计算机网络课程设计:“DNS 中继服务器”的实现 实验报告

毛子恒 李臻 张梓靖
2019211397 2019211458 2019211379
北京邮电大学 计算机学院

日期：2021 年 6 月 13 日

1 概览

1.1 任务描述

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：

- 检索到 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）
- 检索到普通 IP 地址，则向客户端返回该地址（服务器功能）
- 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）

1.2 开发环境

- macOS Big Sur 11.3
- Apple clang version 12.0.5
- cmake version 3.19.1
- Clion 2021.1.1
- Visual Studio Code 1.56.2

1.3 成员分工

姓名	分工
毛子恒	开发 文档
李臻	测试 文档
张梓靖	开发 文档

2 功能需求

基本需求 细化1.1节指定的三个功能，我们提出如下几个基本需求：

1. 解析和构建 DNS 报文。
2. 监听本地 53 端口，获取 DNS 请求；将查询到的 DNS 回复发送回本地。

3. 加载本地的“IP 地址-域名”对照表，维护一个数据结构，用于增添、查询 DNS 记录。
4. 将 DNS 请求发送到远程服务器的 53 端口，并且监听来自远程服务器的回复。

额外需求 此外，基于性能和实用性的考虑，我们实现了数个额外需求：

1. 在对照表中增加对 IPv6 的支持。
2. 输出调试信息和 DNS 报文内容。
3. 在转发 DNS 请求时重新分配序号，以区分不同的查询。
4. 避免阻塞式 I/O，采用异步方式进行网络通信。
5. 采用事件驱动的方式实现高并发。
6. 对于数据结构中查询不到的 DNS 请求，存储来自服务器的 DNS 回复以供之后的重复查询，并且在到期后删除记录。
7. 实现高速缓存，对于经常访问的记录，加快查询效率。
8. 命令行参数解析。

3 模块介绍

3.1 模块划分

各模块及其关系如图 1。

DNS 中继服务器的基础框架包括服务端、客户端及一个查询池，服务端通过 socket 与本地进程通信，客户端通过 socket 与远程 DNS 服务器通信；客户端和服务端收到报文时，将其转换成结构体的形式，发送报文时，将结构体转化成字节流；中继服务器的各个模块之间通信均采用结构体的形式，相比字节流而言比较直观。中继服务器的基础框架如图 2。

此外，本地缓存分为两层，分别为一个高速缓存 LRU 以及一个红黑树，LRU 的内容是红黑树的子集，每次查询时，首先在 LRU 中查找结果，再在红黑树中查找结果，如果未命中，则向远程服务器发送查询报文。当收到来自远程 DNS 服务器的回复报文时，会复制两个副本，分别插入到 LRU 和红黑树中，这三者的关系如图 3。

3.2 DNS 报文解析和构建

3.2.1 DNS 报文格式分析

一个 DNS 报文由如下五个部分构成：

```
+---------------------+
|        Header       |
+---------------------+
|       Question      |
+---------------------+
|        Answer       |
+---------------------+
|      Authority      |
+---------------------+
```

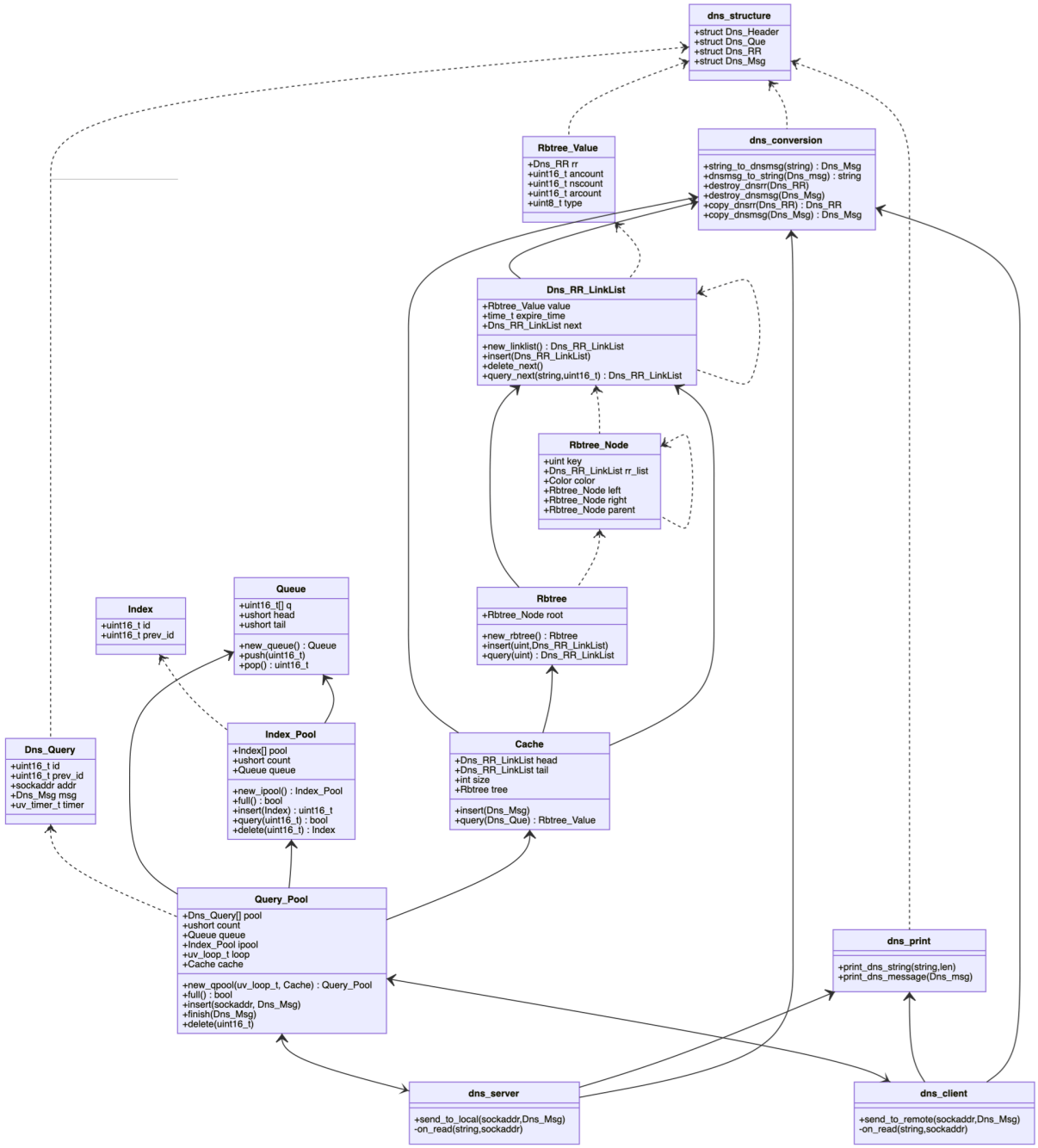


图 1: 模块关系图

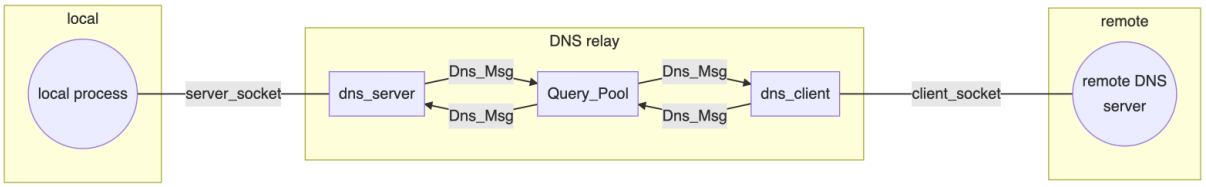


图 2: DNS 中继服务器基础框架

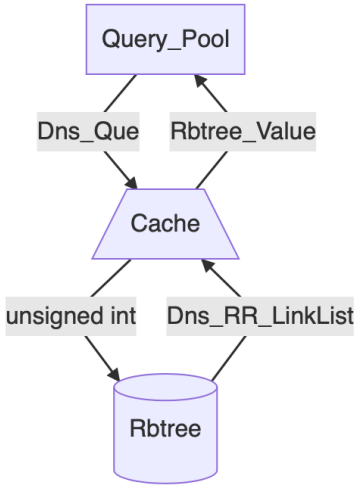


图 3: 查询池、高速缓存和红黑树的关系

```
|      Additional     |
+---------------------+
```

Header 部分格式　Header 部分为报文头，Question 部分的内容为向域名服务器的查询；之后的三个部分有相同的格式，即 Resource Record(RR)，并且都可能为空；Answer 部分是对于查询的回复，Authority 部分的内容指向权威域名服务器，Additional 部分包含一些相关额外信息。

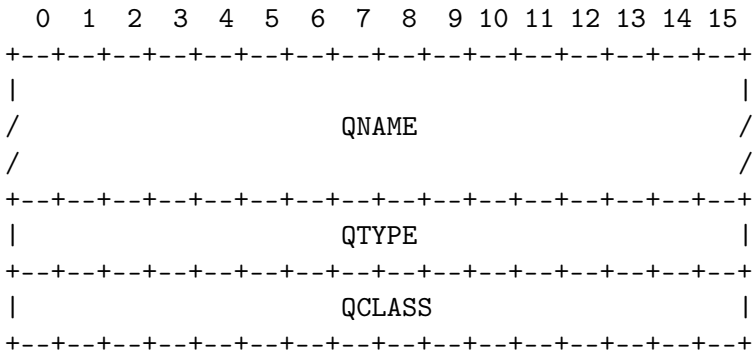
Header 部分的结构如下：

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                      ID                       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|QR|   Opcode  |AA|TC|RD|RA|   Z    |   RCODE   |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    QDCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    ANCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    NSCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    ARCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

