmp(argv[index], "-p") == 0) {

process\_mode = 1;

printf("简单调试模式\n");

}

if (strcmp(argv[index], "-d") == 0) {

debug\_mode = 1;

printf("复杂调试模式\n");

}

if (strcmp(argv[index], "-l") == 0) {

log\_mode = 1;

printf("日志模式\n");

}

if (strcmp(argv[index], "-t") == 0) {

intercept\_mode = 1;

printf("拦截IPv6模式\n");

}

else if (strcmp(argv[index], "-s") == 0) //设置远程DNS服务器

{

char\* addr = malloc(16);

memset(addr, 0, 16);

index++;

memcpy(addr, argv[index], strlen(argv[index]) + 1);

remote\_dns = addr;

}

}

else if (strcmp(argv[index], "-s") == 0) //设置远程DNS服务器

{

char\* addr = malloc(16);

memset(addr, 0, 16);

index++;

memcpy(addr, argv[index], strlen(argv[index]) + 1);

remote\_dns = addr;

}

else if (strcmp(argv[index], "-m") == 0) //设置模式

{

index++;

if (strcmp(argv[index], "0") == 0) {

mode = 0;

}

else if (strcmp(argv[index], "1") == 0) {

mode = 1;

}

}

}

printf("运行模式: ");//阻塞模式还是非阻塞模式

printf(mode == 1 ? "非阻塞模式\n" : "阻塞模式\n");

}

}

else if (strcmp(argv[index], "-i") == 0) //打印基本信息

{

printf("本地对照表路径: %s\n", host\_path);//dnsrelay.txt的路径

printf("远程DNS服务器地址: %s (默认地址: 10.3.9.6, BUPT DNS) \n", remote\_dns);//远程DNS服务器地址

}

else if (strcmp(argv[index], "-m") == 0) //设置模式

{

index++;

if (strcmp(argv[index], "0") == 0) {

mode = 0;

}

else if (strcmp(argv[index], "1") == 0) {

mode = 1;

}

}

}

printf("运行模式: ");//阻塞模式还是非阻塞模式

printf(mode == 1 ? "非阻塞模式\n" : "阻塞模式\n");

}

get\_config() 函数的主要功能是根据命令行参数配置 DNS 中继服务器的运行模式、调试选项以及远程 DNS 服务器地址。该函数通过解析传入的参数，决定是否启用调试模式、日志模式，以及选择服务器的工作模式等配置。下面是该函数的核心思路与工作流程。

首先，函数通过 argc 和 argv 获取命令行传入的参数。argc 表示参数个数，argv 是一个字符串数组，包含了所有命令行参数。函数首先减少 argc 的值以去掉程序名，然后通过调用 print\_help\_info() 打印帮助信息，引导用户理解如何正确传入参数。

接下来，函数开始解析参数。使用一个循环遍历每个参数，并根据不同的标志（例如 -p、-d、-l）进行不同的处理：

如果参数是 -p，则启用简单调试模式，并打印相关提示。

如果参数是 -d，则启用复杂调试模式，并打印详细的调试信息。

如果参数是 -l，则启用日志模式，在程序运行期间记录客户端的查询操作。

如果参数是 -t，则启用拦截IPv6模式，即拦截相同的IPv6地址的请求

如果参数是 -i，则打印当前的基本配置信息，包括本地对照表路径、远程 DNS 服务器地址以及当前运行模式（阻塞模式或非阻塞模式）。

此外，函数还允许用户通过 -s 设置远程 DNS 服务器地址。若用户传入此参数，函数会动态分配内存并将新的 DNS 地址赋值给 remote\_dns 变量，修改服务器的查询目标。若参数为 -m，则允许用户设置服务器的工作模式，0 表示阻塞模式，1 表示非阻塞模式。

最后，函数通过 printf 输出当前的运行模式（阻塞或非阻塞），使用户在程序启动时能够看到最终的配置。

**5.4 socket通信的实现**

**5.4.1 初始化socket**

void init\_socket() {

/\* 初始化，否则无法运行socket \*/

WORD wVersion = MAKEWORD(2, 2);//设置版本号

WSADATA wsadata;

if (WSAStartup(wVersion, &wsadata) != 0)

{

return;

}

client\_sock = socket(AF\_INET, SOCK\_DGRAM, 0);

server\_sock = socket(AF\_INET, SOCK\_DGRAM, 0);

memset(&client\_addr, 0, sizeof(client\_addr));

memset(&server\_addr, 0, sizeof(server\_addr));

client\_addr.sin\_family = AF\_INET;

client\_addr.sin\_addr.s\_addr = INADDR\_ANY;

client\_addr.sin\_port = htons(PORT);

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_addr.s\_addr = inet\_addr(remote\_dns);

server\_addr.sin\_port = htons(PORT);

const int REUSE = 1;//

setsockopt(client\_sock, SOL\_SOCKET, SO\_REUSEADDR, (const char\*)&REUSE, sizeof(REUSE));

//允许在套接字处于 TIME\_WAIT 状态时重新绑定到该套接字

if (bind(client\_sock, (SOCKADDR\*)&client\_addr, addr\_len) < 0)

{

printf("ERROR: Could not bind: %s\n", strerror(errno));//如果绑定失败，输出错误信息

exit(-1);

}

char\* DNS\_server = remote\_dns;

printf("\nDNS server: %s\n", DNS\_server);//输出DNS服务器地址

printf("Listening on port 53\n\n");

}

init\_socket() 函数负责完成网络通信的底层准备工作，为后续的 DNS 查询与转发奠定基础。该函数首先调用 WSAStartup() 初始化 WinSock 库，确保程序能够在 Windows 平台上使用套接字接口。通过传入版本号 MAKEWORD(2,2)，我们明确要求使用 Winsock 2.2 标准。如果初始化失败，函数将直接返回，避免后续调用因未初始化而导致的异常。

在Winsock初始化成功后，函数接着通过两次 socket(AF\_INET, SOCK\_DGRAM, 0) 分别创建了用于客户端请求接收（client\_sock）和上游服务器通信（server\_sock）的 UDP 套接字。这里选择 UDP 协议是因为 DNS 查询本身基于无连接的 UDP 交互，能够在高并发场景中提供更低的延迟和更简洁的资源管理。

随后，函数使用 memset() 将 client\_addr 和server\_addr 两个 sockaddr\_in 结构体清零，并依次填充必要字段：

地址族：均设为 AF\_INET（IPv4）。

端口号：通过 htons(PORT) 将主机字节序的 53 端口转换为网络字节序，符合 DNS 服务的默认端口要求。

IP 地址：client\_addr 使用 INADDR\_ANY 绑定本地所有可用网卡；server\_addr 则通过 inet\_addr(remote\_dns) 将用户配置或默认的远程 DNS 服务器地址转换为网络二进制形式。

为了支持在快速重启或短时间内重新绑定同一端口，函数调用 setsockopt() 并设置 SO\_REUSEADDR 选项，使得当 client\_sock 处于 TIME\_WAIT 状态时仍能立即复用端口。这在开发调试阶段尤其有用，避免频繁重启程序导致的端口占用错误。

最后，函数通过 bind() 将 client\_sock 与本地地址结构 client\_addr 绑定，开始监听来自客户端的 DNS 查询报文；若绑定失败，立即打印错误信息并终止程序，保证后续处理的可靠性。绑定成功后，函数打印出远程 DNS 服务器地址和监听端口，帮助用户确认初始化配置无误。

**5.5 处理客户端和服务器数据交互处理**

**5.5.1 阻塞模式与非阻塞模式**

void nonblock() {

int server\_result = ioctlsocket(server\_sock, FIONBIO, &mode);// 设置非阻塞模式

int client\_result = ioctlsocket(client\_sock, FIONBIO, &mode);// 设置非阻塞模式

if (server\_result != 0 || client\_result != 0) {

// 设置失败就返回

printf("ioctlsocket failed with error: %d\n", WSAGetLastError());

closesocket(server\_sock);

closesocket(client\_sock);

return ;

}

while (1) {

receive\_client(); // 接收来自客户端的数据

receive\_server(); // 接收来自服务器的数据

}

}

void block() {

struct pollfd fds[2];//设置两个用于监视的结构体

while (1)

