北京邮电大学课程设计报告

<table>
<tr><td>课程设计名称</td><td colspan="2">计算机网络课程设计</td><td>学 院</td><td>计算机</td><td>指导教师</td><td>程莉</td></tr>
<tr><td>班 级</td><td>班内序号</td><td colspan="2">学 号</td><td>学生姓名</td><td colspan="2">成绩</td></tr>
<tr><td>305</td><td></td><td colspan="2">2022211683</td><td>张晨阳</td><td colspan="2"></td></tr>
<tr><td>305</td><td></td><td colspan="2">2022211637</td><td>廖轩毅</td><td colspan="2"></td></tr>
<tr><td>305</td><td></td><td colspan="2">2022211121</td><td>刘晟毅</td><td colspan="2"></td></tr>
<tr><td>课程设计内容</td><td colspan="6">设计一个 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）
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）
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）
除此之外，还有自己设计的高级功能。
团队分工：
张晨阳：实现报文结构设计、解析转换，实现调试日志输出，整合代码并测试优化，优化报告。
刘晟毅：实现查询、中继功能，实现 socket 通信，实现超时、并发、更新 cache 功能。
廖轩毅：测试程序代码，分析模块结构，撰写设计报告。</td></tr>
<tr><td>学生课程设计报告
(附页)</td><td colspan="6"></td></tr>
<tr><td>课程设计成绩评定</td><td colspan="6">评语：

成绩：

指导教师签名：
年 月 日</td></tr>
</table>

注：评语要体现每个学生的工作情况，可以加页。

目录

1 开发环境

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

2 功能设计

2.1 基本功能

2.1.1 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。

2.1.2 不良网站拦截

采用分布式防火墙策略，系统将所有不良网站的 IP 地址重定向到本地，并将其归为无效 IP(即 0.0.0.0)，实现不良网站的拦截。

2.1.3 服务器功能

作为中继服务器，系统实现了中继转发和缓存功能。接收来自客户端的 DNS 查询请求，首先在本地缓存和字典树中查找，若没有命中则向远程 DNS 服务器查询，并将结果返回给客户端。

2.1.4 多客户端并发处理

系统支持多个客户端的并发查询，通过处理消息 ID 的转换以区分不同客户端的请求。

2.1.5 中继功能

为隐藏内网结构，系统实现了客户端查询 ID 和内网设备 ID 之间的转换。客户端查询时使用客户端 ID，中继服务器内部使用内网 ID 进行查询。

2.1.6 超时处理

考虑到 UDP 的不可靠性，系统实现了超时处理机制，以应对外部 DNS 服务器无响应或响应延迟的情况。

2.2 高级功能

2.2.1 Cache 缓存功能

维护最近 DNS 查询记录及对应的结果缓存。在接收到新的 DNS 查询时，首先检查缓存。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。缓存使用 LRU 置换策略管理缓存条目。

2.2.2 字典查询算法优化

采用字典树查询算法，提高查询速度。

2.2.3 输出调试信息和 DNS 报文内容

系统提供了调试信息输出功能，以便开发者或用户能够跟踪和解决运行中的问题。

3 模块划分

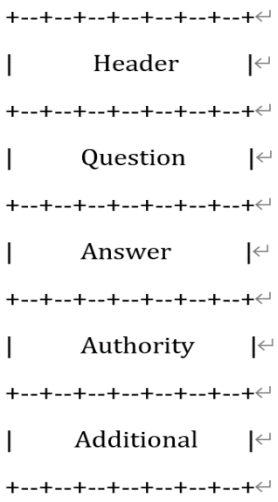
本系统共分为七个模块，分别为 DNS_struct，DNS_print，DNS_convert，DNS_config，DNS_server，DNS_cache，DNS_Trie。

3.1 DNS_struct

本模块实现了 DNS 报文的基本结构并定义了 DNS 报文中的常量值。

3.1.1 DNS 报文格式

DNS 报文格式如下：



其中，各字段定义如下：

Header：报文头，固定 12 字节，由结构体 DNS_header 存储。

Question：向域名服务器的查询请求，由结构体 DNS_question 存储。

之后的三个部分有相同的格式，即 Resource Record(RR)，并且都可能为空，由 DNS_resource_record 结构储存；

Answer：对于查询问题的回复。

Authority：指向授权域名服务器。

Additional：附加信息。

结构体定义如下：

```
// DNS 报文结构体
typedef struct DNS_mes {
    Dns_Header* header;
    Dns_Question* question;
    Dns_rr* answer;
    Dns_rr* authority;
    Dns_rr* additional;
} Dns_Mes;
```

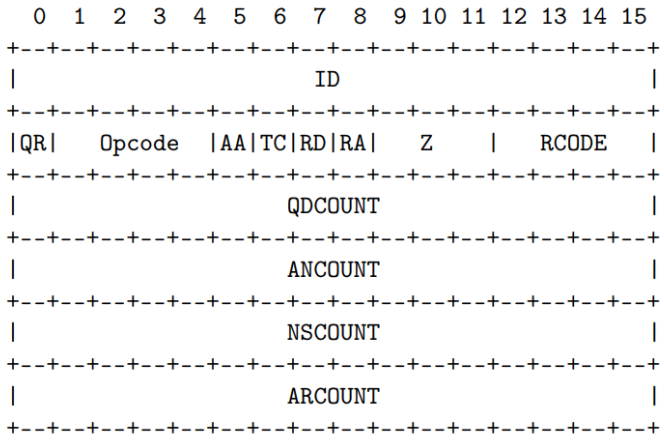
3.1.2 DNS 报文中的常量值

```
// DNS 报文的最大长度
#define DNS_STRING_MAX_SIZE 8192

// DNS 资源记录中域名的最大长度
#define DNS_RR_NAME_MAX_SIZE 512

// DNS 查询或响应标志，0 表示查询，1 表示响应
#define DNS_QR_QUERY 0
#define DNS_QR_ANSWER 1

// DNS 操作码，表示查询类型
// 0 表示标准查询，1 表示反向查询，2 表示服务器状态请求
#define DNS_OPCODE_QUERY 0
#define DNS_OPCODE_IQUERY 1
#define DNS_OPCODE_STATUS 2

// DNS 资源记录类型，表示不同类型的 DNS 记录
#define DNS_TYPE_A 1       // A 记录，表示 IPv4 地址
#define DNS_TYPE_NS 2      // NS 记录，表示权威名称服务器
#define DNS_TYPE_CNAME 5   // CNAME 记录，表示规范名称
#define DNS_TYPE_SOA 6     // SOA 记录，表示起始授权机构
#define DNS_TYPE_PTR 12    // PTR 记录，表示指针记录
#define DNS_TYPE_HINFO 13  // HINFO 记录，表示主机信息
#define DNS_TYPE_MINFO 14  // MINFO 记录，表示邮件信息
```

```
#define DNS_TYPE_MX 15     // MX 记录，表示邮件交换记录
#define DNS_TYPE_TXT 16    // TXT 记录，表示文本记录
#define DNS_TYPE_AAAA 28   // AAAA 记录，表示 IPv6 地址

// DNS 类，表示地址类型，通常为 1，表示因特网
#define DNS_CLASS_IN 1

// DNS 响应代码，表示查询的返回状态
// 0 表示无错误，3 表示名字错误
#define DNS_RCODE_OK 0
#define DNS_RCODE_NXDOMAIN 3
```