- ID，用于由产生 DNS 查询的程序分配，用于标识一个请求；一对 DNS 查询和回复的 ID 相同。
- QR，查询报文此位为 0，回复报文此位为 1。
- OPCODE，查询的类型：
  - 0，标准查询；
  - 1，反向查询；
  - 2，服务器状态请求。

- AA，在回复报文中有效，如果为 1，表示回复 Question 部分查询的域名服务器是权威服务器。
- TC，如果为 1，说明这条消息由于信道的限制而被截断。
- RD，在查询报文中设置，如果为 1，表示期望域名服务器递归查询这个请求。
- RA，在回复报文中设置，如果为 1，表示递归查询在域名服务器中有效。
- Z，预留字段，全 0.
- RCODE，回复状态编号：
  - 0，没有错误；
  - 1，查询格式错误；
  - 2，由于服务器错误而无法处理查询；
  - 3，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；
  - 4，查询的类型不受支持；
  - 5，服务器拒绝处理请求。
- QDCOUNT，Question 部分中查询记录的个数（通常是 1）。
- ANCOUNT，Answer 部分中 RR 的个数。
- NSCOUNT，Authority 部分中 RR 的个数。
- ARCOUNT，Additional 部分中 RR 的个数。

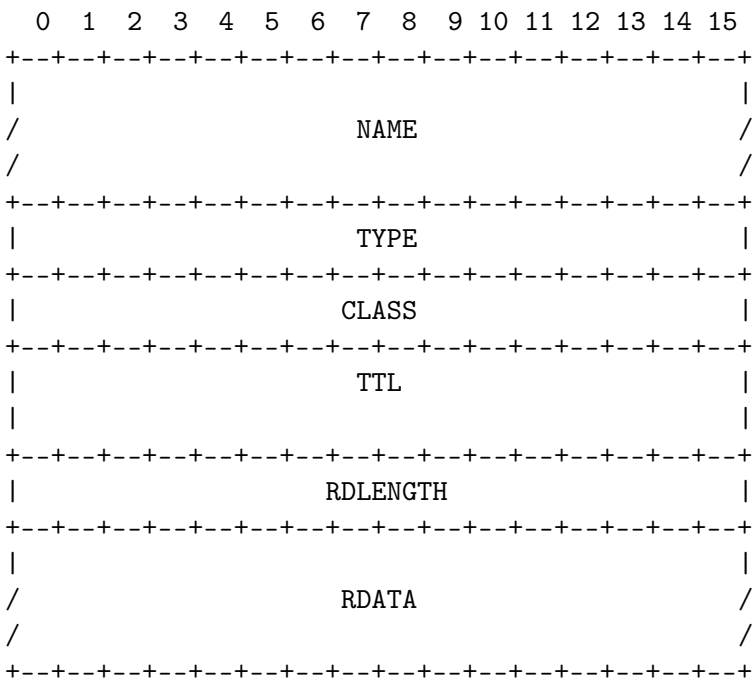
Question 部分格式 Question 部分的结构如下：



- QNAME，查询域名，格式在3.2.1节说明。
- QTYPE，查询类型，受支持的类型如下：
  - 1，A，主机地址；
  - 2，NS，权威域名服务器；
  - 5，CNAME，域名引用；
  - 6，SOA，授权机构起始；
  - 12，PTR，域名指针，用于反向域名查找；
  - 13，HINFO，主机信息；
  - 14，MINFO，邮箱或邮件列表信息；
  - 15，MX，邮件交换；
  - 16，TXT，字符串。

  - 28，AAAA，IPv6 地址。
- QCLASS，查询类别，仅支持 1，IN，因特网。

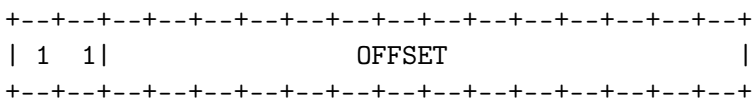
Resource Record 格式　Resource Record 的格式如下：



- NAME，此 RR 从属的域名，格式在3.2.1节说明。
- TYPE，RR 类型，支持的类型与 QTYPE 部分相同。
- CLASS，RR 类别，仅支持 1，IN，因特网。
- TTL，期望此 RR 被缓存的时间。
- RDLENGTH，RDATA 部分的长度。
- RDATA，资源内容，根据 TYPE 的不同，其内容也不同。

域名格式　Question 部分和 RR 中的域名都是由一串字符 `<domain-name>` 表示，一般由 0 表示终止，其长度任意，并且不包含填充。`<domain-name>` 由若干个 `<character-string>` 构成，每个 `<character-string>` 由一个八位数字开头，之后跟着长度等于这个数字的字符串。`<character-string>` 最多包含 256 个字符。

为了压缩 DNS 报文的长度，增大传输效率，引入了域名的压缩，此时 `<domain-name>` 可能由 0 或者一个指针终止，一个指针由两个字节构成，其格式如下：



这个指针指向从 DNS 报文头起第 `OFFSET` 个字节，原本域名的剩余部分应该从这个字节开始继续读取。

RDATA 格式　对于 DNS 中继服务器来说，RDATA 部分的内容并不重要，只是涉及到域名压缩，以及调试输出的需求，所以需要对一部分类型的 RDATA 作处理。需要处理的 RDATA 如下。

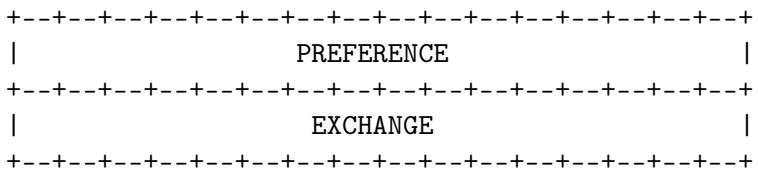
A 类型 RDATA 格式　如下：

- ADDRESS，32 位 IPv4 地址。

NS/CNAME 类型 RDATA 格式　如下：

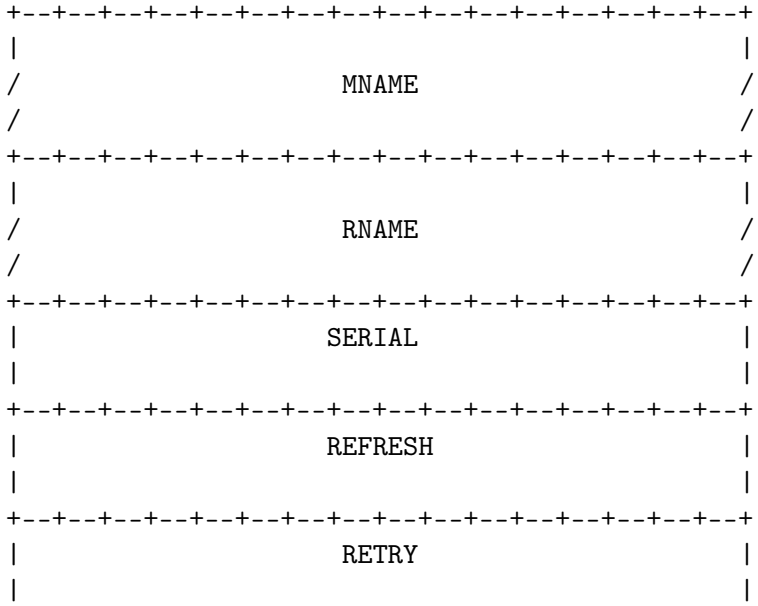
- NAME，一个域名。

MX 类型 RDATA 格式　如下：



- PREFERENCE，16 位数字。
- EXCHANGE，一个域名。

SOA 类型 RDATA 格式　如下：



```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                     EXPIRE                    |
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                     MINIMUM                   |
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

- MNAME，一个域名。
- RNAME，一个域名。
- SERIAL，一个 32 位数字。
- REFRESH，一个 32 位数字。
- RETRY，一个 32 位数字。
- EXPIRE，一个 32 位数字。
- MINIMUM，一个 32 位数字。

AAAA 类型 RDATA 格式 如下：

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                               |
|                                               |
|                                               |
|                     ADDRESS                   |
|                                               |
|                                               |
|                                               |
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

- ADDRESS，128 位 IPv6 地址。

3.2.2 DNS 报文结构体

DNS 报文结构体定义在 `dns_structure.h` 中，由四个结构体构成，其结构如图 4所示。

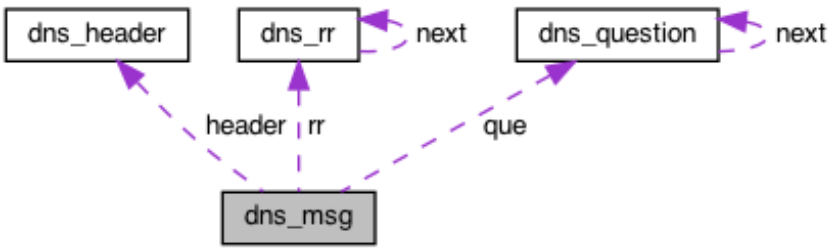


图 4: DNS 报文结构体的结构

Header 部分长度固定，其中部分不足一个字节的字段采用位域 (bit field) 的方式定义，节省空间；Question 部分的查询记录个数不定，采用单向链表实现，便于顺序访问；之后的 Answer、Authority 和 Additional 部分的格式相同，采用一个单向链表顺序连接起来。

四个结构体的定义如下所示：