{

fds[0].fd = client\_sock;//设置第一个结构体的fd为客户端的socket

fds[0].events = POLLIN;//设置第一个结构体的监视事件为POLLIN，即有数据可读

fds[1].fd = server\_sock;//设置第二个结构体的fd为服务器的socket

fds[1].events = POLLIN;//设置第二个结构体的监视事件为POLLIN，即有数据可读

int ret = WSAPoll(fds, 2, 1);//监视两个socket，超时时间为1ms，阻塞等待直到有数据可读或超时

if (ret == SOCKET\_ERROR)

{

printf("ERROR WSAPoll: %d.\n", WSAGetLastError());//如果发生错误，输出错误码

}

else if (ret > 0)

{

if (fds[0].revents & POLLIN)//如果客户端socket有数据可读

{

receive\_client();//接收来自客户端的数据

}

if (fds[1].revents & POLLIN)//如果服务器socket有数据可读

{

receive\_server();//接收来自服务器的数据

}

}

}

}

nonblock() 和 block() 函数分别实现了 DNS 中继服务器中用于处理客户端和服务器数据交互的两种不同的 I/O 模式：非阻塞模式和阻塞模式。非阻塞模式下，nonblock() 函数通过调用 ioctlsocket 设置客户端和服务器的套接字为非阻塞模式，从而使得 I/O 操作不会阻塞进程，允许程序在没有数据的情况下继续执行其他任务。这对于高并发环境下尤其重要，因为它可以避免在等待数据时浪费处理资源。该函数通过不断调用 receive\_client() 和 receive\_server() 函数来接收客户端和服务器的数据，若无数据可读，函数会立即返回并继续处理其他任务。此外，函数内还包含了错误处理机制，当设置非阻塞模式失败时，会输出错误信息并关闭套接字，防止程序进入异常状态。

相较之下，block() 函数实现了阻塞模式的 I/O 操作。在这种模式下，程序会使用 WSAPoll() 来监视客户端和服务器的套接字，直到其中有一个套接字变得可读为止。在等待过程中，程序会被阻塞，这意味着 CPU 不会空转，而是等待数据到达。这种方式适用于请求量较小、并发较低的场景，因为它能够减少 CPU 资源的消耗，同时避免不必要的频繁轮询。当 WSAPoll() 检测到有数据可读时，程序会根据返回的事件信息调用 receive\_client() 或 receive\_server() 来分别处理来自客户端和服务器的数据。阻塞模式的最大优势是其实现简单，适用于数据量较小、实时性要求不高的环境，但在高并发场景下可能会导致响应延迟。

总结来说，nonblock() 函数通过非阻塞模式提高了并发处理能力，使得程序能够在高并发的情况下更高效地运行；而 block() 函数则通过阻塞模式减少了空转，适用于低负载或稳定流量的场景。

**5.5.2 客户端接收数据和从远程DNS服务器接收数据的实现**

void receive\_client() {

thread\_param\_t\* param = (thread\_param\_t\*)malloc(sizeof(thread\_param\_t));

param->addr\_len = sizeof(struct sockaddr\_in);

param->msg\_size = recvfrom(client\_sock, param->buffer, sizeof(param->buffer), 0, (struct sockaddr\*)&param->client\_addr, &param->addr\_len);

if (param->msg\_size >= 0) {

if (process\_mode == 1 || debug\_mode == 1)

{

printf("启动多线程\n");

}

\_beginthreadex(NULL, 0, handle\_client, param, 0, NULL);

}

else {

free(param);

}

}

void receive\_server()//仅为DNS中继器调用，为从远程DNS服务器接收数据

{

uint8\_t buffer[BUFFER\_SIZE]; // 接收的报文

dns\_message msg;

int msg\_size = -1;

uint8\_t addr[10][4] = { 0 }; int addr\_index=0;

if (is\_listen == 1) //若正在等待远程DNS服务器的响应

msg\_size = recvfrom(server\_sock, buffer, sizeof(buffer), 0, (struct sockaddr\*)&server\_addr, &addr\_len); //recvfrom函数用于接收数据

if (msg\_size > 0 && is\_listen == 1)//若成功接收报文且正在等待远程DNS服务器的响应{

get\_message(&msg, buffer, buffer);//解析远程DNS服务器发来的DNS报文，将其保存到msg结构体内

if (process\_mode == 1 || debug\_mode == 1)

printf("服务器已接收远程服务器查询内容%s的结果！\n", msg.questions->q\_name);

dns\_rr\* current = msg.answers;

prev\_index = 0;

for (int i = 0; i < msg.header->anCount; i++){

if (current->type == 1) { // 检查是否为A记录 (type 1 表示A记录)

update\_cache(current->rd\_data.a\_record.IP\_addr, msg.questions->q\_name);

memcpy(addr[addr\_index], current->rd\_data.a\_record.IP\_addr, sizeof(current->rd\_data.a\_record.IP\_addr));

addr\_index++;

if (intercept\_mode == 1){

memcpy(prev[prev\_index].ip\_addr, current->rd\_data.a\_record.IP\_addr, sizeof(current->rd\_data.a\_record.IP\_addr));//将IP地址保存在prev[i]中

free(prev[prev\_index].q\_name); //释放prev[i]中的q\_name

prev[prev\_index].q\_name = (char\*)malloc(strlen(msg.questions->q\_name) + 1);

memcpy(prev[prev\_index].q\_name, msg.questions->q\_name, strlen(msg.questions->q\_name) + 1);

prev\_index++;

}

}

current = current->next;

}

uint16\_t ID = msg.header->id;//获取报文结构体中的ID

uint16\_t old\_ID = htons(ID\_list[ID].client\_ID);//获取ID列表中对应ID的客户端ID

memcpy(buffer, &old\_ID, sizeof(uint16\_t)); //把待发回客户端的包ID改回原ID

struct sockaddr\_in ca = ID\_list[ID].client\_addr;//获取原ID的客户端地址

ID\_list[ID].expire\_time = 0;//把id列表中对应ID的expire\_time设置为0，表示该位置已空缺 sendto(client\_sock, buffer, msg\_size, 0, (struct sockaddr\*)&ca, addr\_len);

is\_listen = 0;//设置监听标志位为0，表示已发送完DNS应答报文，

if (debug\_mode == 1 || process\_mode == 1) {

printf("查询结果已发送回客户端，请查看！\n");

message\_type = 2;

}

if (log\_mode == 1) {

write\_log(msg.questions->q\_name, NULL,addr\_index,addr);//打印日志

}

}

}

receive\_client() 函数的主要任务是从客户端接收 DNS 查询请求，并启动一个

新的线程来处理该请求。首先，函数为每个接收到的请求分配内存空间，存储请求的相关信息（如客户端的地址和接收到的消息）。通过调用 recvfrom() 函数，receive\_client() 函数从客户端的套接字接收数据，并将数据存储在 param->buffer 中。如果接收到的数据有效（即返回的消息大小大于等于零），则说明客户端的请求已成功到达。接着，函数判断当前是否启用了调试模式或简单调试模式，如果启用，则输出调试信息，提示启动多线程处理客户端请求。随后，函数通过 \_beginthreadex() 启动一个新线程来调用 handle\_client() 函数，该函数负责进一步处理客户端的请求。若接收数据失败，函数则释放分配的内存资源，以避免内存泄漏。