3.1.3 结构体 DNS_header

Header 报文部分格式如下：



其中，各字段定义如下：

ID：标识符，用于标识查询和响应的唯一 ID。

QR：查询/响应标志，为 0 表示查询，为 1 表示响应。

Opcode：操作码，表示查询类型。为 0 表示标准查询（QUERY），为 1 表示反向查询（IQUERY），为 2 表示服务器状态请求（STATUS）。

AA：权威回答标志，在回复报文中有效。为 1 表示该响应由权威服务器生成。

TC：截断标志。为 1 表示消息由于超过长度而被截断。

RD：期望递归标志，在查询报文中设置。为 1 表示客户端希望服务器递归查询。

RA：可用递归标志，在回复报文中设置。为 1 表示服务器支持递归查询。

Z：保留字段。必须为 0。

RCODE：返回码，表示查询的返回状态。为 0 表示无错误，为 1 表示查询格式错误，为 2 表示服务器错误，为 3 表示域名错误（仅在权威服务器的回复中有意义，指查询中请

求的域名不存在），为 4 表示查询的类型不受支持，为 5 表示服务器拒绝处理请求。

QDCOUNT：问题数目，表示查询部分的问题条目数。

ANCOUNT：回答数目，表示回答部分的回答条目数。

NSCOUNT：权威记录数目，表示权威记录部分的记录条目数。

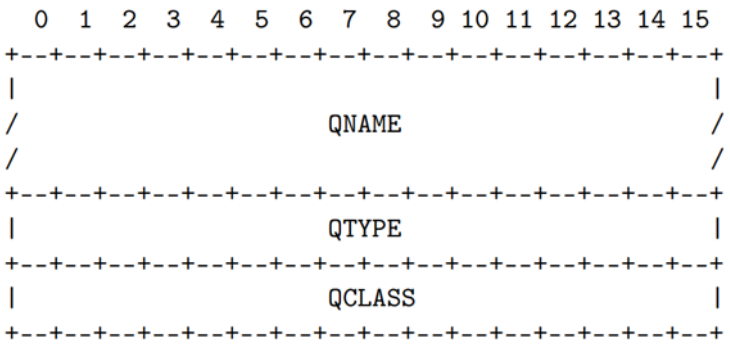
ARCOUNT：附加记录数目，表示附加记录部分的记录条目数。

结构体定义如下：

```
// DNS 报文 Header 结构体
typedef struct DNS_header {
    uint16_t id;           // 标识符，用于标识查询和响应的唯一 ID

    // 标志位字段，位域表示
    uint8_t qr : 1;        // 查询/响应标志，0 表示查询，1 表示响应
    uint8_t opcode : 4;    // 操作码，表示查询类型：0=标准查询（QUERY），1=反向查询（IQUERY），2=服务器状态请求（STATUS）
    uint8_t aa : 1;        // 权威回答标志，1 表示该响应由权威服务器生成
    uint8_t tc : 1;        // 截断标志，1 表示消息由于超过长度而被截断
    uint8_t rd : 1;        // 期望递归标志，1 表示客户端希望服务器递归查询
    uint8_t ra : 1;        // 可用递归标志，1 表示服务器支持递归查询
    uint8_t z : 3;         // 保留字段，必须为 0
    uint8_t rcode : 4;     // 返回码，表示查询的返回状态：0=无错误，1=格式错误，2=服务器错误，3=名字错误，4=未实现，5=拒绝

    // 计数器字段
    uint16_t qdcount;      // 问题数目，表示查询部分的问题条目数
    uint16_t ancount;      // 回答数目，表示回答部分的回答条目数
    uint16_t nscount;      // 权威记录数目，表示权威记录部分的记录条目数
    uint16_t arcount;      // 附加记录数目，表示附加记录部分的记录条目数
} Dns_Header;
```

3.1.4 结构体 DNS_question

Question 报文部分格式如下：



其中，各字段定义如下：

QNAME：查询的域名或 IP 地址。

QTYPE：查询类型。

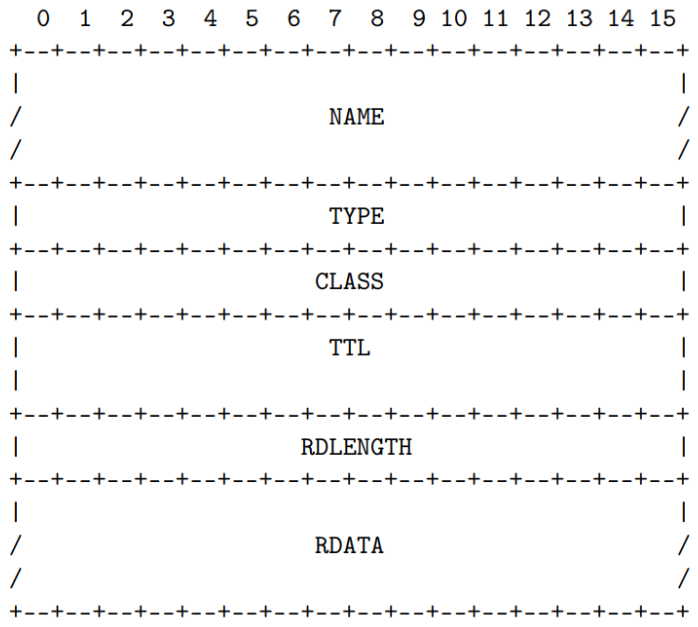
QCLASS：查询类别。

结构体定义如下：

```
// 报文 Question 结构体
typedef struct DNS_question {
    char* q_name;           // 查询的域名或 IP 地址，使用字符串指针表示
    uint16_t q_type;        // 查询类型，如 A（IPv4 地址记录）、MX（邮件交换记录）、CNAME（规范名称记录）等
    uint16_t q_class;       // 查询类，通常为 1（表示 IN，因特网）
    struct DNS_question* next;      // 指向下一个查询问题的指针，用于链表结构
} Dns_Question;
```

3.1.5 结构体 DNS_resource_record

Resource Record 报文部分格式如下：



其中，各字段定义如下：

NAME：此 RR 从属的域名或主机名。

TYPE：资源数据类型，表示该记录的类型。

CLASS：资源数据类。

TTL：生存时间（Time to Live），表示该记录在缓存中的有效时间（秒）。

RDLENGTH：资源数据长度，以字节为单位，表示 rdata 字段的长度。

RDATA：资源数据内容，根据 type 字段的不同，存储不同类型的资源数据。

结构体定义如下：

```
// 报文 Resource Record 结构体
typedef struct DNS_resource_record {
    char* name;                         // 域名或主机名，使用字符串指针表示
    uint16_t type;                      // 资源数据类型，表示该记录的类型，如 A、AAAA、CNAME、MX 等
    uint16_t rr_class;                  // 资源数据类，通常为 1，表示 IN（因特网）
    uint32_t ttl;                       // 生存时间（Time to Live），表示该记录在缓存中的有效时间（秒）
    uint16_t rd_length;                 // 资源数据长度，以字节为单位，表示 rdata 字段的长度
    union ResourceData rd_data;         // 资源数据内容，根据 type 字段的不同，存储不同类型的资源数据
    struct DNS_resource_record* next;   // 指向下一个资源记录的指针，用于形成链表结构
} Dns_rr;
```