```
typedef struct dns_header
{
    uint16_t id;
    uint8_t qr: 1;
    uint8_t opcode: 4;
    uint8_t aa: 1;
    uint8_t tc: 1;
    uint8_t rd: 1;
    uint8_t ra: 1;
    uint8_t z: 3;
    uint8_t rcode: 4;
    uint16_t qdcount;
    uint16_t ancount;
    uint16_t nscount;
    uint16_t arcount;
} Dns_Header;

typedef struct dns_question
{
    uint8_t * qname;
    uint16_t qtype;
    uint16_t qclass;
    struct dns_question * next;
} Dns_Que;

typedef struct dns_rr
{
    uint8_t * name;
    uint16_t type;
    uint16_t class;
    uint32_t ttl;
    uint16_t rdlength;
    uint8_t * rdata;
    struct dns_rr * next;
} Dns_RR;

typedef struct dns_msg
{
    Dns_Header * header;
    Dns_Que * que;
    Dns_RR * rr;
} Dns_Msg;
```

dns_structure.h 的文档见dns_structure.h 文件参考。

3.2.3 DNS 报文字节流和结构体的转换

从 UDP socket 中获取到的 DNS 报文是字节流的形式，我们需要将它转换成结构体，便于分析与输出，之后需要将结构体转换回字节流进行发送。

此外，我们还需要关注报文结构体和 RR 结构体的复制和销毁操作。

上述操作定义在 `dns_conversion.h` 中，如下：

```
1  void string_to_dnsmsg(Dns_Msg * pmsg, const char * pstring);
2
3  unsigned dnsmsg_to_string(const Dns_Msg * pmsg, char * pstring);
4
5  void destroy_dnsrr(Dns_RR * prr);
6
7  void destroy_dnsmsg(Dns_Msg * pmsg);
8
9  Dns_RR * copy_dnsrr(const Dns_RR * src);
10
11 Dns_Msg * copy_dnsmsg(const Dns_Msg * src);
```

`dns_conversion.h` 的文档见dns_conversion.h 文件参考。

将字节流转换成结构体 `string_to_dnsmsg` 函数实现将字节流转换成结构体，它的函数调用图如图 5。

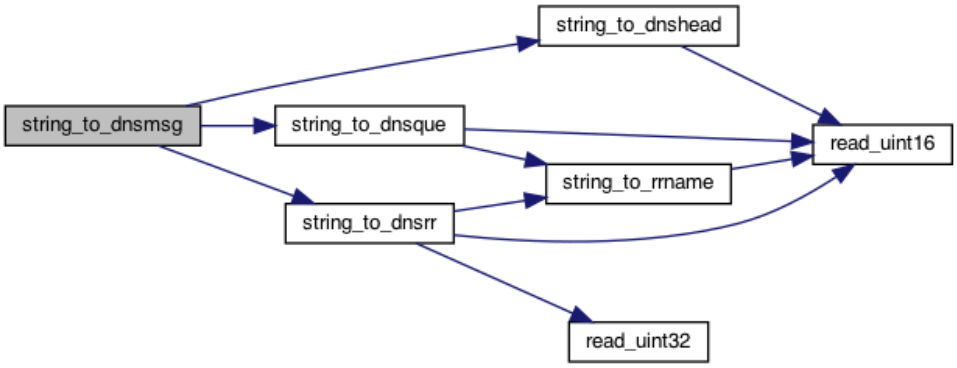


图 5: `string_to_dnsmsg` 函数调用图

从图中可见，此函数分别调用 `string_to_dnshead`、`string_to_dnsque` 和 `string_to_dnsrr` 将报文的三个部分转换为结构体，对于 Question 部分和 RR 部分，需要维护一个链表头指针和尾指针，每次转换后的新记录插入链表尾即可。

此外，考虑到域名格式的特殊性，`string_to_rrname` 函数负责通过递归的形式将域名解析成可输出的常见的格式，`read_uint16` 和 `read_uint32` 函数用于将大端法数字转换成小端法数字。

将结构体转换成字节流 `dnsmsg_to_string` 函数实现将结构体转换成字节流，它的函数调用图如图 6。

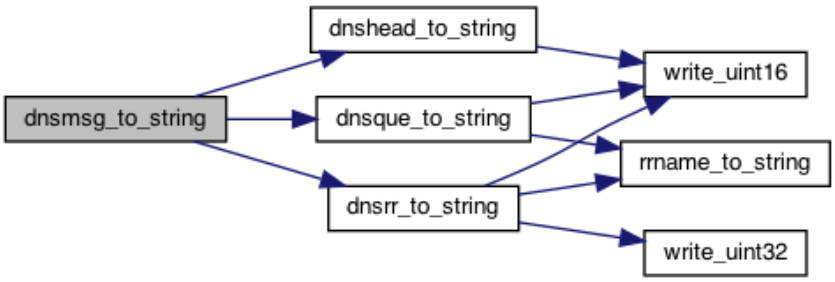


图 6: `dnsmsg_to_string` 函数调用图

`dnsmsg_to_string` 函数基本上是 `string_to_dnsmsg` 的逆过程，分别调用类似的六个函数实现相应的功能。

结构体的销毁 `destroy_dnsmsg` 和 `destroy_dnsrr` 函数分别实现整个 DNS 报文的销毁和 RR 链表的销毁。

结构体的复制 `copy_dnsmsg` 和 `copy_dnsrr` 函数分别实现整个 DNS 报文的复制和 RR 链表的复制。

上面提到的函数接口的的文档见dns_conversion.c 文件参考。

3.2.4 DNS 报文的输出

通过输出 DNS 字节流和结构体，可以观察 DNS 报文的转换过程是否正确，并且跟踪程序运行的过程，便于 debug 的工作。

上述操作定义在 `dns_print.h` 中，如下：

```
1 void print_dns_string(const char * pstring, unsigned int len);
2
3 void print_dns_message(const Dns_Msg * pmsg);
```

`dns_print.h` 的文档见dns_print.h 文件参考。

`print_dns_string` 函数用于输出 DNS 字节流，输出的效果如下：

```
0000 bd da 81 80 00 01 00 01 00 00 00 00 03 77 77 77
0010 04 62 75 70 74 03 65 64 75 02 63 6e 00 00 01 00
0020 01 03 77 77 77 04 62 75 70 74 03 65 64 75 02 63
0030 6e 00 00 01 00 01 ff ff ff ff 00 04 0a 03 09 a1
```

`print_dns_message` 函数用于输出 DNS 结构体，其函数调用图如**图 7**。

该函数分别调用 `print_dns_header`、`print_dns_question` 和 `print_dns_rr` 函数输出结构体的三个部分。由于 RR 的 RDATA 字段格式较多，所以另外实现了几个函数分别用于输出 A、AAAA、MX(NS)、CNAME、SOA 类型的 RDATA 字段。

输出的效果如下：

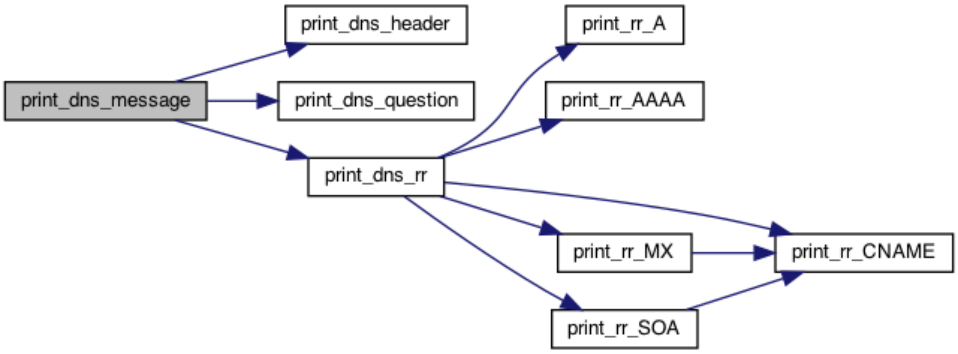


图 7: `print_dns_message` 函数调用图

```
=======Header==========
ID = 0xbdda
QR = 1
OPCODE = 0
AA = 0
TC = 0
RD = 1
RA = 1
RCODE = 0
QDCOUNT = 1
ANCOUNT = 1
NSCOUNT = 0
ARCOUNT = 0

=======Question========
QNAME = www.bupt.edu.cn.
QTYPE = 1
QCLASS = 1

=======Answer==========
NAME = www.bupt.edu.cn.
TYPE = 1
CLASS = 1
TTL = 4294967295
RDLENGTH = 4
RDATA = 10.3.9.161