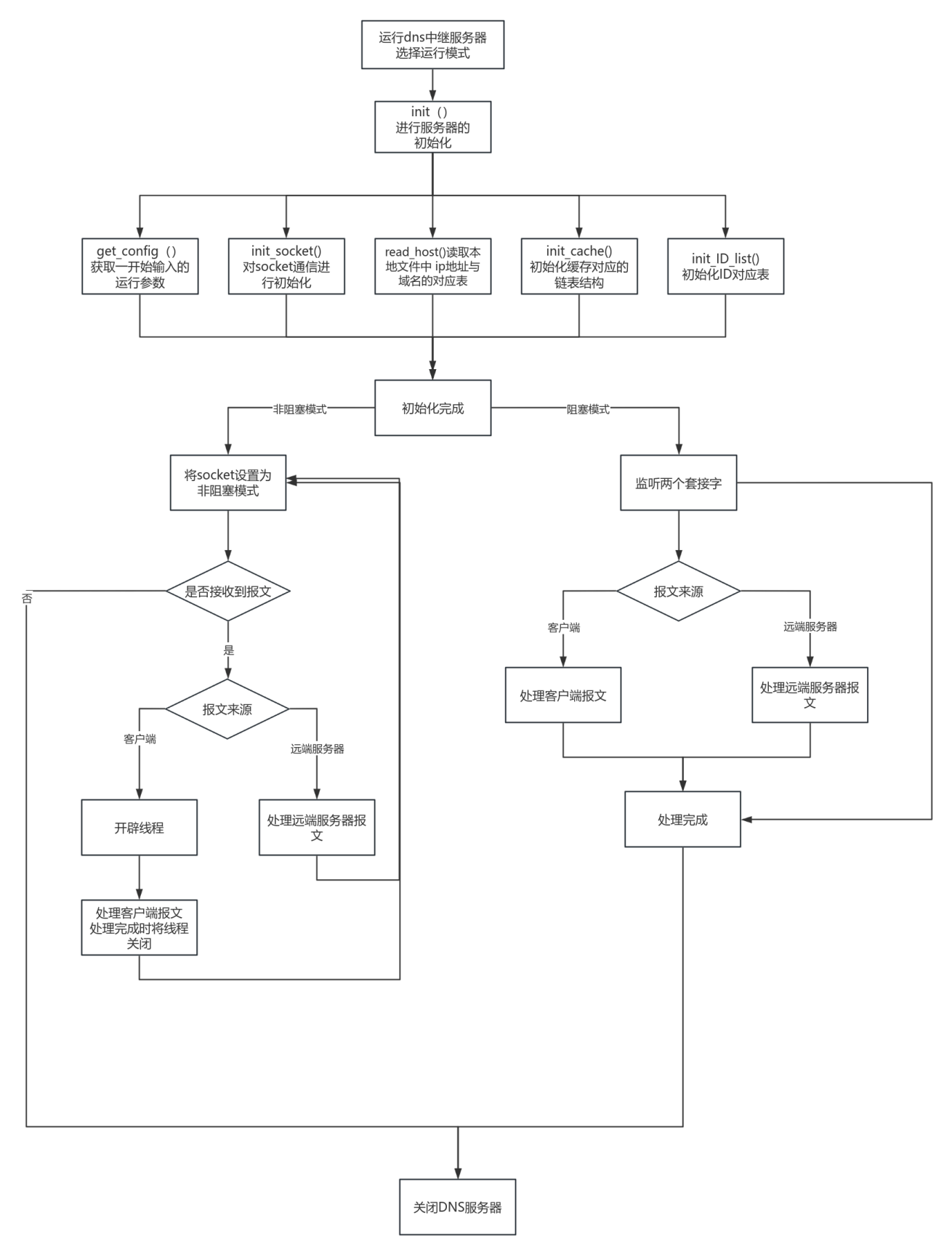
receive\_client() 函数在多线程环境下运行，使得每个客户端请求都能独立处理，避免了单线程情况下的性能瓶颈，能够有效地支持多个客户端并发访问。

receive\_server() 函数的功能是从远程DNS服务器接收响应报文，并将响应返回给相应的客户端。首先，函数检查当前是否正在等待远程DNS服务器的响应（即 is\_listen == 1）。如果是，它通过 recvfrom() 函数从远程DNS服务器的套接字接收响应报文。成功接收到数据后，函数会调用 get\_message() 解析DNS报文，获取报文中的相关信息，并将解析后的数据存储在 msg 结构体中。接着，函数通过循环遍历DNS响应中的所有答案记录，寻找类型为 A（即IPv4地址记录）的条目，将有效的 IP 地址缓存到本地，并更新相应的缓存记录。

然后，函数从响应报文中获取 DNS 请求的 ID，并将该 ID 转换为客户端的原始 ID，通过修改报文中的 ID 字段，将响应数据发送回客户端。发送完成后，is\_listen 标志被设置为 0，表示已完成响应，停止等待远程DNS服务器的响应。

在调试模式下，函数会输出相关的调试信息，告知查询结果已成功返回客户端。同时，如果启用了日志模式，函数还会记录查询日志，包含查询的域名、返回的 IP 地址等信息，以便后续审计和分析。

**6 软件流程图**

****

**6.1初始化阶段**

服务器启动后，首先执行 init() 函数完成各种模块的准备工作。它依次调用 get\_config() 从配置文件或命令行参数中读取运行时选项；调用 init\_socket() 在指定端口上创建并绑定 UDP（或 TCP）的 socket；通过 read\_host() 从本地 hosts 文件或数据库中加载域名—IP 地址的映射；执行 init\_cache() 初始化缓存数据结构，用于后续快速查找；最后调用 init\_ID\_list() 生成本地事务 ID 管理表，用以跟踪并发请求。所有子模块初始化完毕后，服务器进入“初始化完成”状态，开始选择运行模式。

**6.2 运行阶段**

**6.2.1 非阻塞模式**

如果选择了“非阻塞”模式，服务器会将 socket 切换到非阻塞方式，并进入一条主循环：在每次轮询中尝试调用 recvfrom() 接收数据包。如果当前没有可读数据，立即跳回循环开头；若成功接收到报文，则根据报文来源分流——如果来自客户端，就在内部新建一个线程去处理该查询请求，线程负责向上游服务器转发、收集应答并回写给客户端；如果来自远程上游服务器，则在主线程中直接调用应答处理函数，将返回结果写回对应的客户端套接字。客户端请求处理完毕后，相应线程关闭并回收资源，主循环继续侦听下一次报文。

**6.2.2 阻塞模式**

若选择“阻塞”模式，服务器在初始化后调用 select()或 poll()监视两个文件描述符集：一个对应客户端请求的 socket，另一个对应上游服务器的 socket。每当 select() 返回时，检查就绪的描述符：若是客户端 socket 可读，则在主线程中调用客户端请求处理逻辑；若是上游服务器 socket 可读，则调用上游应答处理逻辑。处理完任一报文后，立即返回 select()，继续等待下一个事件。这样可以在单线程中按事件驱动方式顺序处理所有 DNS 请求与应答。