其中，ResourceData 联合体定义如下：

```
// 定义 ResourceData 联合体，用于存储不同类型的资源数据
union ResourceData {
    // A 记录 (IPv4 地址)
    struct {
        uint8_t IP_addr[4];      // 4 字节的 IPv4 地址
    } a_record;

    // AAAA 记录 (IPv6 地址)
    struct {
        uint8_t IP_addr[16];     // 16 字节的 IPv6 地址
    } aaaa_record;

    // SOA 记录 (权威记录的起始)
    struct {
        char* MName;             // 主服务器域名
        char* RName;             // 管理员邮箱
        uint32_t serial;         // 版本号
        uint32_t refresh;        // 刷新数据间隔
        uint32_t retry;          // 重试间隔
        uint32_t expire;         // 超时重传时间
        uint32_t minimum;        // 最小生存时间
    } soa_record;

    // CNAME 记录 (规范名称)
    struct {
        char* cname;             // 规范名称
    } cname_record;

    // MX 记录 (邮件交换)
    struct {
        uint16_t preference;     // 优先级，值越小优先级越高
        char* exchange;          // 邮件交换服务器域名
    } mx_record;
```

```
    // TXT 记录 (文本记录)
    struct {
        char* text;             // 文本内容
    } txt_record;
};
```

3.2 DNS_print

本模块实现了打印 DNS 报文（字节流、结构体）的接口定义。

```
// 打印 DNS 报文字节流
void print_dstring(char* pstring, unsigned int length);

// 打印 header
void print_header(Dns_Mes* msg);

// 打印 question
void print_question(Dns_Mes* msg);

// 打印 answer（RR）
void print_answer(Dns_Mes* msg);
```

其中，print_dstring 函数用于打印 DNS 报文字节流；print_header 函数用于打印 header 报文段；print_question 函数用于打印 question 报文段；print_answer 函数用于打印 answer（RR）报文段。

3.3 DNS_convert

本模块实现了 DNS 报文结构体与字串的相互转换的接口定义。

```
// 用于获取 DNS 报文头各值的掩码
static const uint32_t QR_MASK = 0x8000;
static const uint32_t OPCODE_MASK = 0x7800;
static const uint32_t AA_MASK = 0x0400;
static const uint32_t TC_MASK = 0x0200;
static const uint32_t RD_MASK = 0x0100;
static const uint32_t RA_MASK = 0x0080;
static const uint32_t RCODE_MASK = 0x000F;

// DNS 报文字串转换为 DNS 结构体
void string_to_dnsstruct(Dns_Mes* pmsg, uint8_t* buffer, uint8_t* start);

// 读取指定位数
size_t read_bits(uint8_t** buffer, int bits);

// 从字串获取 header
uint8_t* get_dnsheader(Dns_Mes* msg, uint8_t* buffer);

// 从字串获取 question
uint8_t* get_dnsquestion(Dns_Mes* msg, uint8_t* buffer, uint8_t* start);

// 从字串获取 answer
uint8_t* get_dnsanswer(Dns_Mes* msg, uint8_t* buffer, uint8_t* start);

// 从字串获取域名 Name
uint8_t* get_domain(uint8_t* buffer, char* name, uint8_t* start);

// DNS 结构体转换为字串
uint8_t* dnsstruct_to_string(Dns_Mes* pmsg, uint8_t* buffer, uint8_t* ip_addr);

// 写入指定位数
void write_bits(uint8_t** buffer, int bits, int value);

// 将 header 写入字串
uint8_t* set_dnsheader(Dns_Mes* msg, uint8_t* buffer, uint8_t* ip_addr);

// 将 question 写入字串
uint8_t* set_dnsquestion(Dns_Mes* msg, uint8_t* buffer);

// 将 answer 写入字串
uint8_t* set_dnsanswer(Dns_Mes* msg, uint8_t* buffer, uint8_t* ip_addr);
```

```
// 设置域名
uint8_t* set_domain(uint8_t* buffer, char* name);

// 释放空间
void free_message(Dns_Mes* msg);
```

其中，string_to_dnsstruct 函数用于将 DNS 报文字串转换为 DNS 结构体；read_bits 函数用于读取指定位数；get_dnsheader 函数用于从字串中获取 header 报文段；get_dnsquestion 函数用于从字串中获取 question 报文段；get_dnsanswer 函数用于从字串中获取 answer 报文段；get_domain 函数用于从字串中获取域名 Name；dnsstruct_to_string 函数用于将 DNS 结构体转换为字串；write_bits 函数用于写入指定位数；set_dnsheader 函数用于将 header 报文段写入字串；set_dnsquestion 函数用于将 question 报文段写入字串；set_dnsanswer 函数用于将 answer 报文段写入字串；set_domain 函数用于设置域名；free_message 函数用于释放空间。

其中，read_bits 函数如下所示：

```
size_t read_bits(uint8_t** buffer, int bits) {
    if (bits == 8) {
        uint8_t val;
        memcpy(&val, *buffer, 1);
        *buffer += 1;
        return val;
    }
    if (bits == 16) {
        uint16_t val;
        memcpy(&val, *buffer, 2);
        *buffer += 2;
        return ntohs(val); // 网络字节序转换为主机字节序
    }
    if (bits == 32) {
        uint32_t val;
        memcpy(&val, *buffer, 4);
        *buffer += 4;
        return ntohl(val); // 网络字节序转换为主机字节序
    }
}
```

write_bits 函数如下所示：

```
// 将指定数量的位（8, 16, 32）设置到缓冲区中
void write_bits(uint8_t** buffer, int bits, int value) {
    if (bits == 8) {
```

```
        **buffer = (uint8_t)value;
        (*buffer)++;
    }
    if (bits == 16) {
        uint16_t val = htons((uint16_t)value);
        memcpy(*buffer, &val, 2);
        *buffer += 2;
    }
    if (bits == 32) {
        uint32_t val = htonl(value);
        memcpy(*buffer, &val, 4);
        *buffer += 4;
    }
}
```

由于主机字节顺序和网络字节顺序不同，我们需要在 read_bits 函数中使用 ntohl 函数和 ntohs 函数，在 write_bits 函数中使用 htonl 函数和 htons 函数。其中，ntohl 函数用于将一个 32 位整数从网络字节顺序转换为主机字节顺序。ntohs 函数用于将一个 16 位整数从网络字节顺序转换为主机字节顺序。htonl 函数用于将一个 32 位整数从主机字节顺序转换为网络字节顺序，htons 函数用于将一个 16 位整数从主机字节顺序转换为网络字节顺序。