=======Authority=======
=======Additional======
```

上面提到的函数接口的的文档见dns_print.c 文件参考。

3.3 DNS 记录的存储

理论上，通过一个域名和类型可以查询到一个 RR 的列表作为回复。对于 DNS 记录的存储就是基于这个对应关系。

3.3.1 红黑树和 RR 链表

DNS 记录存储在一个红黑树中，红黑树节点的键是域名字符串经过哈希得到的整数，由于哈希可能存在冲突，并且一个域名可能对应多个不同类的查询，可采用拉链法解决冲突。因此，每个红黑树节点对应多个值，这些值被串联成一个有空头结点的单向链表。每个值中存储域名和查询类型、一个 RR 的列表和 RR 列表中各个部分的个数。

上述数据结构的结构体的关系如图 **8**所示。

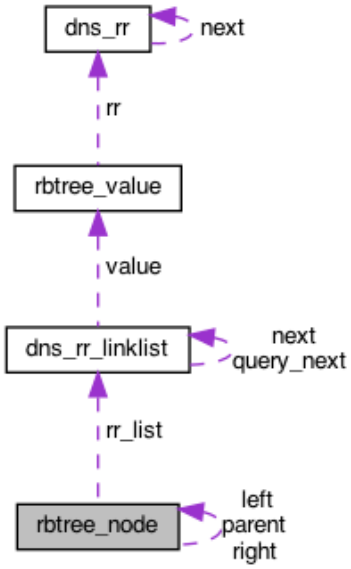


图 8: 红黑树和 RR 链表结构体的关系

红黑树中会存储从对照表中读取到的”IP 地址-域名“对，也会存储运行时收到的回复。当存储对照表中的记录时，默认这些记录是永久的，而存储运行时的回复时，需要根据 RR 列表中最小的 TTL 确认这条记录的过期时间，过期时间记录在链表的节点中。

此外，红黑树和 RR 链表（以及之后的大部分数据结构）都将数据结构的操作作为函数指针封装在结构体中，达到类似成员函数的效果。

红黑树和 RR 链表的数据结构和操作定义在 `rbtree.h` 中，如下：

```
1  typedef enum
2  {
3      BLACK, RED
4  } Color;
5
6  typedef struct rbtree_value
7  {
8      Dns_RR * rr;
9      uint16_t ancount;
10     uint16_t nscount;
11     uint16_t arcount;
12     uint8_t type;
13 } Rbtree_Value;
14
```

```
15  typedef struct dns_rr_linklist
16  {
17      Rbtree_Value * value;
18      time_t expire_time;
19      struct dns_rr_linklist * next;
20
21      void (* insert)(struct dns_rr_linklist * list, struct dns_rr_linklist *
        ↪  new_list_node);
22
23      void (* delete_next)(struct dns_rr_linklist * list);
24
25      struct dns_rr_linklist * (* query_next)(struct dns_rr_linklist * list,
        ↪  const uint8_t * qname, const uint16_t qtype);
26  } Dns_RR_LinkList;
27
28  typedef struct rbtree_node
29  {
30      unsigned int key;
31      Dns_RR_LinkList * rr_list;
32      Color color;
33      struct rbtree_node * left;
34      struct rbtree_node * right;
35      struct rbtree_node * parent;
36  } Rbtree_Node;
37
38  typedef struct rbtree
39  {
40      Rbtree_Node * root;
41
42      void (* insert)(struct rbtree * tree, unsigned int key, Dns_RR_LinkList *
        ↪  list);
43
44      Dns_RR_LinkList * (* query)(struct rbtree * tree, unsigned int data);
45  } Rbtree;
46
47  Dns_RR_LinkList * new_linklist();
48
49  Rbtree * new_rbtree();
```

rbtree.h 的文档见rbtree.h 文件参考。

RR 链表 Dns_RR_LinkList 结构体有三个方法：

- insert 方法用于在链表中的某个节点后插入新节点。
- delete_next 方法用于删除链表中某个节点的下一个结点。
- query_next 方法用于遍历一个链表，根据域名和查询类型从中筛选出一个节点。

此外，new_linklist 函数用于创建一个新的 RR 链表节点。

红黑树的插入操作 Rbtree 结构体的 insert 方法用于向红黑树中插入键-值对。

具体来说，函数传入一个键和一个 RR 链表节点。首先在红黑树中按照键查找对应的节点，如果查找到节点，则将 RR 链表节点插入这个节点的 RR 链表尾；如果查找不到节点，则创建一个新节点以及一个有空头结点的 RR 链表，并将传入的 RR 链表节点插入到这个链表尾，在此之后，需要调用 `insert_case` 函数维护红黑树的平衡。

对于红黑树的维护细节不属于本报告的范畴，故略去。

这个方法对应 `rbtree.c` 中的 `rbtree_insert` 函数，该函数的调用图如**图 9**所示。

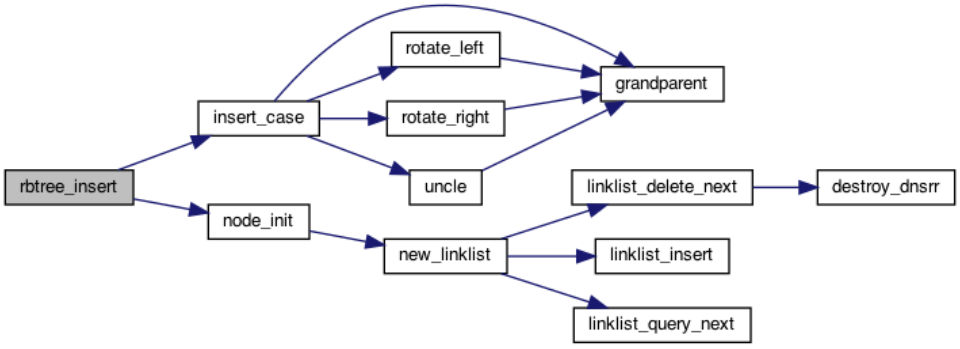


图 9: `rbtree_insert` 函数的调用图

红黑树的查询操作 `Rbtree` 结构体的 `query` 方法用于根据键在红黑树中查询 RR 链表。

函数调用 `rbtree_find` 函数，在红黑树中根据键查找结点。如果查找到节点，将节点的 RR 链表中超时的节点删去。如果此时节点的链表为空（即只有一个头节点），则调用 `rbtree_delete` 函数删除这个节点，并且维护红黑树的平衡；否则返回这个 RR 链表。

这个方法对应 `rbtree.c` 中的 `rbtree_query` 函数，该函数的调用图如**图 10**所示。

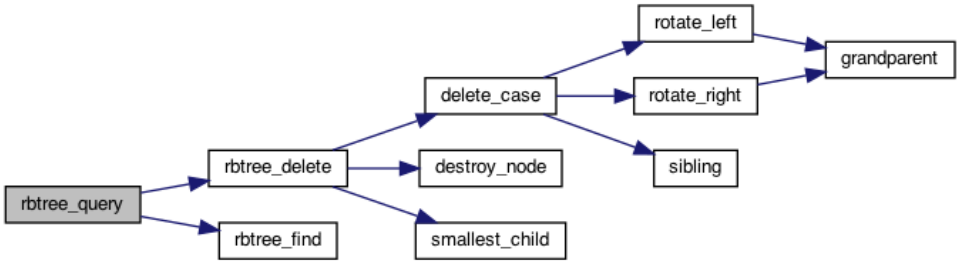


图 10: `rbtree_query` 函数的调用图

此外，`new_rbtree` 函数用于创建一棵新的空红黑树。

上面提到的函数接口的的文档见rbtree.c 文件参考。

3.3.2 LRU 高速缓存

对于经常访问的缓存，将其存储到高速缓存中可以加快查询速度。利用3.3.1节提到的 RR 链表，我们实现了一个 LRU(Least Recently Used，最近最少使用) 缓存。

对于在这个 RR 链表中，最近被访问的记录会被移到链表尾，当链表长度达到上限时，优先移出链表头的节点，即最久未使用的值最先被移出缓存。

缓存的数据结构和操作定义在 `cache.h` 中，如下：

```c
#define CACHE_SIZE 30

typedef struct cache_
{
    Dns_RR_LinkList * head;
    Dns_RR_LinkList * tail;
    int size;
    Rbtree * tree;

    void (* insert)(struct cache_ * cache, const Dns_Msg * msg);

    Rbtree_Value * (* query)(struct cache_ * cache, const Dns_Que * que);
} Cache;

Cache * new_cache(FILE * hosts_file);
```

cache.h 的文档见cache.h 文件参考。

缓存的初始化 new_cache 函数用于创建一个新的缓存。

此函数会调用 new_rbtree 创建一棵新的红黑树，之后从对照表的文件中读入“域名-IP”对，构造 RR 记录和 RR 链表节点并插入红黑树中。

一个对照表的例子如下：