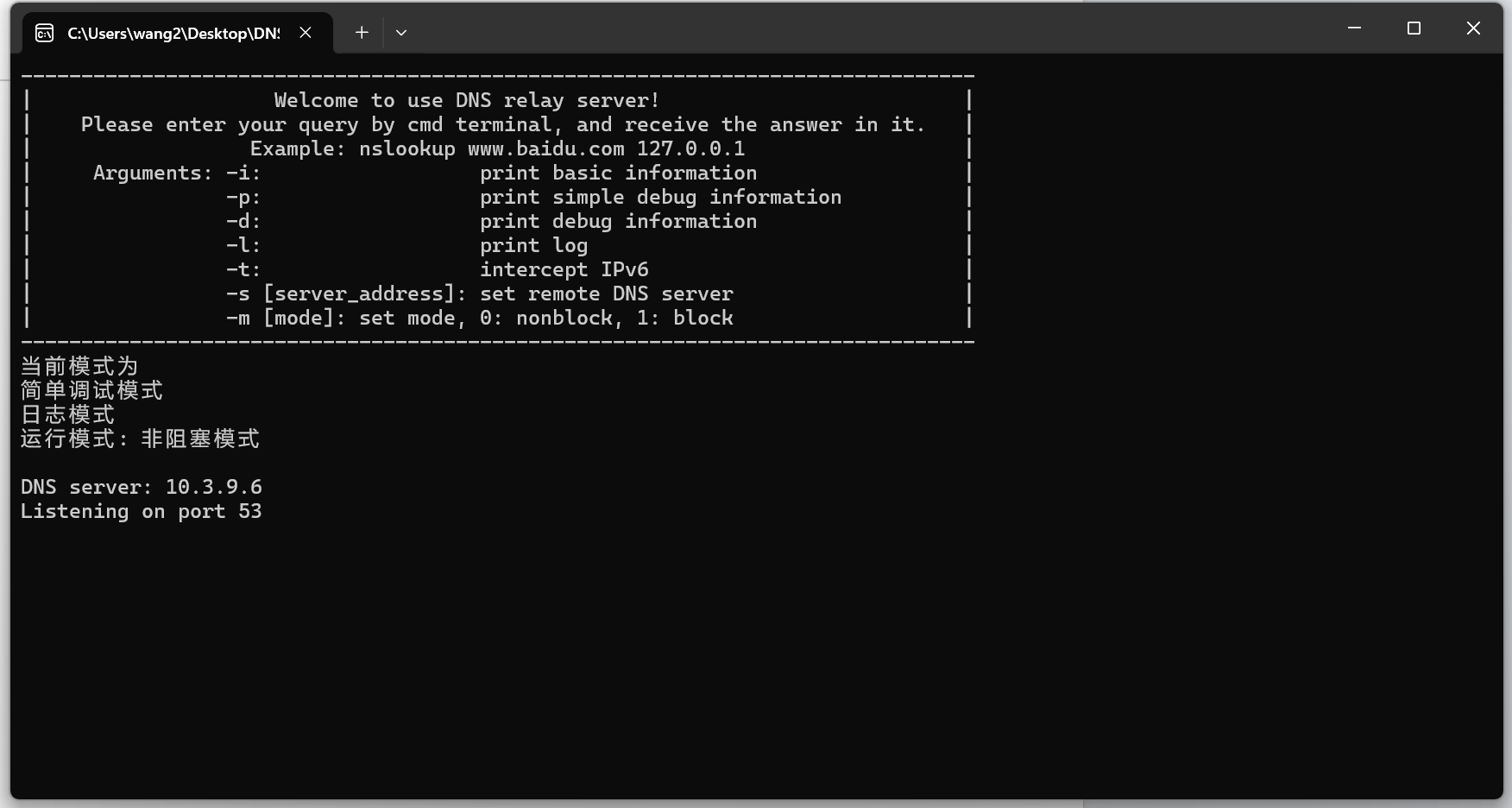
**6.3 关闭阶段**

当接收到退出信号时，服务器会跳出主循环，依次关闭所有打开的 socket、释放缓存与 ID 管理表等资源，最终干净利落地退出，完成 DNS 中继服务器的整个生命周期管理。