这四个函数均已在 WinSock2.h 中定义。

3.4 DNS_config

本模块实现了命令行参数解析及日志输出。

```
char IPAddr[DNS_RR_NAME_MAX_SIZE];
char domain[DNS_RR_NAME_MAX_SIZE];
char* host_path;  // HOST 文件目录
char* LOG_PATH;   // 日志文件目录

int debug_mode;
int log_mode;

// 初始化函数，用于设置程序的运行环境
void init(int argc, char* argv[]);

// 读取命令行参数，设置程序的配置选项
void get_config(int argc, char* argv[]);

// 打印帮助信息，展示如何使用程序以及支持的参数
void print_help_info();

// 写入日志，记录域名查询的结果
void write_log(char* domain, uint8_t* ip_addr);

// 读取本地 HOST 文件
void read_host();

// 解析本地 HOST 文件中的 IP 地址和域名
void get_host_info(FILE* ptr);
```

其中，init 函数用于初始化程序的运行环境；get_config 函数用于读取命令行参数，设置程序的配置选项；print_help_info 函数用于打印帮助信息，展示如何使用程序以及支持的参数；write_log 函数用于写入日志，记录域名查询的结果；read_host 函数用于读取本地 HOST 文件；get_host_info 函数用于解析本地 HOST 文件中的 IP 地址和域名。

3.5 DNS_server

本模块实现了 DNS 服务器的基本功能逻辑和接口定义。

```
int socketMode;          // 阻塞/非阻塞模式
int clientSocket;        // 客户端 socket
int serverSocket;        // 服务端 socket
struct sockaddr_in clientAddress;
struct sockaddr_in serverAddress;
int addressLength;

int clientPort;          // 客户端端口号
char* dnsServerAddress;   // 远程主机（BUPT 的 DNS 服务器）

int islisten;

void initializeSocket();
void closeSocketServer();
void setNonBlockingMode();
void setBlockingMode();
void receive_client();
void receive_server();
```

其中，initializeSocket 函数用于 DNS 服务器 socket 的初始化；closeSocketServer 函数用于关闭 DNS 服务器的 socket 并清理 Winsock；setNonBlockingMode 函数用于将 socket 设置为非阻塞模式；setBlockingMode 函数用于将 socket 设置为阻塞模式；receive_client 函数用于接收来自客户端的数据并进行处理；receive_server 函数用于接收来自远程 DNS 服务器的响应并转发给客户端。

receive_client 函数如下所示：

```
// 接收来自客户端的数据并进行处理
void receive_client() {
    uint8_t buffer[BUFFER_SIZE];      // 接收的报文
    uint8_t buffer_new[BUFFER_SIZE];  // 回复给客户端的报文
    Dns_Mes msg;                  // 报文结构体
    uint8_t ip_addr[4] = { 0 };         // 查询域名得到的 IP 地址
    int msg_size = -1;               // 报文大小
    int is_found = 0;                // 是否查到

    msg_size = recvfrom(clientSocket, buffer, sizeof(buffer), 0, (struct sockaddr*)&clientAddress,
&addressLength);
    if (debug_mode==1)
        printf(BLUE"received for one message from client!\n\n" RESET);
```

```
    if (msg_size >= 0) {
        uint8_t* start = buffer;
        /* 解析客户端发来的 DNS 报文，将其保存到 msg 结构体内 */
        string_to_dnsstruct(&msg, buffer, start);

        /* 从缓存查找 */
        is_found = cache_query(ip_addr,msg.question->q_name);

        /* 若 cache 未查到，则从 host 文件查找 */
        if (is_found == 0) {
            if (debug_mode == 1)
                printf(RED"Address not found in cache.  Try to find in local.\n\n" RESET);

            is_found = query_node(msg.question->q_name, ip_addr);

            /* 若未查到，则上交远程 DNS 服务器处理*/
            if (is_found == 0) {
                /* 给将要发给远程 DNS 服务器的包分配新 ID */
                uint16_t newID = reset_id(msg.header->id, clientAddress);
                memcpy(buffer, &newID, sizeof(uint16_t));
                if (newID == ID_LIST_SIZE) {
                        printf("ID list is full.\n\n");
                } else {
                    islisten = 1;
                    sendto(serverSocket, buffer, msg_size, 0, (struct sockaddr*)&serverAddress,
addressLength);
                    if (debug_mode == 1)
                        printf(RED"Address not found in local.  Send to Remote Server.\n\n" RESET);
                }
                return;
            }
        }

        uint8_t* end;
        end =dnsstruct_to_string(&msg, buffer_new, ip_addr);

        int len = end - buffer_new;

        /* 将 DNS 应答报文发回客户端 */
        sendto(clientSocket, buffer_new, len, 0, (struct sockaddr*)&clientAddress, addressLength);

        if (log_mode == 1) {
            write_log(msg.question->q_name, ip_addr);
        }
    }
```

```
}
```

当接受到来自客户端的数据时，解析其发来的 DNS 报文并将其保存到 msg 结构体内。首先在 cache 中查找。若 cache 命中，则将 DNS 应答报文发回客户端；若未命中，则在 host 文件中查找。若在 host 文件中命中，则将 DNS 应答报文发回客户端；若未命中，则上交远程 DNS 服务器处理。最后将 DNS 应答报文发回客户端。

islisten 是一个标记，用于判断在 receive_client 函数中是否上交远程 DNS 服务器处理。默认值为 0，只有当 cache 未命中且 host 文件未命中，上交远程 DNS 服务器处理时设为 1。

receive_server 函数如下所示：

```
// 接收来自远程 DNS 服务器的响应并转发给客户端
void receive_server() {
    uint8_t buffer[BUFFER_SIZE];  // 接收的报文
    Dns_Mes msg;
    int msg_size = -1;  // 报文大小

    /* 接受远程 DNS 服务器发来的 DNS 应答报文 */
    if (islisten == 1) {
        msg_size = recvfrom(serverSocket, buffer, sizeof(buffer), 0, (struct
sockaddr*)&serverAddress, &addressLength);
        if (debug_mode == 1)
            printf(BLUE"received for one message from remote server!\n\n" RESET);
        string_to_dnsstruct(&msg, buffer, buffer);
    }

    /* 将 DNS 应答报文转发回客户端 */
    if (msg_size > 0 && islisten == 1) {
        /* ID 转换 */
        uint16_t ID = msg.header->id;
        uint16_t old_ID = htons(ID_list[ID].user_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_address;
        ID_list[ID].expire_time = 0;

        sendto(clientSocket, buffer, msg_size, 0, (struct sockaddr*)&clientAddress, addressLength);
        islisten = 0;

        if (log_mode == 1) {
            write_log(msg.question->q_name, NULL);
        }
    }
}
```

当 islisten 不为 1，即从未上交远程 DNS 服务器处理时，该函数不做任何操作。否则接受远程 DNS 服务器发来的 DNS 应答报文，并将其转发回客户端。