```
baidu.com 220.181.38.148
www.cloudflare.com 104.16.124.96
www.bupt.edu.cn 10.3.9.161
google.com 172.217.160.78
test.com 0.0.0.0
bt.byr.cn 2001:da8:215:4078:250:56ff:fe97:654d
```

缓存的插入操作 Cache 结构体的 insert 方法用于向缓存中插入 DNS 记录。

该函数会从 DNS 报文数据结构中提取出 QNAME、QTYPE、ANCOUNT、NSCOUNT、ARCOUNT，将 RR 部分复制两遍以创建两个 RR 链表节点，将这两个节点分别插入 LRU 缓存中和红黑树中。

函数调用 BKDRHash 函数对域名生成字符串哈希，调用 get_min_ttl 得到一个 RR 列表中最小的 TTL。

这个方法对应 cache.c 中的 cache_insert 函数，该函数的调用图如**图 11**所示。

缓存的查询操作 Cache 结构体的 query 方法用于在缓存中查询 DNS 记录。

该函数会首先在 LRU 中查询 DNS 记录，如果查询到，将该节点移到链表的末尾，之后生成该节点中的值的副本并且返回；

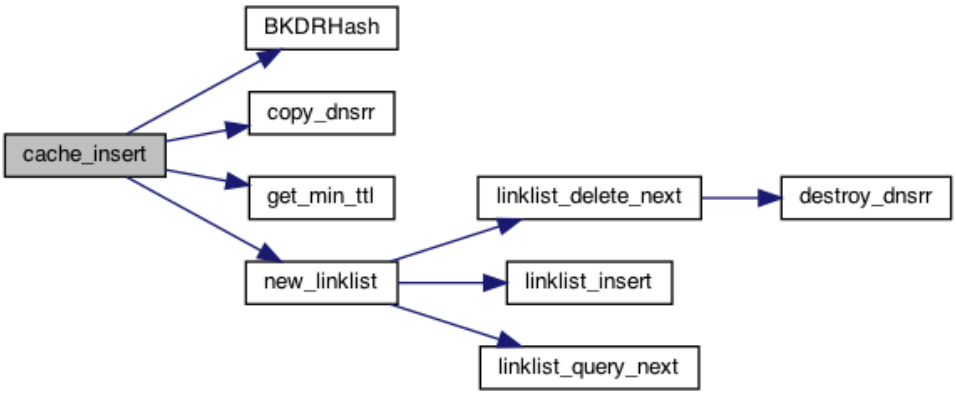


图 11: `cache_insert` 函数的调用图

如果在 LRU 中查找不到，则调用红黑树的 `query` 方法进行查询，在返回的链表中查找满足 QNAME 和 QTYPE 的值，将查找到的值复制两份，一份插入到 LRU 缓存中，一份返回。

如果红黑树中也查找不到，则返回空指针。

这个方法对应 `cache.c` 中的 `cache_query` 函数，该函数的调用图如**图 12**所示。

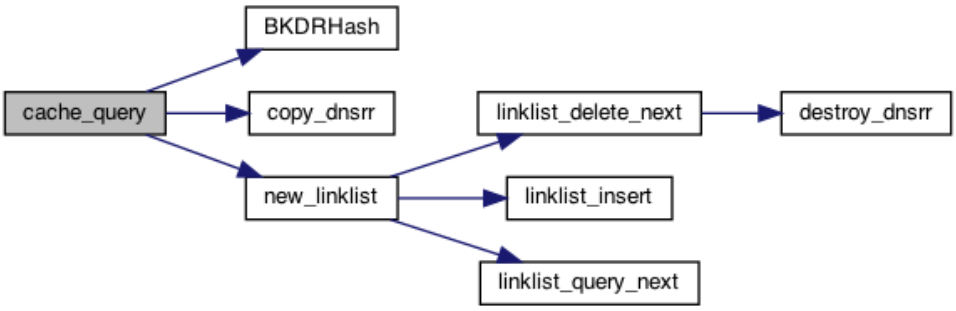


图 12: `cache_query` 函数的调用图

上面提到的函数接口的的文档见cache.c 文件参考。

3.4 序号池和查询池

为了实现并发查询，我们实现了一个查询池，查询池中的内容为 DNS 查询。中继服务器会为每个 DNS 查询分配一个序号，同时保存以下信息：

- 请求方的地址，中继服务器向这个地址发送回复。
- 原本的 DNS 查询报文，中继服务器收到远程回复时和原本的查询报文比对。
- 计时器，DNS 查询超时后自动从查询池中移除并销毁。

为了防止短时间内查询序号重复，导致无法区分当前查询和之前的查询，我们采用滑动窗口的方式分配查询序号。查询池的大小为 256，而查询序号为 0 至 65535。

具体来说，查询池外维护一个队列，队列中保存未使用的查询序号，初始情况下，队列中有 0 至 255 的序号；当需要在查询池中添加查询时，从队列中取出一个序号并创建新的 DNS 查询；当一个从查询池中移除一个查询时，将它的查询编号加 256 后再加入队列。

DNS 报文中的 ID 是由不同进程自主生成的，为了区分不同的报文，中继服务器在向远程服务器发送报文之前，需要重新分配报文 ID，并且建立报文 ID 和 DNS 查询序号的对应关系。

因此，我们另外实现了一个序号池，实现方式和查询池类似，即采用一个队列保存未分配的序号，需要分配新序号时从队列里取出并加入查询池，序号用完后从查询池中移除并加入队列。在分配序号时不需要考虑重复的情况，所以序号池的大小为 65536。

3.4.1 队列

为了节省空间，我们实现了一个循环队列，大小为 65536。队列的下标采用自然溢出的方式。

队列的数据结构和操作定义在 `queue.h` 中，如下：

```
1  #define QUEUE_MAX_SIZE 65536
2
3  typedef struct queue
4  {
5      uint16_t q[QUEUE_MAX_SIZE];
6      unsigned short head;
7      unsigned short tail;
8
9      void (* push)(struct queue * queue, uint16_t num);
10
11     uint16_t (* pop)(struct queue * queue);
12
13     void (* destroy)(struct queue * queue);
14 } Queue;
15
16 Queue * new_queue();
```

`queue.h` 的文档见queue.h 文件参考。

队列的初始化 `new_queue` 函数用于创建一个新的队列。

队列的入队操作 `Queue` 结构体的 `push` 方法用于向队尾插入值。

队列的出队操作 `Queue` 结构体的 `pop` 方法取出队首的值并返回。

队列的销毁 `Queue` 结构体的 `destroy` 方法销毁队列。

上面提到的函数接口的的文档见queue.c 文件参考。

3.4.2 序号池

如前所述，我们定义一个结构体保存报文的序号及其对应查询的序号。一个报文序号在序号池中的位置是固定的（即，如果结构体中的序号是 x，则查询池中下标为 x 的指针指向这个结构体），所以判断一个序号是否在序号池中，只需要判断序号池中对应位置的指针是否为空即可。

序号池的数据结构和操作定义在 `index_pool.h` 中，如下：

```c
#define INDEX_POOL_MAX_SIZE 65535

typedef struct index_
{
    uint16_t id;
    uint16_t prev_id;
} Index;

typedef struct index_pool
{
    Index * pool[INDEX_POOL_MAX_SIZE];
    unsigned short count;
    Queue * queue;

    bool (* full)(struct index_pool * ipool);

    uint16_t (* insert)(struct index_pool * ipool, Index * req);

    bool (* query)(struct index_pool * ipool, uint16_t index);

    Index * (* delete)(struct index_pool * ipool, uint16_t index);

    void (* destroy)(struct index_pool * ipool);
} Index_Pool;

Index_Pool * new_ipool();
```

index_pool.h 的文档见index_pool.h 文件参考。

序号池的初始化 new_ipool 函数用于创建一个新的序号池。

判断序号池满 Index_Pool 结构体的 full 方法用于判断序号池是否已满。

序号池的插入操作 Index_Pool 结构体的 insert 方法从队首取出序号加入序号池。

序号池的移除操作 Index_Pool 结构体的 delete 方法从序号池中删除给定的序号，并且将序号插入队列。

序号池的销毁 Index_Pool 结构体的 destroy 方法销毁序号池。

上面提到的函数接口的的文档见index_pool.c 文件参考。

3.4.3 查询池

如前所述，我们定义一个结构体表示 DNS 查询，以及另一个结构体表示查询池。

查询池的数据结构和操作定义在 query_pool.h 中，如下：

```c
#define QUERY_POOL_MAX_SIZE 256

typedef struct dns_query
{
    uint16_t id;
    uint16_t prev_id;
    struct sockaddr addr;
    Dns_Msg * msg;
    uv_timer_t timer;
} Dns_Query;

typedef struct query_pool
{
    Dns_Query * pool[QUERY_POOL_MAX_SIZE];
    unsigned short count;
    Queue * queue;
    Index_Pool * ipool;
    uv_loop_t * loop;
    Cache * cache;

    bool (* full)(struct query_pool * qpool);

    void (* insert)(struct query_pool * qpool, const struct sockaddr * addr,
    ↪  const Dns_Msg * msg);

    void (* finish)(struct query_pool * qpool, const Dns_Msg * msg);

    void (* delete)(struct query_pool * qpool, uint16_t id);
} Query_Pool;