**7 测试用例以及运行结果**

**①不屏蔽IPv6模式**

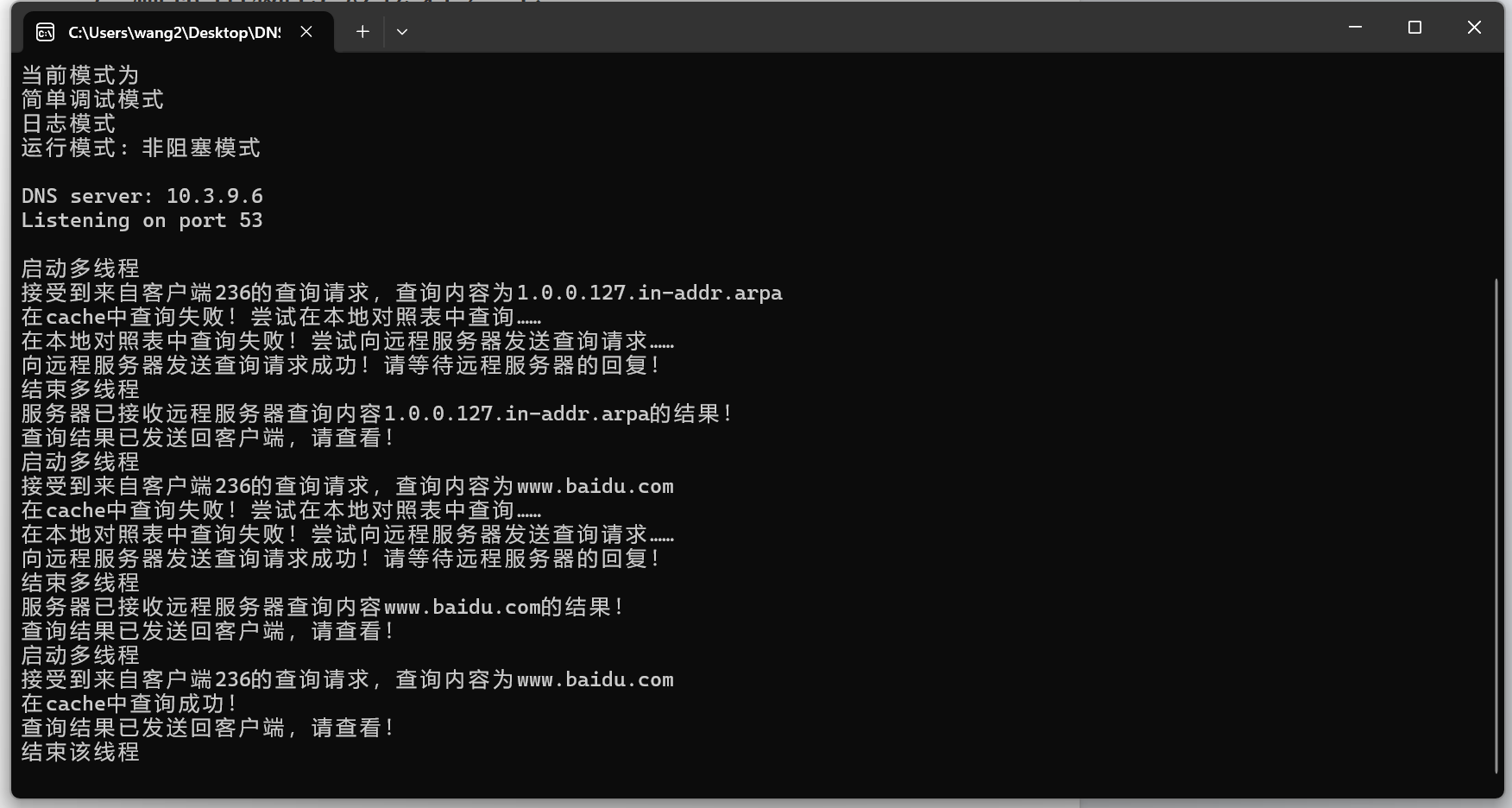
在命令行界面输入./main.exe -p -l，进入简单调试模式和日志模式

****

服务器初始页面

1、验证中继功能

启动系统的命令行界面输入nslookup [www.baidu.com](http://www.baidu.com) 127.0.0.1，回车运行



服务器处理输入nslookup [www.baidu.com](http://www.baidu.com) 127.0.0.1页面

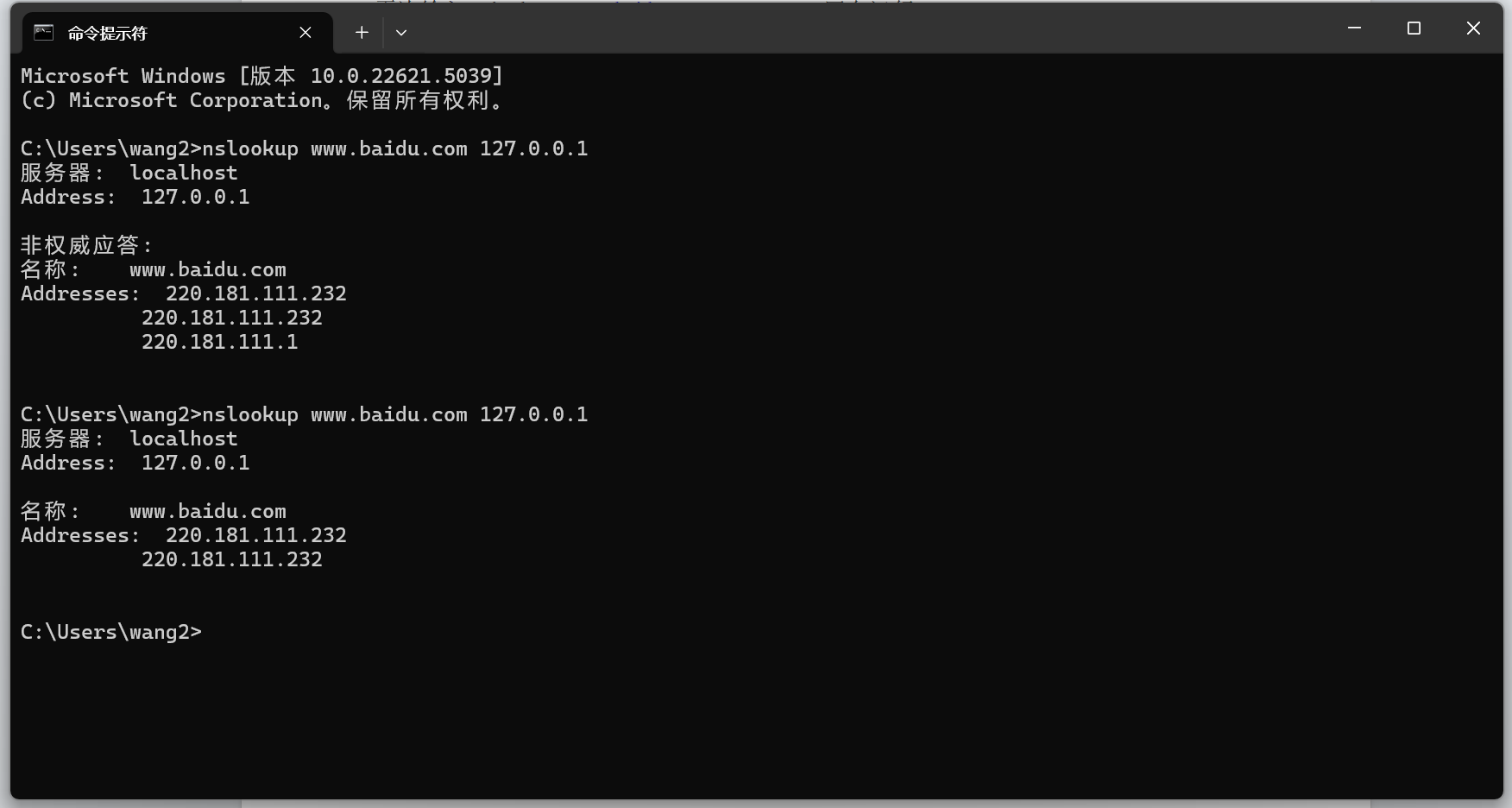
****

客户端输出页面

从服务器和客户端输出可知当输入nslookup [www.baidu.com](http://www.baidu.com) 127.0.0.1时，服务器会接受3次查询，分别查询1.0.0.127.in-addr.arpa的IP地址，www.baidu.com的IPv4地址，www.baidu.com的IPv6地址。