在 receive_client 函数和 receive_server 函数中均需要超时检测。使用到的相关功能在 DNS_ResetID.h 中实现。

DNS_ResetID.h 如下所示：

```
#define MAX_ID_SIZE 200   //ID 映射表大小
#define ID_EXPIRE_TIME 4  // ID 过期时间

typedef struct {
    uint16_t user_id;          // 用户 ID
    time_t expire_time;        // 过期时间
    struct sockaddr_in client_address; // 客户端地址
} ClientSession;

ClientSession ID_list[MAX_ID_SIZE];  // 存储客户端会话信息的数组

uint16_t reset_id(uint16_t user_id, struct sockaddr_in client_address);
void init_ID_list();
```

在 ID_list 数组中，每个元素代表了一个客户端会话的信息，均包含以下字段：

user_id：用户 ID，用于标识特定的客户端会话。

expire_time：用户 ID 的过期时间。当当前时间超过了这个时间戳时，表示该用户 ID 可以被重新分配。

client_address：客户端地址信息。

init_ID_list 函数用于初始化 ID 列表。它将每个元素的 user_id 设置为 0，expire_time 设置为 0，并清空 client_address 字段。该函数实现如下：

```
void init_ID_list() {
    for (int i = 0; i < MAX_ID_SIZE; i++)
    {
        ID_list[i].user_id = 0;
        ID_list[i].expire_time = 0;
        memset(&(ID_list[i].client_address), 0, sizeof(struct sockaddr_in));
    }
}
```

reset_id 函数用于分配一个新的用户 ID 给指定的客户端地址。它首先获取当前时间，然后遍历 ID 列表，寻找过期的 ID。如果找到，它会更新该 ID 为新的用户 ID，并设置新的过期时间。如果没有找到过期的 ID，它会返回一个错误代码 UINT16_MAX。该函数实

现如下：

```
uint16_t reset_id(uint16_t user_id, struct sockaddr_in client_address) {
    uint16_t i;
    time_t current_time = time(NULL); // 只调用一次 time() 函数，避免多次系统调用
    for (i = 0; i < MAX_ID_SIZE; i++) {
        if (ID_list[i].expire_time < current_time) { // 检查 ID 是否已过期
            ID_list[i].user_id = user_id;
            ID_list[i].client_address = client_address;
            ID_list[i].expire_time = ID_EXPIRE_TIME + current_time;
            break; // 在找到第一个过期 ID 并分配新值后终止循环
        }
    }
    if (i == MAX_ID_SIZE) {
        return UINT16_MAX; // 如果没有可用的 ID，返回最大值表示失败
    }
    return i; // 返回分配 ID 的索引位置
}
```

3.6 DNS_cache

本模块实现了 DNS 缓存系统。

本模块采用了 LRU（Least Recently Used）替换算法，即最近最少使用算法，是一种常用的缓存置换策略。它的核心思想是：如果一个数据在最近一段时间内没有被访问过，那么在将来它被访问的可能性也很小。因此，当缓存达到其容量上限时，LRU 算法会优先淘汰那些最长时间未被访问的数据项。

LRU 算法是通过链表实现的，插入的时间复杂度是 O(1)，查找和删除的时间复杂度是 O(n)，由于 Cache 的容量设置不大，Cache 运行效率是相当高的。对于这个链表，最近被访问的记录会被移到链表头；当链表长度达到上限时，优先移出链表尾的节点，即最久未使用的值最先被移出缓存。

```
//思想：用有序链表实现 LRU，越老放越后面
typedef struct nodes {
          uint8_t IP[4];                    // 十进制 IP 地址,四个十进制数
          char domain[DNS_RR_NAME_MAX_SIZE];  //域名
          struct nodes* next;
}lru_cache;

lru_cache* head;
lru_cache* tail;

// 初始化 cache
void init_cache();

// 在 cache 中查询某个域名
int cache_query(uint8_t* ipv4, char* domain);

// 插入新的域名及 ip
void insert_cache(uint8_t ipv4[4], char* domain);

//删除最老的域名及 ip
void delete_node();
```

其中，init_cache 函数用于初始化 cache；cache_query 函数用于在 cache 中查询某个域名；insert_cache 函数用于插入新的域名及 IP；delete_node 函数用于删除最老的域名及 IP。

3.7 DNS_Trie

本模块实现了一个基于字典树的数据结构，用于存储和查询域名与 IPv4 地址之间的映射关系。

字典树（Trie），又称为前缀树或查找树，是一种用于存储字符串集合的数据结构。它允许快速检索信息，特别是用于自动补全、拼写检查和 IP 路由等场景。主要思想是利用字符串的公共前缀来节约存储空间。很好地利用了串的公共前缀，节约了存储空间。字典树主要包含两种操作：插入和查找。

例如，如何用字典树存储单词"abc",“abb”,“bca”,"bc"？过程如下：

初始时仅有 root 节点。

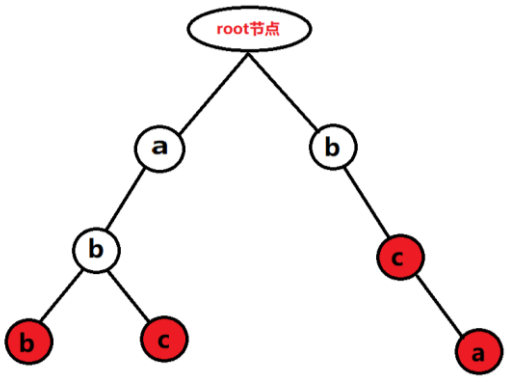
读入单词"abc"。初始时处于 root 节点。逐字分析该单词：a 不属于当前节点的子节点，则在当前节点加入子节点 a，并将该子节点作为当前节点。b 不属于当前节点的子节点，则在当前节点加入子节点 b，并将该子节点作为当前节点。c 不属于当前节点的子节点，则在当前节点加入子节点 c。

读入单词"abb"。初始时处于 root 节点。逐字分析该单词：a 属于当前节点的子节点，则将子节点 a 作为当前节点。b 属于当前节点的子节点，则将子节点 b 作为当前节点。b 不属于当前节点的子节点，则在当前节点加入子节点 b。

读入单词"bca"。初始时处于 root 节点。逐字分析该单词：b 不属于当前节点的子节点，则在当前节点加入子节点 b，并将该子节点作为当前节点。c 不属于当前节点的子节点，则在当前节点加入子节点 c，并将该子节点作为当前节点。a 不属于当前节点的子节点，则在当前节点加入子节点 a。

读入单词" bc "。初始时处于 root 节点。逐字分析该单词：b 属于当前节点的子节点，则将子节点 b 作为当前节点。c 属于当前节点的子节点，则将子节点 c 作为当前节点。