Query_Pool * new_qpool(uv_loop_t * loop, Cache * cache);
```

query_pool.h 的文档见query_pool.h 文件参考。

查询池的初始化 `new_qpool` 函数用于创建一个新的查询池。

判断查询池满 `Query_Pool` 结构体的 `full` 方法用于判断查询池是否已满。

查询池的插入操作 `Query_Pool` 结构体的 `insert` 方法用于在查询池中创建新 DNS 查询。

当 DNS 服务端收到来自本地的查询请求时，会调用这个函数。此函数会创建新的 DNS 查询结构体，从队列中取出序号并分配给它。之后，查询池会首先在缓存中查找记录，如果查找成功，则销毁这个 DNS 查询、构造回复报文（如果需要 DNS 污染则构造特殊报文）并且发送回本地；否则查询池利用序号池分配新的报文序号，并启动计时器，向远程服务器发送报文。

查询池的删除操作 `Query_Pool` 结构体的 `delete` 方法用于从查询池中移除一个 DNS 查询，销毁这个结构体并且将查询序号放回队列。

当一个 DNS 查询超时或者正常结束时会调用这个函数。

查询池的结束查询操作 `Query_Pool` 结构体的 `finish` 方法用于结束一个 DNS 查询。

当 DNS 客户端收到来自远程的回复时，会调用这个函数。此函数根据报文序号在序号池中找到查询的序号，再在查询池中找到相应的查询，和原本的查询报文进行比对，如果比对成功，则向缓存中插入这个回复报文，并且将报文发回本地，之后调用 `delete` 方法从查询池中删除查询。

上面提到的函数接口的的文档见query_pool.c 文件参考。

拦截 拦截功能是在普通的域名查询功能上的改进，它和正常查询流程的不同之处在于，仅通过域名确定查询结果而不考虑查询类型。

在对照表文件中，需要拦截的域名对应的 IP 以 0.0.0.0 表示，当存入红黑树时，若检测到这个 IP，则将值的 TYPE 设置成 255（这个类型不会在正常 RR 的 TYPE 字段中出现）。当查询时，如果在红黑树或者 LRU 中查找到 TYPE 为 255 的值，则作为合法结果返回。在查询池中构造返回报文时，将头部分的 OPCODE 字段设置成 5，并且将 Answer 部分置为空。

3.5 DNS 服务端和客户端

DNS 服务端与本地的进程通信，DNS 客户端与远程 DNS 服务器通信。

通常，DNS 报文采用 UDP 协议发送。服务端和客户端采用libuv提供的跨平台异步 I/O，实现无阻塞的 UDP 报文发送和接收。采用事件驱动的方式实现并发查询。

3.5.1 DNS 服务端

DNS 服务端的接口定义在 `dns_server.h` 中，如下：

```
1 void init_server(uv_loop_t * loop);
2
3 void send_to_local(const struct sockaddr * addr, const Dns_Msg * msg);
```

`dns_server.h` 的文档见dns_server.h 文件参考。

服务端的初始化 `init_server` 函数用于初始化服务端。

该函数首先将服务端 socket 绑定到事件循环，之后将 0.0.0.0:53 绑定到服务端 socket，作为接收地址。

当服务端 socket 上有消息发来时，首先调用 `alloc_buffer` 函数分配一个固定大小的缓冲区，之后由 libuv 负责异步的消息接收，当消息接收完毕后调用 `on_read` 函数，对收到的消息进行处理。

服务端收到本地查询报文 定义在 `dns_server.c` 中的函数 `on_read` 对收到的 DNS 报文进行处理，其函数原型如下：

```
static void on_read(uv_udp_t * handle, ssize_t nread, const uv_buf_t * buf,
    const struct sockaddr * addr, unsigned flags);
```

其中各个参数的解释如下：

- `handle` 为 UDP 句柄；
- `nread` 表示收到报文的长度；
- `buf` 为接收消息之前分配的缓冲区，其中存储 DNS 报文；
- `addr` 为发送方地址；
- `flags` 为一些标志。

收到消息后，函数执行如下几个步骤：

1. 输出报文信息；
2. 将字节流转换成结构体；
3. 如果查询池未满，则将报文加入查询池；
4. 释放报文结构体和缓冲区。

服务端向本地发送回复报文 `send_to_local` 函数将回复报文发送至本地进程，该函数执行如下几个步骤：

1. 输出报文信息；
2. 将结构体转换成字节流；
3. 分配缓冲区，拷贝字节流；
4. 创建 UDP 句柄；
5. 发送报文。

由 libuv 负责异步的报文发送，报文发送完成后，调用定义在 `dns_server.c` 中的 `on_send` 函数，该函数负责 UDP 句柄和缓冲区的释放。

上面提到的函数接口的的文档见dns_server.c 文件参考。

3.5.2 DNS 客户端

DNS 客户端的接口定义在 `dns_client.h` 中，如下：

```
void init_client(uv_loop_t * loop);

void send_to_remote(const Dns_Msg * msg);
```

`dns_client.h` 的文档见dns_client.h 文件参考。

客户端的初始化 `init_client` 函数用于初始化客户端。

该函数首先将客户端 socket 绑定到事件循环，之后将本地某一端口和远程 DNS 服务器的 53 端口绑定到客户端 socket。

当客户端 socket 上有消息发来时，首先调用 `alloc_buffer` 函数分配一个固定大小的缓冲区，之后由 libuv 负责异步的消息接收，当消息接收完毕后调用 `on_read` 函数，对收到的消息进行处理。

客户端收到远程回复报文 定义在 `dns_client.c` 中的函数 `on_read` 对收到的 DNS 报文进行处理，其函数原型如下：

```
1 static void on_read(uv_udp_t * handle, ssize_t nread, const uv_buf_t * buf,
  ↪  const struct sockaddr * addr, unsigned flags);
```

其参数和执行步骤和服务端的同名函数几乎一致，其中调用查询池的 `finish` 方法结束查询。

客户端向远程发送查询报文 `send_to_remote` 函数将查询报文发送至远程 DNS 服务器，该函数的步骤和服务端的 `send_to_local` 函数基本一致。

由 libuv 负责异步的报文发送，报文发送完成后，调用定义在 `dns_client.c` 中的 `on_send` 函数，该函数负责 UDP 句柄和缓冲区的释放。

上面提到的函数接口的的文档见dns_client.c 文件参考。

3.6 其他工具模块

3.6.1 命令行参数解析

命令行参数解析的接口定义在 `config_jar.h` 中，如下：

```
1 extern char * REMOTE_HOST;
2 extern int LOG_MASK;
3 extern int CLIENT_PORT;
4 extern char * HOSTS_PATH;
5 extern char * LOG_PATH;
6
7 void init_config(int argc, char * const * argv);
```

`config_jar.h` 的文档见config_jar.h 文件参考。

程序提供五个命令行参数：

- `--remote_host`，远程 DNS 服务器地址，默认为 10.3.9.4；
- `--log_mask`，调试等级掩码，调试信息分为四个等级，分别为 DEBUG、INFO、ERROR、FATAL，调试等级掩码是一个 4 位二进制数，从低位到高位分别对应这四个等级，如果对

应位为 1，表示输出这个调试等级的信息，例如，当调试等级为 3 时，输出 DEBUG 和 INFO 等级的信息；

- `--client_post`，客户端端口，可以指定一个 1024 至 65535 之间的端口，默认任意分配一个端口；
- `--hosts_path`，“域名-IP”对照表文件路径，默认为上一级目录下的 `hosts.txt`；
- `--log_path`，日志文件路径，默认为空，表示调试信息输出到 `stderr`。

`init_config` 函数的参数应和 `main` 函数的参数保持一致，该函数通过解析字符数组获得参数。

3.6.2 调试信息输出

如3.6.1节所述，调试等级分为四级，在 `util.h` 中定义了四个宏，分别用于输出四个等级的调试信息，如下：

```
1  extern FILE * log_file;
2
3  #define log_debug(args...) \
4      if (LOG_MASK & 1) \
5      { \
6          if (log_file != stderr) \
7              fprintf(log_file, "[DEBUG] %s:%d ", __FILE__, __LINE__); \
8          else \
9              fprintf(log_file, "\x1b[37m[DEBUG]\x1b[36m %s:%d \x1b[0m",
   ↪  __FILE__, __LINE__); \
10         fprintf(log_file, args); \
11         fprintf(log_file, "\n"); \
12     }
13
14 #define log_info(args...) \
15     if (LOG_MASK & 2) \
16     { \
17         if (log_file != stderr) \
18             fprintf(log_file, "[INFO ] %s:%d ", __FILE__, __LINE__); \
19         else \
20             fprintf(log_file, "\x1b[34m[INFO ]\x1b[36m %s:%d \x1b[0m",
   ↪  __FILE__, __LINE__); \
21         fprintf(log_file, args); \
22         fprintf(log_file, "\n"); \
23     }
24
25 #define log_error(args...) \
26     if (LOG_MASK & 4) \
27     { \
28         if (log_file != stderr) \
29             fprintf(log_file, "[ERROR] %s:%d ", __FILE__, __LINE__); \
30         else \
31             fprintf(log_file, "\x1b[33m[ERROR]\x1b[36m %s:%d \x1b[0m",
   ↪  __FILE__, __LINE__); \
```

```
32          fprintf(log_file, args); \
33          fprintf(log_file, "\n"); \
34      }
35
36  #define log_fatal(args...) \
37      if (LOG_MASK & 8) \
38      { \
39          if (log_file != stderr) \
40              fprintf(log_file, "[FATAL] %s:%d ", __FILE__, __LINE__); \
41          else \
42              fprintf(log_file, "\x1b[31m[FATAL]\x1b[36m %s:%d \x1b[0m",
    ↪  __FILE__, __LINE__); \
43          fprintf(log_file, args); \
44          fprintf(log_file, "\n"); \
45          exit(EXIT_FAILURE); \
46      }
```

`util.h` 的文档见util.h 文件参考。

在 `stderr` 中输出调试信息时，行首的调试等级会加以颜色区分，文件名和行号信息会以蓝色加以区分；在文件中输出调试信息时不会打印颜色信息。输出 FATAL 等级的调试信息后，程序会异常终止。

输出调试信息的效果如**图 13**。

4 用户指南

4.1 安装依赖

所需环境：libuv和CMake。

Linux 下安装 libuv：