由于缓存和本地均未储存1.0.0.127.in-addr.arpa对应的IP地址，故服务器向远程服务器查询，得到服务器的名称为localhost；同样的当接收到www.baidu.com的IPv4地址的查询请求时，由于缓存和本地均未储存www.baidu.com对应的IP地址，故服务器向远程服务器查询，得到220.181.111.232和220.181.111.1两个IP地址，同时将这两个IP地址存入缓存；当接收到www.baidu.com的IPv6地址查询请求时，由于缓存中已存在www.baidu.com对应的IP地址，故直接返回。

****再次输入nslookup [www.baidu.com](http://www.baidu.com) 127.0.0.1，回车运行

服务器处理再次输入nslookup [www.baidu.com](http://www.baidu.com) 127.0.0.1页面

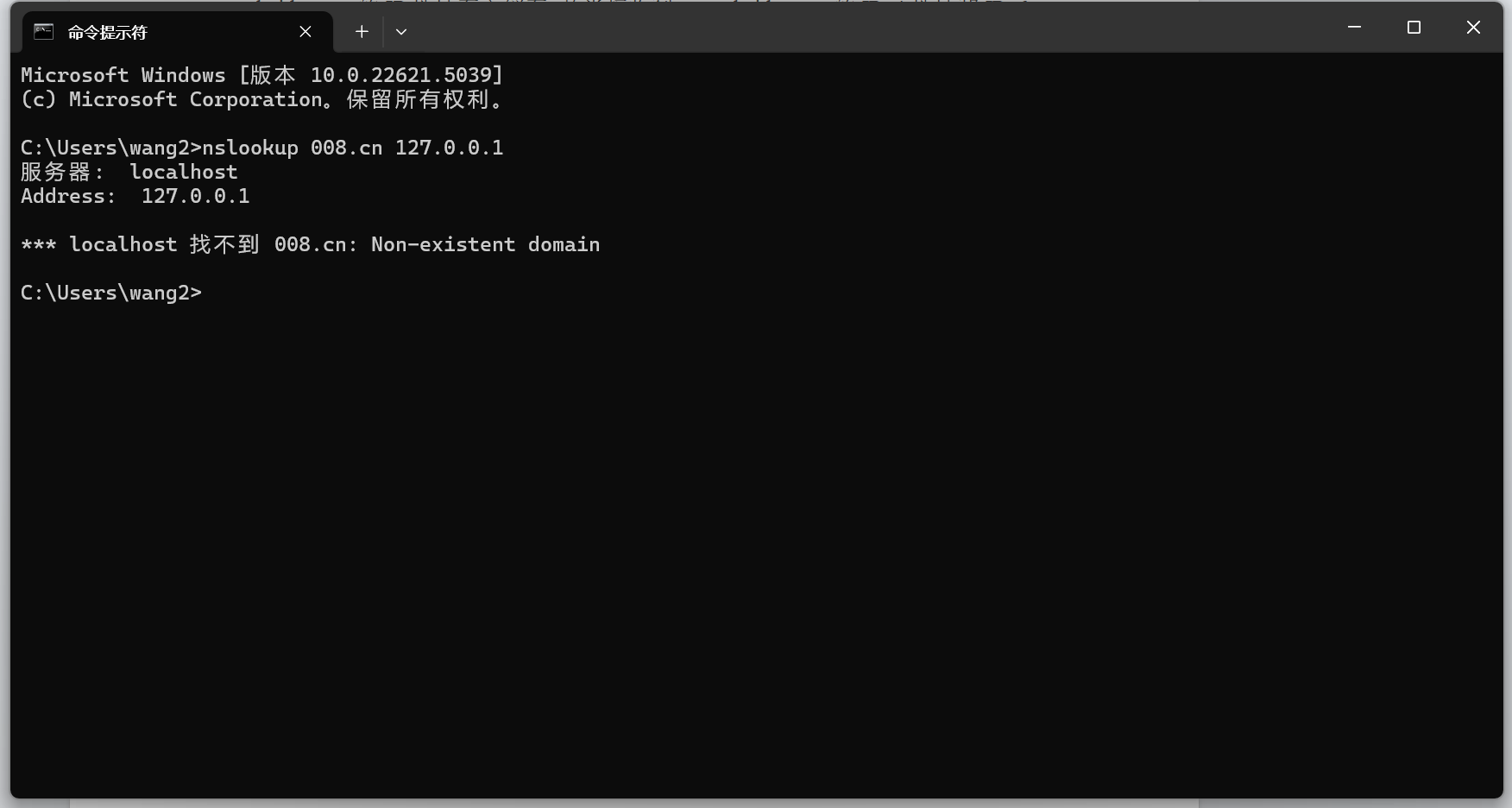
****

客户端输出页面

从服务器和客户端输出可知在第二次查询中，因为在第一次查询中已经将www.baidu.com的IP地址存入缓存，故当接收到www.baidu.com的IPv4地址和IPv6地址查询请求时，服务器直接返回缓存中的查询结果。