最终字典树如图所示：



在图中，红点代表有一个以此节点为终点的单词。

本模块中 Trie 树的基本性质：

1、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。

2、从根节点到某一个节点，路径上经过的字符连接起来，为该节点对应的域名。

3、每个节点的所有子节点包含的字符互不相同。

4、两个有公共前缀的关键字，在 Trie 树中前缀部分的路径相同。

5、仅在叶子节点中存储该节点对应域名的 IP 地址。

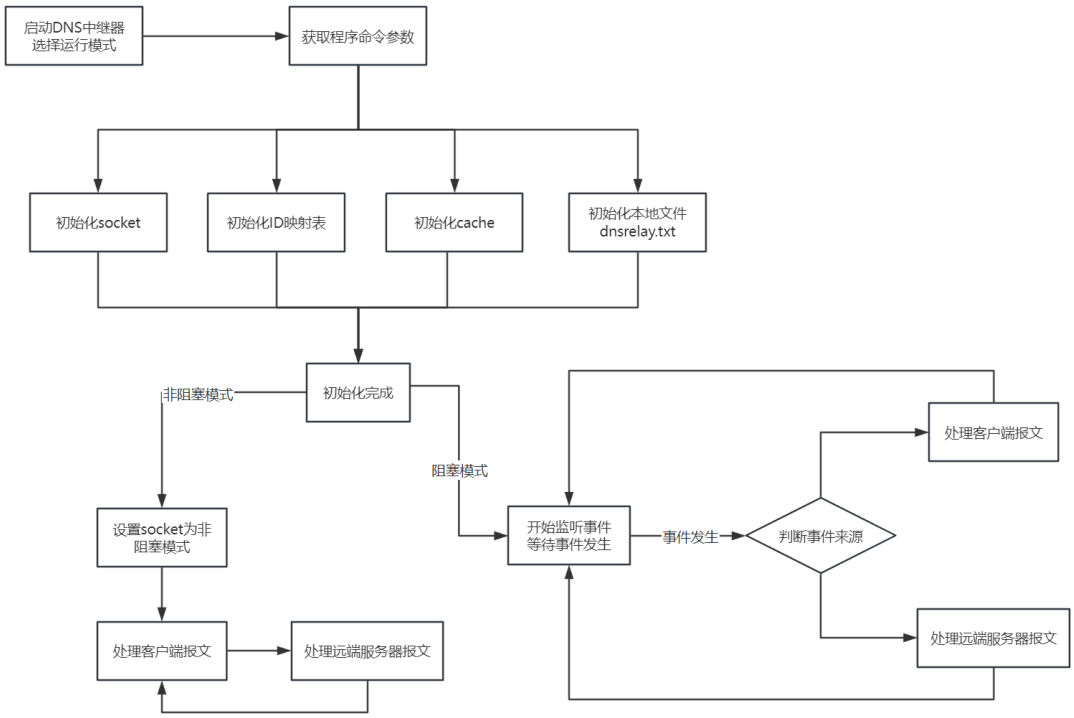
6、通过 isEnd 标志判断该节点是否为一个域名的结束节点，是则对应的 IP 地址有效。

```
typedef struct node
{
    uint8_t IP[4];      // 存储 IPv4 地址,四个十进制数
    uint16_t val[37];   // 域名中的每个字符的编号 num，val[num]存放读入某字符时，该结点的编号,一个一个字符去对应,0-9 对应 0-9,a-z 对应 10-35,.对应 36,-对应 37
    uint8_t isEnd;      // 标志节点是否为一个域名的结束节点，是则对应的 IP 地址有效
    uint16_t father;    // 存储父节点下标，字典树可能是多叉树，但是每个节点只能有一个父亲
}trie_node; // 字典树节点的结构体定义

trie_node Trie[MAX_SIZE];   // 字典树的节点数组声明
int list_size;          // 字典树当前节点数量

/**
 * @brief 提取字符串形式的 IPv4 地址，转换为整数数组
 *
 * @param IP 存储转换后的 IP 地址的数组
 * @param address 字符串形式的 IPv4 地址，如"192.168.1.1"
 */
void TranIP(uint8_t* IP, char* address);

/**
 * @brief 根据字符获取其对应的编号
 *
 * @param val 输入字符的 ASCII 值
 * @return int 字符对应的编号，0-9 返回 0-9，a-z 返回 10-35，'.'返回 36，'-'返回 37，未知字符返回-1
 */
int getnum(uint8_t val);

/**
 * @brief 向字典树中插入域名和对应的 IP 地址
 *
 * @param IP IPv4 地址的数组表示
```

```
 * @param domain 要插入的域名字符串
 */
void insert_node(uint8_t* IP, char* domain);

/**
 * @brief 查询字典树中是否存在指定的域名，并输出其对应的 IP 地址
 *
 * @param domain 要查询的域名字符串
 * @return int 如果存在该域名，返回 1；否则返回 0
 */
int query_node(char* domain,uint8_t* ip_addr);
```

其中，TranIP 函数用于将字符串形式的 IP 地址转换为整数数组形式存储；getnum 函数用于将域名中的字符转换为对应的数值；insert_node 函数用于向字典树中插入一个域名和对应的 IPv4 地址；query_node 函数用于在字典树中查询指定的域名，并返回其对应的 IPv4 地址。

4 软件流程图



首先启动 DNS 中继器选择运行模式，并通过命令行输入获取程序命令参数，同时初始化。初始化包括初始化 socket，初始化 ID 映射表，初始化 cache，初始化本地 Host 文件，即 dnsrelay.txt。初始化完成后根据用户输入的程序命令参数判断运行模式为阻塞模式还是非阻塞模式。

若为非阻塞模式，则设置 socket 为非阻塞模式。在非阻塞模式下，循环接收来自客户端和服务器的数据。

若为阻塞模式，则开始监听事件，循环等待直到有事件发生。事件发生后判断事件来源，根据来源不同程序选择处理客户端报文或处理远端服务器报文。接着继续监听事件，循环等待直到有事件发生。

5 测试用例以及运行结果

5.1 启动程序

启动程序，以非阻塞模式为例，输入./DNS -d -l -m 0 -i。

其中，-d 表示开启调试模式，-l 表示开启日志记录，-m 0 表示设置程序的运行模式为非阻塞，-i 表示打印系统基本信息。

```
PS D:\Users\SevenGrass\Documents\WILLIAMZHANG\ComputerNetworkCourseDesign\version2\DNS\x64\Debug> ./DNS -d -l -m 0 -i
+-------------------------------------------------------------------------------+
|                            Welcome to use DNS!                                |
|                                                                               |
| Example: nslookup www.bupt.edu.cn 127.0.0.1                                   |
|                                                                               |
| Arguments:                                                                    |
|   -i                        打印系统基本信息                                  |
|   -l:                       日志记录                                          |
|   -d                        开启调试模式                                      |
|   -s [server_address]       设置远程DNS服务器地址                            |
|   -m [mode]                 设置程序的运行模式:0/1  非阻塞/阻塞               |
+-------------------------------------------------------------------------------+
Hosts path: ./dnsrelay.txt
Remote DNS server address: 10.3.9.4
Mode: nonblock
DNS server: 10.3.9.4
Listening on port 53
910 domain name address info has been loaded.
```

5.2 使用 nslookup 测试基本功能

5.2.1 查询功能

向本机服务器询问域名 www.google.com 的 IP 地址。

```
C:\Users\SevenGrass>nslookup www.google.com 127.0.0.1
服务器:  UnKnown
Address:  127.0.0.1

名称:    www.google.com
Addresses:  64.233.168.106
          64.233.168.106
```