```
apt install libuv1-dev -y
```

macOS 下安装 libuv，首先从 GitHub 下载源程序，通过以下方式编译：

```
sh autogen.sh
./configure
make
make check
make install
```

4.2 编译项目

libuv 安装完成后，需要将 `CMakeLists.txt` 中 `link_directories` 和 `include_directories` 的值改为 libuv 的动态链接库和头文件路径。

```
[INFO ] /Users/xqmmcqs/Documents/dnsr/src/query_pool.c:157 初始化查询池
[INFO ] /Users/xqmmcqs/Documents/dnsr/src/dns_client.c:88 启动client
[INFO ] /Users/xqmmcqs/Documents/dnsr/src/dns_server.c:94 启动server
[DEBUG] /Users/xqmmcqs/Documents/dnsr/src/dns_server.c:74 收到本地DNS查询报文
[DEBUG] /Users/xqmmcqs/Documents/dnsr/src/dns_print.c:23 DNS报文字节流：
0000 77 b7 01 00 00 01 00 00 00 00 00 00 08 74 69 6d
0010 65 2d 69 6f 73 01 67 07 61 61 70 6c 69 6d 67 03
0020 63 6f 6d 00 00 01 00 01
[DEBUG] /Users/xqmmcqs/Documents/dnsr/src/dns_print.c:166 DNS报文内容：
=======Header==========
ID = 0x77b7
QR = 0
OPCODE = 0
AA = 0
TC = 0
RD = 1
RA = 0
RCODE = 0
QDCOUNT = 1
ANCOUNT = 0
NSCOUNT = 0
ARCOUNT = 0

=======Question========
QNAME = time-ios.g.aaplimg.com.
QTYPE = 1
QCLASS = 1

=======Answer==========
=======Authority=======
=======Additional======
[DEBUG] /Users/xqmmcqs/Documents/dnsr/src/query_pool.c:39 添加新查询请求
[INFO ] /Users/xqmmcqs/Documents/dnsr/src/cache.c:98 查询cache
[DEBUG] /Users/xqmmcqs/Documents/dnsr/src/rbtree.c:40 在链表中查找元素
[INFO ] /Users/xqmmcqs/Documents/dnsr/src/cache.c:119 cache未命中
[DEBUG] /Users/xqmmcqs/Documents/dnsr/src/rbtree.c:463 查询红黑树
[INFO ] /Users/xqmmcqs/Documents/dnsr/src/cache.c:152 红黑树未命中
[INFO ] /Users/xqmmcqs/Documents/dnsr/src/dns_client.c:112 向服务器发送消息
```

图 **13:** 调试信息输出效果

之后，在项目目录中执行以下命令来编译：

```
mkdir build
cd build
cmake ..
make
```

编译完成后，运行：

```
./main
```

程序运行参数参考3.6.1节的说明，例如：

```
./main --log_path ./log.txt --remote_host 114.114.114.114
```

4.3 配置 DNS 服务器

对于 Linux 系统，执行如下指令：

```
echo 'nameserver 127.0.0.1' >> /etc/resolv.conf
```

对于 macOS 系统，在设置中，进入网络设置，如**图 14(a)**，点击“高级...”。之后按照**图 14(b)**中的提示操作，完成后点击**图 14(a)**右下角的“应用”。

(a) 网络设置页面　　(b) 高级设置

图 14: 网络设置

4.4 抓包与分析

对于 Linux 系统，首先使用 `ifconfig` 查看需要监听的网卡，例如选择 `eth0` 网卡，使用 `tcpdump` 指令监听 UDP 端口 53，进行抓包：

```
tcpdump -i eth0 udp port 53 -vv
```

产生的输出如下：

```
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 262144
↪  bytes
22:05:01.549890 IP (tos 0x0, ttl 64, id 35938, offset 0, flags [DF], proto UDP
↪  (17), length 58)
    km-ubuntu.40868 > 222.172.200.68.domain: [bad udp cksum 0x680e -> 0x6b26!]
    ↪  21821+ A? ourtale.love. (30)
22:05:01.550179 IP (tos 0x0, ttl 64, id 35939, offset 0, flags [DF], proto UDP
↪  (17), length 58)
    km-ubuntu.57546 > 222.172.200.68.domain: [bad udp cksum 0x680e -> 0x7e71!]
    ↪  177+ AAAA? ourtale.love. (30)
22:05:01.551235 IP (tos 0x0, ttl 64, id 35940, offset 0, flags [DF], proto UDP
↪  (17), length 73)
...
```

或者使用 Wireshark 软件，选择网卡后，在筛选栏中输入 `dns`，过滤出 DNS 报文，如图 **15(a)**所示。如果想要抓取本地进程和 DNS 服务端通信的报文，选择网卡 Loopback 即可，如图 **15(b)**。

5 测试结果

5.1 基础功能测试

三个基本功能的测试结果如图 **16**。对照表采用3.3.2节中的例子。

首先测试了对照表中的域名 `baidu.com`，得到预设的 IP，之后测试不在对照表中的域名 `apple.com.cn`，得到正确的 IP。

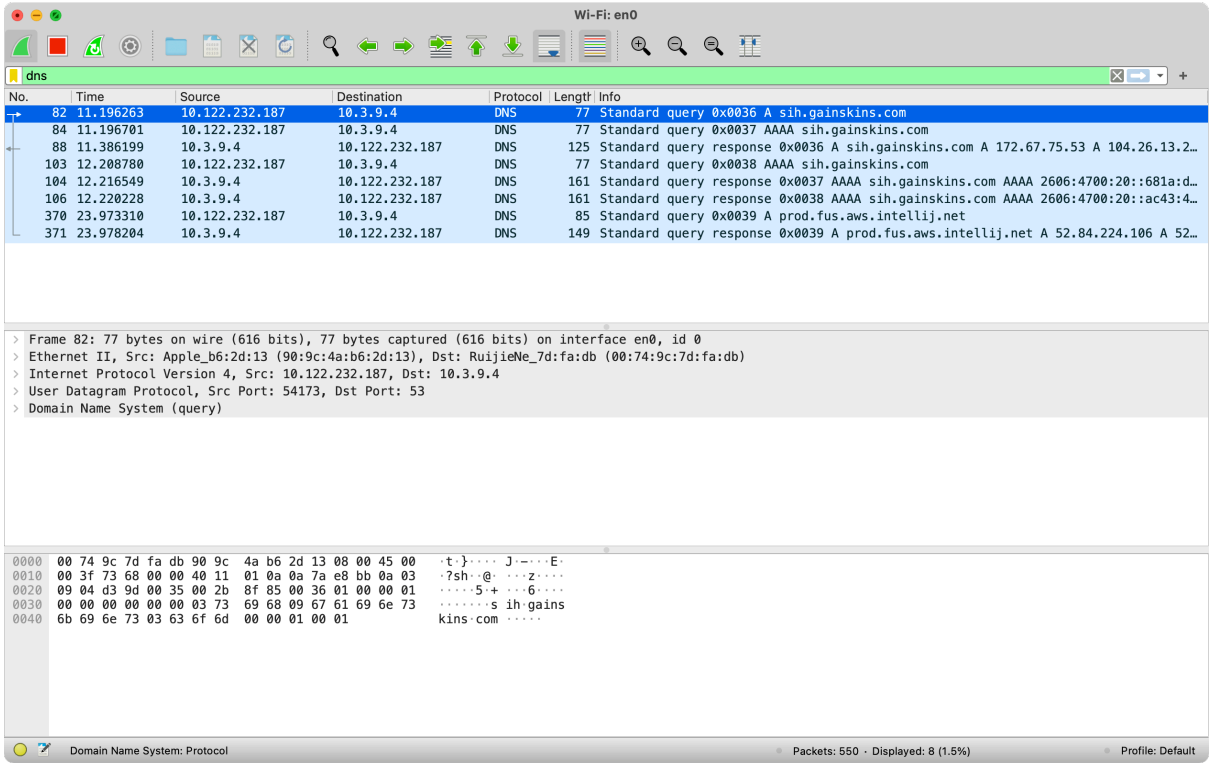
此外，测试了在对照表中的 IPv6 域名 `bt.byr.cn`，得到预设的 IP。最后分别以 A 类型和 AAAA 类型测试对照表中设为拦截的域名 `test.com`，会返回查询不到 IP 的结果。

再次查询 `apple.com.cn` 得到相同的结果，查看报文可知此域名的 TTL 是 10 分钟。

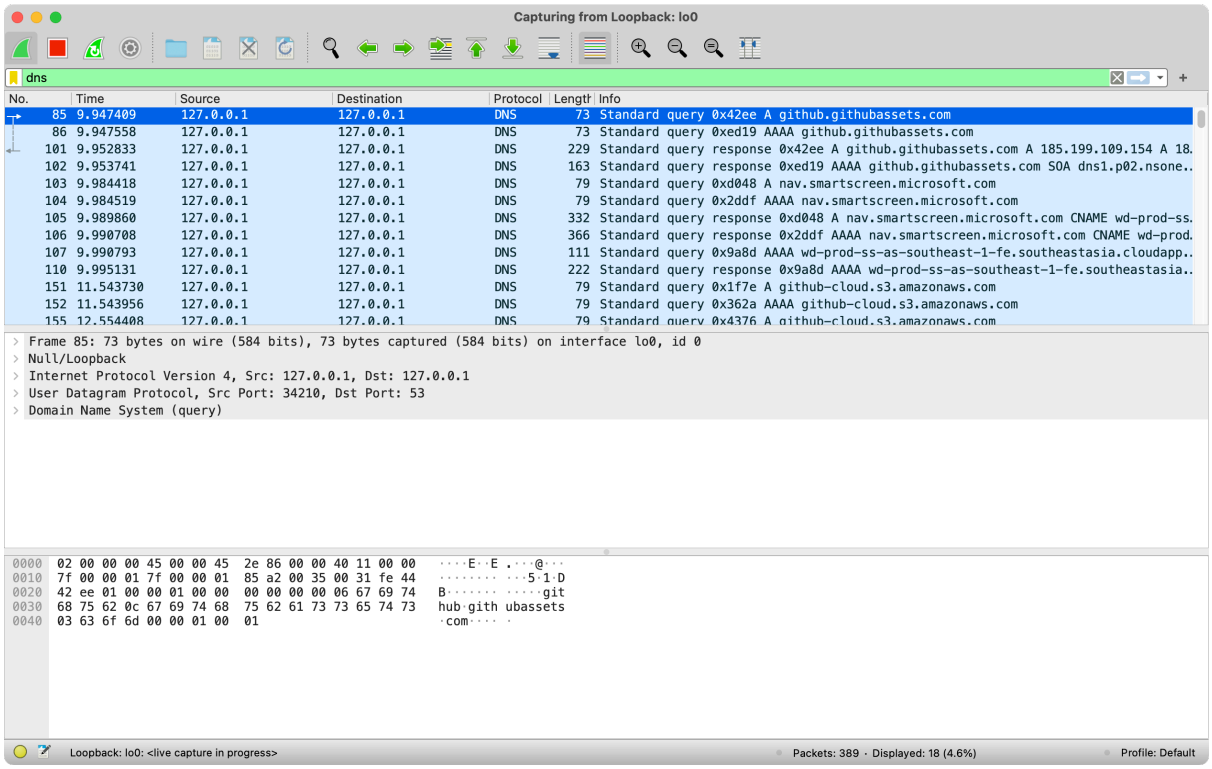
经过 10 分钟，再次查询 `apple.com.cn`，得到相同的结果，查看日志可知，中继服务器删除了红黑树中过期的节点，并且重新向服务器发送查询，如图 **17**。

5.2 手动构造报文测试

通过 `dnspython` 包测试功能，首先执行以下指令安装包：



(a) Wireshark 抓取 DNS 客户端和远程服务器通信的报文



(b) Wireshark 抓取本地进程和 DNS 服务端通信的报文

图 15: Wireshark 抓包