2、验证不良网站拦截功能

启动系统的命令行界面输入nslookup 008.cn 127.0.0.1，回车运行

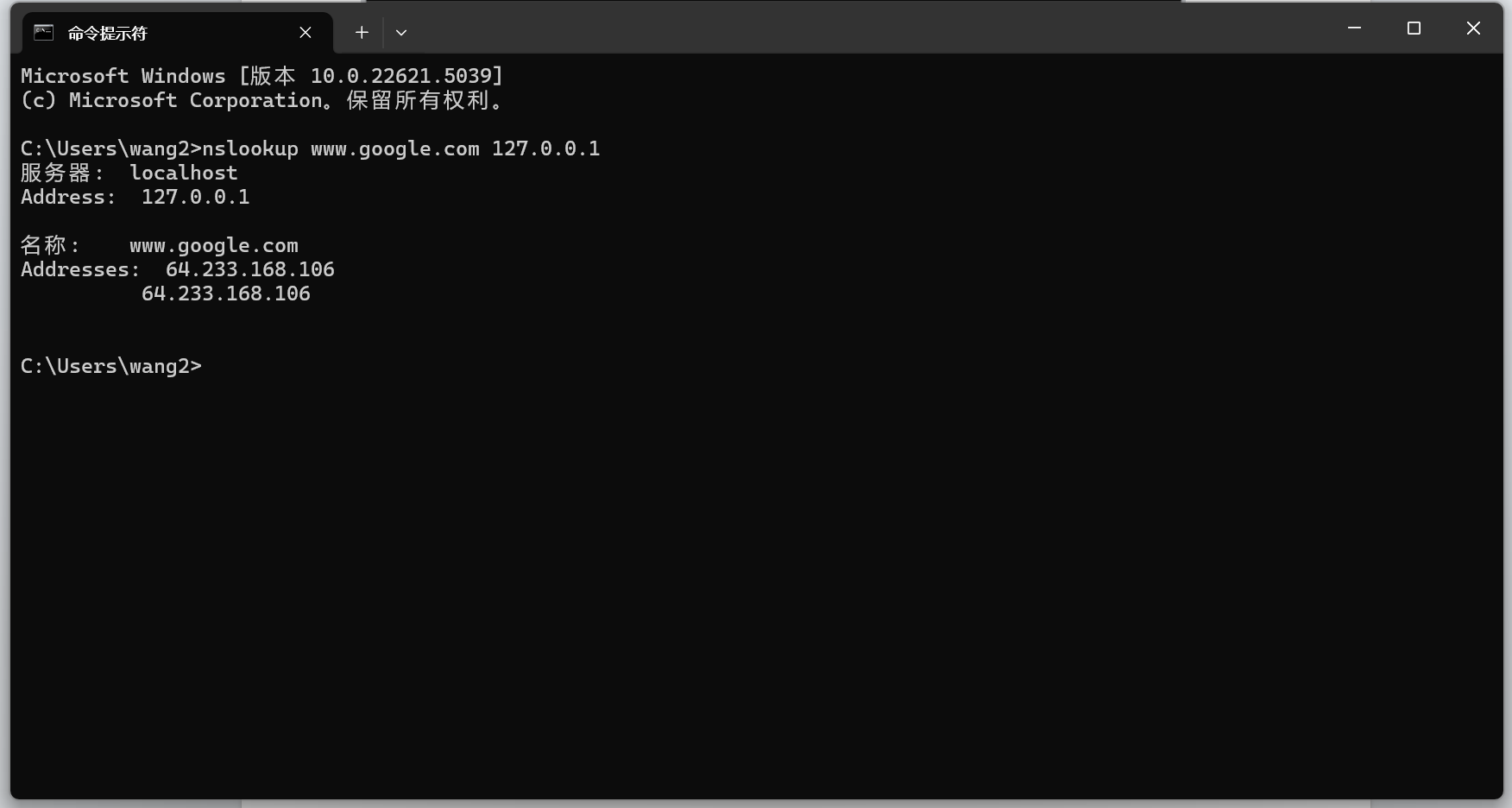


客户端输出页面

从客户端输出可知服务器实现了拦截的操作

3、验证本地服务器实现功能

启动系统的命令行界面输入nslookup www.google.com 127.0.0.1，回车运行

****

客户端输出页面

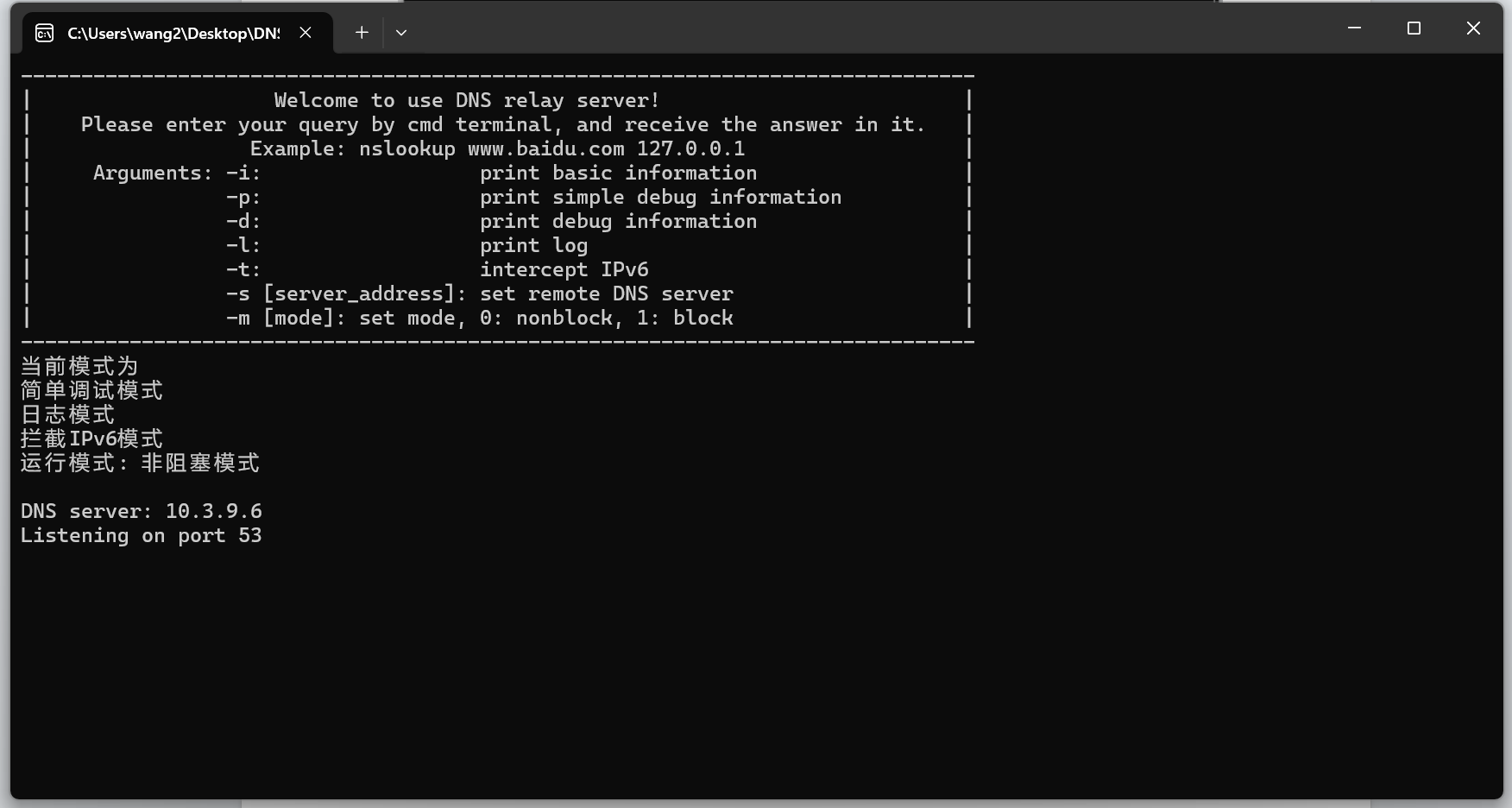


服务器处理输入nslookup [www.google.com](http://www.baidu.com) 127.0.0.1页面

从客户端输出可知服务器实现了查询的操作，同理IPv4地址先未在缓存中查到在本地查到，之后加入缓存，查询IPv6地址时直接在缓存中查询到

**②屏蔽IPv6模式**

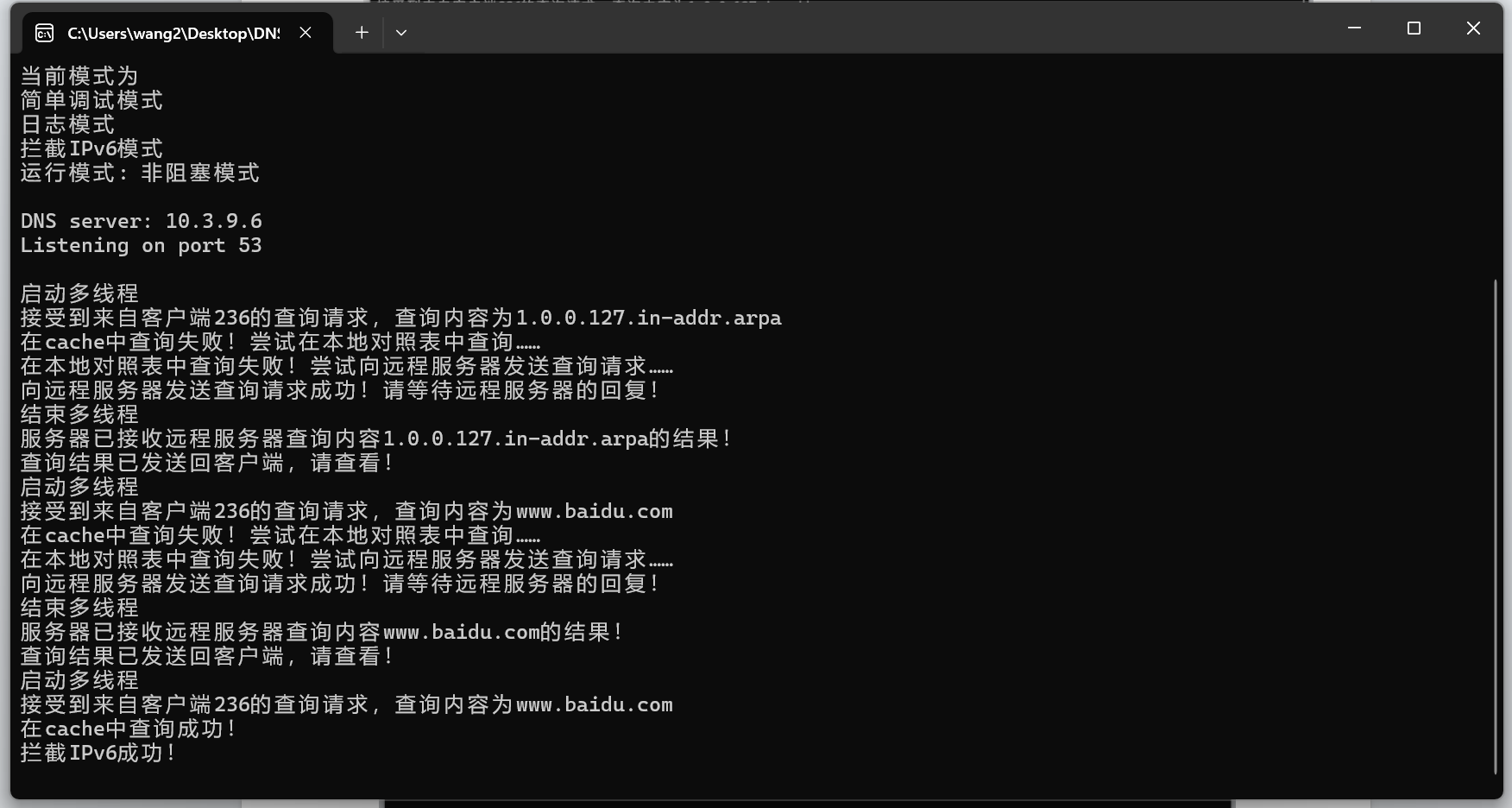
在命令行界面输入./main.exe -p -l -t，进入简单调试模式，日志模式和屏蔽IPv6模式