5.2.2 不良网站拦截功能

向本机服务器询问域名 008.cn 的 IP 地址。

```
C:\Users\SevenGrass>nslookup 008.cn 127.0.0.1
服务器:  UnKnown
Address:  127.0.0.1

*** UnKnown 找不到 008.cn: Non-existent domain
```

5.2.3 中继功能

向本机服务器询问域名 www.baidu.com 的 IP 地址。

```
C:\Users\SevenGrass>nslookup www.baidu.com 127.0.0.1
服务器:  UnKnown
Address:  127.0.0.1

非权威应答:
名称:    www.a.shifen.com
Addresses:  110.242.68.4
          110.242.68.3
Aliases:  www.baidu.com
```

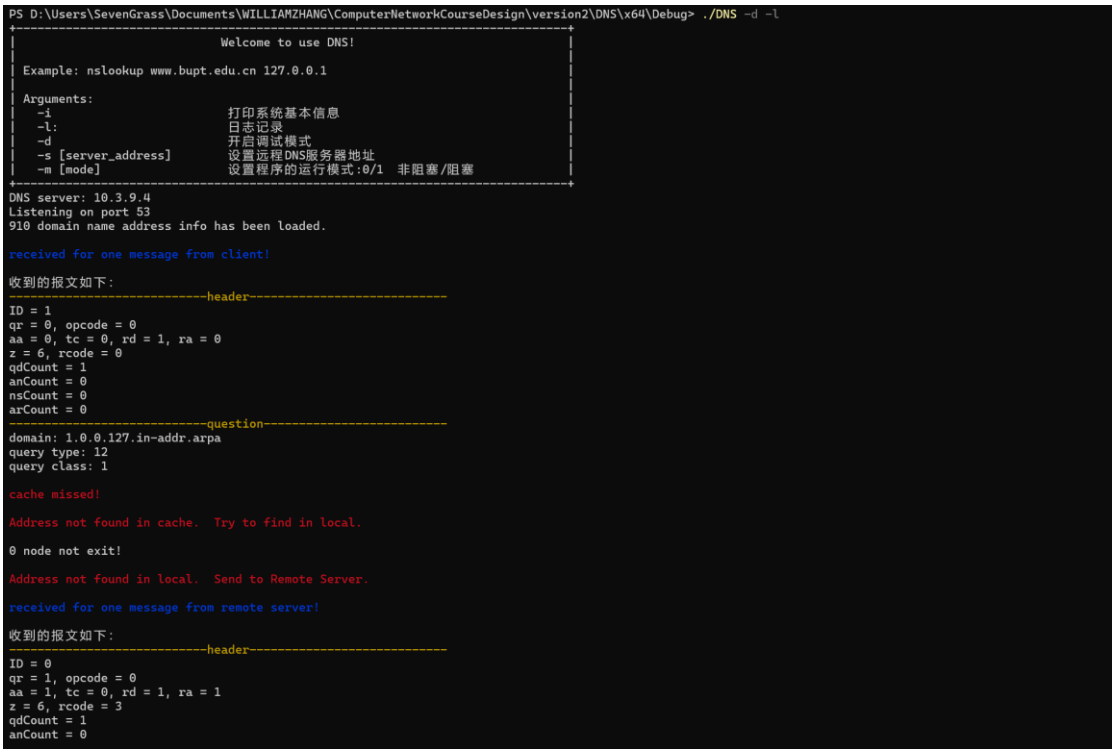
5.3 cache 测试

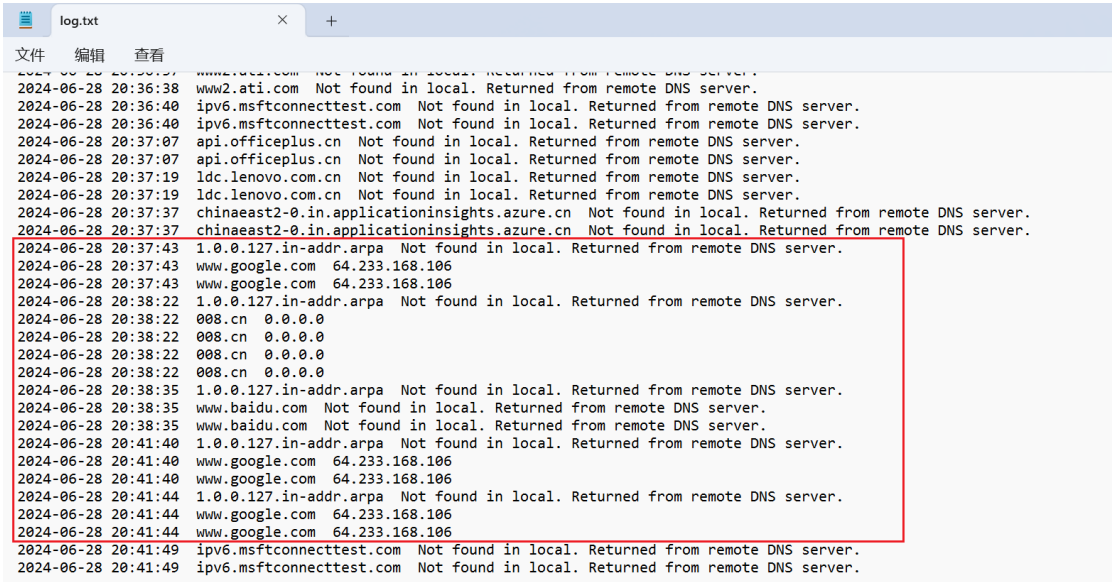
再次向本机服务器询问域名 www.google.com 的 IP 地址。第一次在 cache 中未命中，则在本地的“域名-IP 地址”对照表中查询。命中后加入 cache 缓存中。第二次在 cache 中命中。

```
received for one message from client!

收到的报文如下:
----------------------------header----------------------------
ID = 2
qr = 0, opcode = 0
aa = 0, tc = 0, rd = 1, ra = 0
z = 6, rcode = 0
qdCount = 1
anCount = 0
nsCount = 0
arCount = 0
----------------------------question--------------------------
domain: www.google.com
query type: 1
query class: 1

cache missed!

Address not found in cache.  Try to find in local.

Local hit!  Cache updated!

received for one message from client!

收到的报文如下:
----------------------------header----------------------------
ID = 3
qr = 0, opcode = 0
aa = 0, tc = 0, rd = 1, ra = 0
z = 6, rcode = 0
qdCount = 1
anCount = 0
nsCount = 0
arCount = 0
----------------------------question--------------------------
domain: www.google.com
query type: 28
query class: 1

cache hit!
```