```
xqmmcqs@xqmmcqsdeMacBook-Pro ~ % nslookup
> baidu.com
Server:         127.0.0.1
Address:        127.0.0.1#53

Non-authoritative answer:
Name:   baidu.com
Address: 220.181.38.148
> apple.com.cn
Server:         127.0.0.1
Address:        127.0.0.1#53

Non-authoritative answer:
Name:   apple.com.cn
Address: 17.253.144.10
> set type=AAAA
> bt.byr.cn
Server:         127.0.0.1
Address:        127.0.0.1#53

Non-authoritative answer:
bt.byr.cn       has AAAA address 2001:da8:215:4078:250:56ff:fe97:654d

Authoritative answers can be found from:
> set type=A
> test.com
Server:         127.0.0.1
Address:        127.0.0.1#53

** server can't find test.com: NXDOMAIN
> set type=AAAA
> test.com
Server:         127.0.0.1
Address:        127.0.0.1#53

** server can't find test.com: NXDOMAIN
```

图 **16:** 基本功能测试结果

```
[DEBUG] /Users/xqmmcqs/Documents/dnsr/src/query_pool.c:39 添加新查询请求
[INFO ] /Users/xqmmcqs/Documents/dnsr/src/cache.c:98 查询cache
[DEBUG] /Users/xqmmcqs/Documents/dnsr/src/rbtree.c:40 在链表中查找元素
[INFO ] /Users/xqmmcqs/Documents/dnsr/src/cache.c:119 cache未命中
[DEBUG] /Users/xqmmcqs/Documents/dnsr/src/rbtree.c:464 查询红黑树
[DEBUG] /Users/xqmmcqs/Documents/dnsr/src/rbtree.c:30 删除链表中的元素
[DEBUG] /Users/xqmmcqs/Documents/dnsr/src/rbtree.c:420 删除红黑树中的节点
[INFO ] /Users/xqmmcqs/Documents/dnsr/src/cache.c:153 红黑树未命中
[INFO ] /Users/xqmmcqs/Documents/dnsr/src/dns_client.c:121 向服务器发送消息
[DEBUG] /Users/xqmmcqs/Documents/dnsr/src/dns_print.c:166 DNS报文内容:
```

图 **17:** 超时后重新查询的日志

```
pip install dnspython
```

手动构造报文：

```
from dns import message, query
q1 = message.from_text("id 3000\nopcode QUERY\nrcode NOERROR\nflags RD\n;QUESTION\nbaidu.com. IN A\n;ANSWER\n;AUTHORITY\n;ADDITIONAL")
r = query.udp(a, "127.0.0.1")
print(r.to_text())
```

上述代码构造了一个 A 类型 baidu.com 的 DNS 查询报文，并且向 DNS 中继服务器发出查询请求，运行结果如下：

```
id 3000
opcode QUERY
rcode NOERROR
flags QR RD RA
;QUESTION
baidu.com. IN A
;ANSWER
baidu.com. 4294967295 IN A 220.181.38.148
;AUTHORITY
;ADDITIONAL
```

我们用这个方法测试了同时发送多个相同编号 DNS 报文的情况，DNS 中继服务器对于每个报文重新编号并且正确地发送了回复。

5.3 稳定性测试

运行中继服务器 5 个小时，期间正常浏览网页、观看视频，服务器没有出现异常。

6 实验总结

6.1 调试中遇到的问题和解决方案

最初我们期望采用 windows 和 linux 系统原生 socket 接口实现 socket 编程，我们研究了阻塞式 I/O 以及常见的跨平台实现。之后我们找到了实现更简单、具有事件驱动和异步 I/O 功能的 libuv 库，于是决定采用 libuv 进行 socket 编程。

采用 libuv 实现多线程时，我们遇到了难以调试的内存泄漏和重复释放问题，在参考其他项目的实现之后，我们决定仅采用单线程 + 事件驱动的形式实现高并发，实测效果与多线程的方式差距不大。

此外，我们的红黑树实现参考了维基百科-红黑树中的实现，但是维基百科中的代码有一些漏洞，比如右旋函数没有判断当前节点是否为根，`delete_case` 函数中正确的代码被注释掉，这些错误可能导致红黑树中出现二元或者三元环。我们一一修复了这些问题，保证红黑树的正确性。

最后，我们参考常见的实现方法，完成了多个等级的调试信息输出，期间遇到了带有颜色的调试信息无法写入到文件的问题，因此我们需要判断当前输出的文件是否是 `stderr`，再输出颜色信息。

6.2 工作总结

本次实验中我们实现了一个功能比较全面的 DNS 中继服务器，在实用性和效率方面进行了许多改进，使其具有部署到应用环境的价值。

在实验的开始，我们详细研究了 DNS 协议的内容，阅读 RFC 1034 和 1035 文档的重要部分，并且参考一些博客，对报文数据结构的基本格式进行设计。在此期间我们的信息检索能力和英文文献阅读能力均得到了提升。最后，我们通过手动构造报文进行测试，将报文解析结果和 Wireshark 解析结果进行比对。

在掌握 DNS 协议的基础知识之后，我们对跨平台 socket 和高并发的实现进行了研究，最终采用 libuv 作为解决方案。通过学习 libuv 的文档和一些示例，我们实现了基础的报文转发功能，并且在高并发的环境下进行了初步的压力测试。

此后，针对高并发查询和重新编号的需求，我们实现了查询池和序号池，在不考虑缓存需求的情况下对这些部分进行了测试。测试结果显示直接中继能够正常运行。

针对缓存的需求，我们学习红黑树和 LRU 的相关知识，实现了高速缓存 + 红黑树的两层缓存结构，对于这一个模块进行了单独测试，最后将其整合进主题模块。

最后，我们实现了调试信息模块和命令行参数解析模块，并且参考Google 开源项目风格指南和一些其他开源项目，重新格式化了代码，添加了注释，更改了项目的目录结构，完成了文档的撰写工作。

在本次实验中我们主要有如下几个收获：

- 了解 DNS 协议有关知识，掌握 DNS 协议运作的原理，以及 DNS 报文的结构。
- 掌握跨平台的 socket 编程技术。
- 了解常见的高并发解决方案，理解多线程编程和事件驱动编程的方法。
- 学习 C 语言项目的编码规范，掌握 C 语言工程的开发和测试方式，了解完整 C 语言项目的流程，增强 C 语言编程能力。
- 增强英文文档和文献阅读能力和信息检索能力。

A 交互式 API 文档

由Doxygen生成的交互式文档。