****

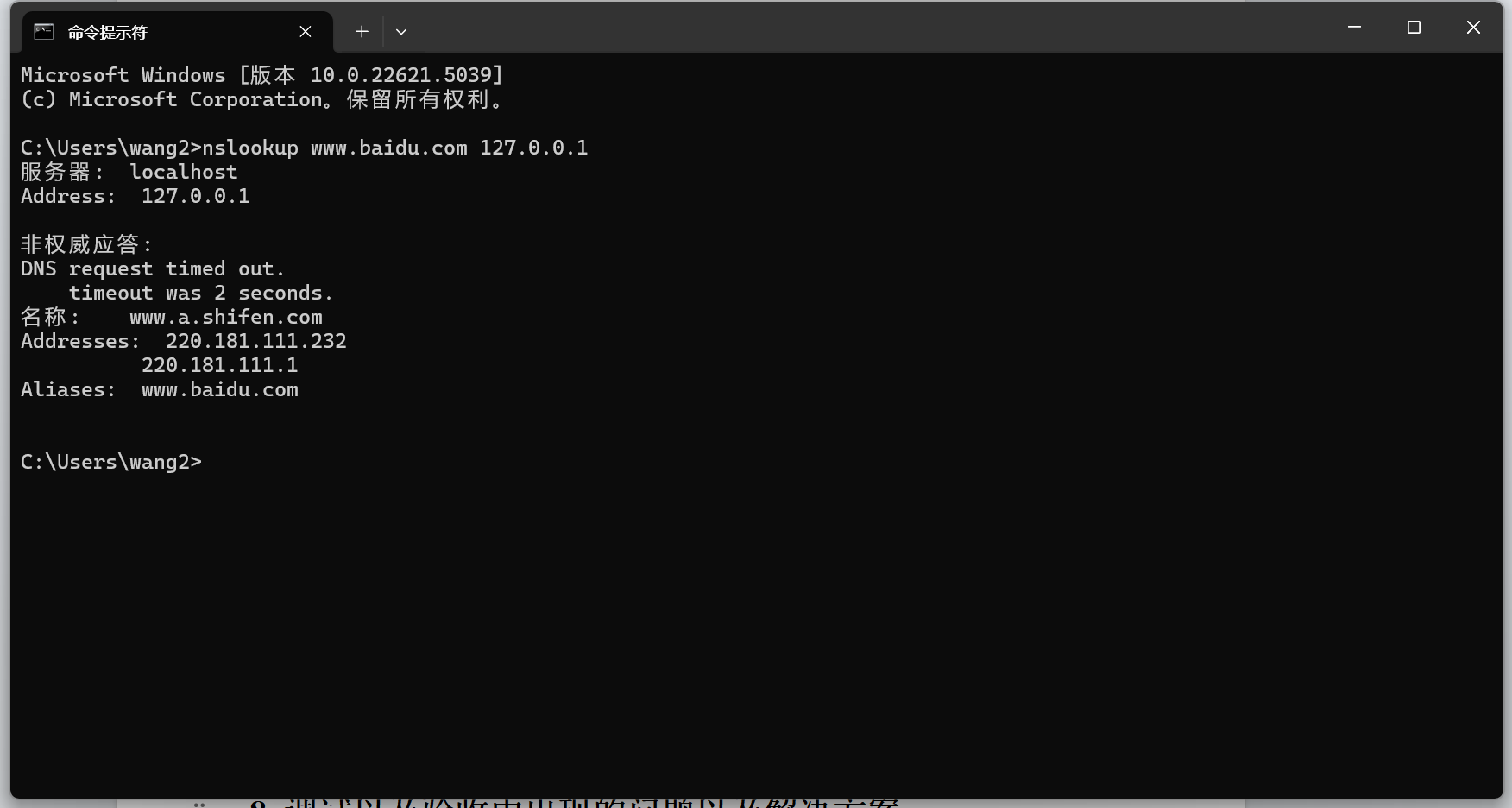
服务器初始页面

1、验证中继功能

启动系统的命令行界面输入nslookup [www.baidu.com](http://www.baidu.com) 127.0.0.1，回车运行

****

服务器处理输入nslookup [www.baidu.com](http://www.baidu.com) 127.0.0.1页面

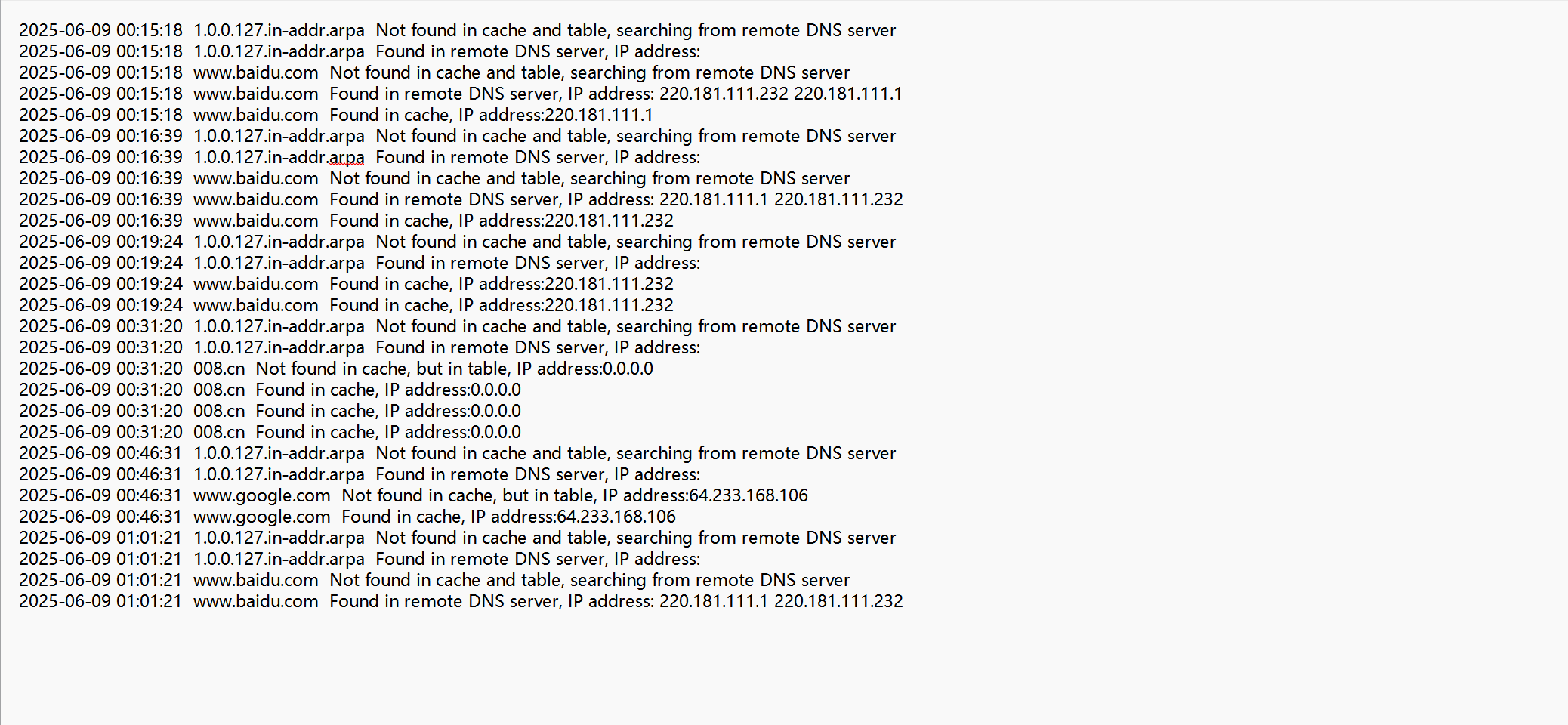
****

客户端输出页面

从服务器和客户端输出可知当输入nslookup [www.baidu.com](http://www.baidu.com) 127.0.0.1时，服务器处理过程和返回客户端输出功能大体与不屏蔽IPv6模式类似，但当接收到www.baidu.com的IPv6地址查询请求时，服务器先在缓存中查询是否有IPv4对应的内容，若有则不输出对应内容。

由于屏蔽IPv6模式下其他功能都与不屏蔽模式下类似，仅仅是屏蔽了IPv6查询内容，故验证其他功能在这里不再演示。

**③服务器日志输出**

****

**8 调试以及验收中出现的问题以及解决方案**

**1、阻塞和非阻塞模式设置错误：**

一开始我们设计阻塞和非阻塞模式时使用一个全局变量int mode，设定非阻塞模式下mode为0，阻塞模式下mode为1，将非阻塞模式函数中调用的ioctlsocket中的u\_long \* argp参数设置为了&mode，结果在测试非阻塞模式中的多线程功能时。发现并没有正确地调用，通过逐过程调试我们发现mode的值设置错误，非阻塞模式下mode应该为1，阻塞模式下mode应该为0，在更改后调用成功。

**2、cache缓存问题：**

当服务器向远端服务器查询时，一个网站的ipv4地址和ipv6地址通常有两个，这四个地址都会返回，但我们的程序一开始只处理单ip地址的报文，导致从远程服务器得到查询结果时无法正确插入缓存。在详细梳理了DNS报文结构和cache缓存的处理方法，我们优化了cache的插入方法，最后实现了将收到的多个ip地址存入缓存的功能。

**3、接收报文问题：**

当远端服务器传回DNS报文时，我们程序有时候在处理输入时会突然中断，调试器显示存在空指针，经过我们逐过程调试发现在receive\_server函数即处理从远程服务器传来的报文时，将DNS报文解析的函数get\_message调用时未判断recvfrom函数返回的DNS报文的长度是否大于0，导致当接收数据发生错误时直接调用get\_message函数导致出错。

**4、ID转换问题：**

我们原本在处理ID问题时使用的生成随机数的方法，即把客户端传递的DNS报文的ID作为种子生成随机数作为向远程服务端发送DNS报文的ID，但这样会导致ID重复的问题，虽然概率比较小，但我们测试的过程中，还是出现了一次重复的情况，于是我们决定引入储存客户端的ID与地址的结构体数组，当需要向远程服务器端发送信息时，将客户端的ID与地址存入该结构体数组，并将数组的角标作为新ID进行转换，这样的好处在于可以同时追踪该转发的发送时间，若超时则将该信息删除。

**5、忙等待问题：**

在服务器诸多功能的实现中，最让我们头疼的就是如何解决忙等待问题。我们一开始实现的思路是一直循环执行receive\_client函数与receive\_server函数，这样虽然能保证能始终获取到新客户端的加入以及输入的信息，但是会让程序一直循环执行不必要的内容，之后我们优化了函数，使用WinSock2.h库中的recvfrom函数，这个函数在接收到实际的DNS报文才会返回内容，这使得程序的CPU占用低了不少

**9 总结以及心得体会**

本次实验以实现 DNS 中继服务器为核心目标，完整地演练了从需求分析、模块化设计、功能实现到性能测试的全过程，在实践中我们深入体会了网络编程与并发模型的精髓。

回顾整个 DNS 中继服务器的开发历程，我们团队在两周内累计投入了近四十个小时的代码编写与调试，另花费数小时对报告中的各项模块细节进行梳理和撰写，真正体会到了一个网络服务从“无到有”、再到“稳定可用”的完整演进。

在项目伊始阶段，面对陌生的DNS协议和网络编程领域，团队通过查阅大量文献和技术学习，逐步构建起完整的知识体系。从Socket通信的基础原理到DNS报文结构的解析，从单线程阻塞模型到多线程并发架构的演进，每一个技术难点都被转化为宝贵的学习契机。

在开发的过程中，通过将DNS报文处理、缓存管理、网络通信等核心功能解耦，团队实现了并行开发与快速迭代。

测试环节同样充满启发，团队采用模拟多客户端使用、错误输入等测试方式来验证系统的稳定性。在此过程中，日志系统和性能监控工具的引入极大提升了问题定位效率。此外，团队还使用了Wireshark抓包工具去抓取数据包并分析其中的含义，加深了团队对网络数据包格式的理解。

综上所述，这份设计报告不仅记录了项目成果，更承载着团队协作的智慧结晶。从需求分析时的激烈讨论，到调试阶段的相互支持，每个环节都展现了成员间的技术默契。在撰写报告过程中，我们通过梳理模块、设计流程图，将零散的技术细节升华为系统的知识图谱。这段开发经历证明，只有经过完整的"设计-实现-优化"闭环，才能真正领悟网络服务的核心原理，这种认知提升远比代码本身更具长远价值。