5.4 debug 信息、日志功能测试

输入./DNS -d -l。





5.5 稳定性测试

实际使用过程中，运行 DNS 中继服务器 3 个小时以上，期间正常浏览网页、观看视频、打游戏，服务器没有出现异常，上网体验也与平时无差。

6 调试中遇到的问题及解决方案

6.1 缓存满状态判断风险

使用等于符号(==)来判断缓存是否已满存在潜在风险，特别是在缓存大小(cache_size)快速变化且可能超过最大缓存限制（MAX_CACHE）的情况下。

例如，当缓存大小此时正好等于最大限制，但接下来由于变化迅速缓存大小超过了最大限制，这可能导致缓存管理逻辑出现问题，甚至陷入死锁状态，无法继续正确运行。

为了解决这个问题，程序改用大于等于符号（>=）来判断缓存是否已满。这种方式更为灵活，能够及时响应缓存大小的变化，确保在任何时候都能正确地管理缓存空间，避免因快速变化而导致的不必要的逻辑错误或死锁现象。

通过使用大于等于符号来判断缓存是否已满，可以有效提升系统的稳定性和健壮性，特别是在面对动态变化的缓存需求时更为可靠。

```
void insert_cache(uint8_t ipv4[4], char* domain)
{
      lru_cache* new_cache = (lru_cache*)malloc(sizeof(struct nodes));        // 分配新节点内存

      if (cache_size > MAX_CACHE)  //说明 cache 满了,要进行替换
          delete_node();

      memcpy(new_cache->IP, ipv4, sizeof(uint8_t) * 4);              // 复制 IP 地址
      memcpy(new_cache->domain, domain, strlen(domain) + 1);         // 复制域名

      cache_size++;

      new_cache->next = head->next;              // 将新节点插入到链表头部
      head->next = new_cache;
}
```

6.2 网址大小写不一致而引发的重复存储或匹配问题

网址大小写不一致问题指的是在处理网址时，由于大小写字母被视为不同的字符，导致相同的网址重复存储或无法匹配。

该问题的解决方案是统一将所有网址转换为小写。这样做的好处是确保不同大小写形式的网址被视为相同的内容，从而避免因大小写不一致而引发的重复存储或匹配问题。

具体实现方法为在处理每个网址时，将其全部转换为小写形式。这可以通过现代编程语言中提供的字符串转换函数或方法来轻松完成，这种方法简单而有效，能够确保网址处理过程中的一致性和准确性，特别是在需要进行查找、比较或存储网址时尤为重要。

// 字符转换为数值：将域名中的字符转换为对应的数值

```
int getnum(uint8_t val)
{
    // 数字直接转换为对应的数值
    if (val >= '0' && val <= '9')
        return val - '0';

    // 统一转为小写字母处理
    if (val >= 'A' && val <= 'Z')
        val += 'a' - 'A';

    // 字母转换为对应的数值（a=10, b=11, ..., z=35）
    if(val>='a'&&val<='z')
        return val - 'a' + 10;

    // 特殊字符
    switch (val) {
    case '.':
        return 36;
    case '-':
        return 37;
    default:
        return -1;  // 如果没有匹配到任何已知字符，返回错误代码
    }
}
```

6.3 多次调用 time 函数导致性能下降和不必要的系统开销

在重新分配 ID 过程中，多次调用 time 函数会导致性能下降和潜在的系统资源浪费问题。

为了避免系统多次调用 time 函数，我们设计了一个函数，只在必要时才调用 time 函数，并使用其结果来更新和管理 ID 的过期状态。以下是改进后的函数：

```
uint16_t reset_id(uint16_t user_id, struct sockaddr_in client_address) {
    uint16_t i;
    time_t current_time = time(NULL); // 只调用一次 time() 函数，避免多次系统调用
    for (i = 0; i < MAX_ID_SIZE; i++) {
        if (ID_list[i].expire_time < current_time) { // 检查 ID 是否已过期
            ID_list[i].user_id = user_id;
            ID_list[i].client_address = client_address;
            ID_list[i].expire_time = ID_EXPIRE_TIME + current_time;
            break; // 在找到第一个过期 ID 并分配新值后终止循环
        }
    }
    if (i == MAX_ID_SIZE) {
        return UINT16_MAX; // 如果没有可用的 ID，返回最大值表示失败
    }
    return i; // 返回分配 ID 的索引位置
}
```

改进说明：

①**单次调用 time 函数**：函数开始时只调用一次 time 函数，将当前时间保存在 current_time 变量中，避免了在循环中重复调用。

②**优化遍历和更新逻辑**：在循环中，只有当发现一个过期的 ID 后才会更新该 ID 的相关信息（用户 ID、客户端地址和过期时间），并立即退出循环。这样可以减少不必要的遍历次数和更新操作。

③**返回分配的索引位置**：如果成功分配了 ID，则返回其在 ID_list 数组中的索引位置；如果没有可用的位置，则返回 UINT16_MAX 表示分配失败。

这种优化方案有效地减少了系统调用和提高了函数的执行效率，同时确保了 ID 分配过程的正确性和稳定性。

7 总结和心得体会

这是一个很有挑战性的课程设计任务。代码的编写调试花费了我们超过 30 个小时，而报告编写因为涉及各模块的细节解析，又花费了几个小时完成。

设计并实现一个 DNS 中继服务器不仅要求我们深入理解 DNS 的工作原理和细节，还要求我们精通 socket 编程技术。通过这次实践，我们不仅掌握了构建高性能 DNS 服务器的技能，更开启了对 DNS 和网络编程深层次理解的大门。

在课程设计初期，我们投入了大量时间研究 DNS 协议和网络编程的基础知识，这为我们后续的代码实现打下了坚实的基础。出于对实际开发情景的考察，以及我们成员对 Linux 的通信并不熟悉，我们采用 Windows 操作系统作为开发环境。

在实际开发中，我们充分发挥团队的作用，明确分工，互相激励，完成了课程设计。在调试和测试阶段，通过 Wireshark 等工具的使用，不仅加深了我们对网络数据包格式的理解，更提升了我们对网络分析工具的熟练运用。

特别是在分析转发、接收、发送的 DNS 包时，我们最初因为并不了解通信的全过程，导致抓到了包，但是不知道表示什么，不知道抓到的包是否正确。为了解决这个问题，我们尝试在老师提供的样例 DNS 的运行下进行抓包，然后比较我们的包和样例的包，再通过查找资料，一点一点明确这个通信的过程，也是在这个过程中，我们完善了我们的 DNS 处理报文的功能。

在探索 DNS 安全的过程中，我们意识到了其在网络中的重要性。在查阅了 RFC 文档后，我们了解到 DNS 安全不仅指 DNS 中的数据（即资源记录，RR）安全，还包含在同步或更新 DNS 服务器内容时的传输安全。鉴于 DNS 在 Internet 运行中的重要作用，现代的网络运营商针对其部署了充分的安全机制，称为域名系统安全扩展（DNSSEC）。这些安全扩展不仅提供了端到端的 DNS 安全（即中间解析器不需要是可信的），还减少了中间服务器的计算负担。因此在日常使用 DNS 时并不需要刻意担心 DNS 的安全问题。

综上所述，计算机网络课程设计带给我们的收获不仅仅是一个功能完备的 DNS 中继服务器，而是一段宝贵的学习和成长经历。这段经历让我们勇敢地探索未知，深化了我们对团队协作重要性的认识，增进了彼此之间的理解和包容。我们对网络的运作机制和安全性有了更深层次的思考，同时也在工业级代码设计的实践中不断尝试和优化。