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 3 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 实验环境

- 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

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. 输出调试信息和 DNS 报文内容。
2. 在转发 DNS 请求时重新分配序号，以区分不同的查询。

3. 避免阻塞式查询，采用事件驱动的方式实现高并发。
4. 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。
5. 实现高速缓存，对于经常访问的记录，加快查询效率。
6. 命令行参数解析。

3 模块介绍

3.1 DNS 报文解析和构建

3.1.1 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      |
+---------------------+
|      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。

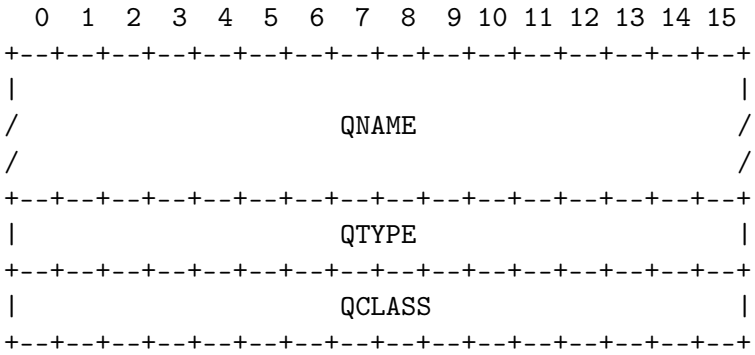
Header 部分的结构如下：

```
  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。

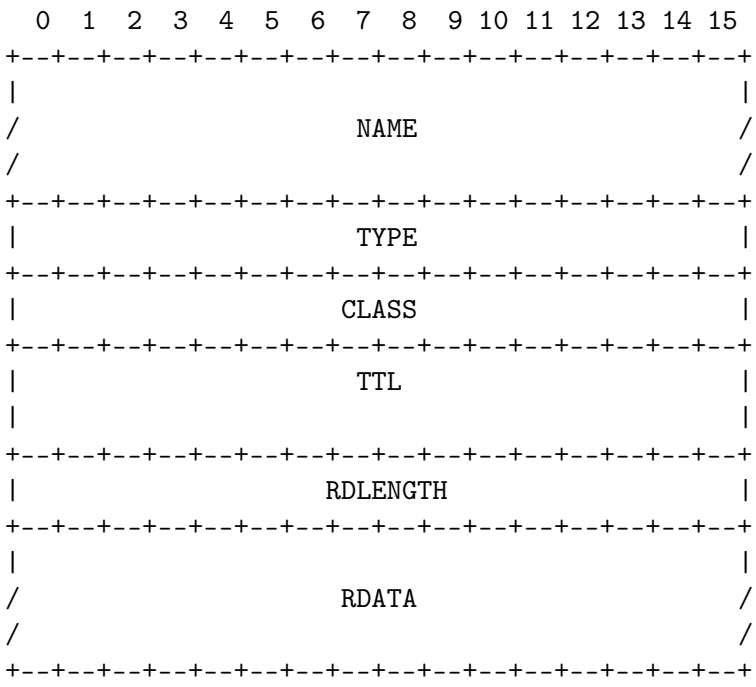
Question 部分格式 Question 部分的结构如下：



- QNAME，查询域名，格式在3.1.1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。

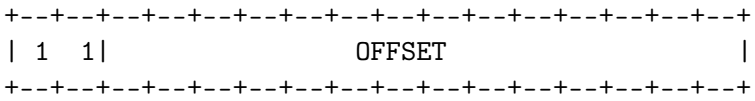
Resource Record 格式　Resource Record 的格式如下：



- NAME，此 RR 从属的域名，格式在3.1.1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。

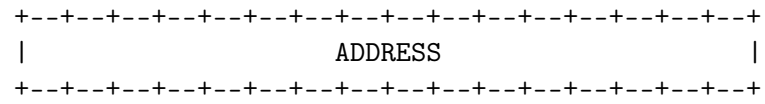
为了压缩 DNS 报文的长度，增大传输效率，引入了域名的压缩，此时 `<domain-name>` 可能由 0 或者一个指针终止，一个指针由两个字节构成，其格式如下：



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。

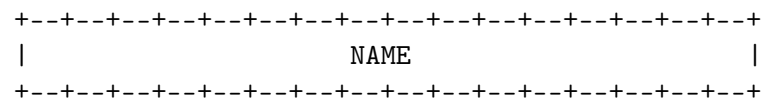
RDATA 格式 对于 DNS 中继服务器来说，RDATA 部分的内容并不重要，只是涉及到域名压缩，以及调试输出的需求，所以需要对一部分类型的 RDATA 作处理。需要处理的 RDATA 如下。

A 类型 RDATA 格式 如下：



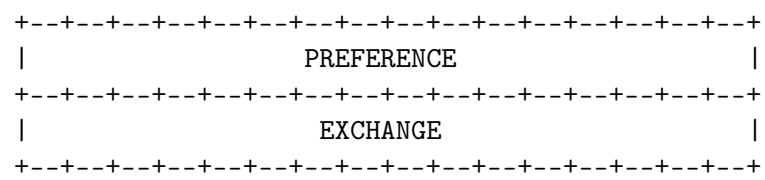
- ADDRESS，32 位 IPv4 地址。

NS/CNAME 类型 RDATA 格式 如下：



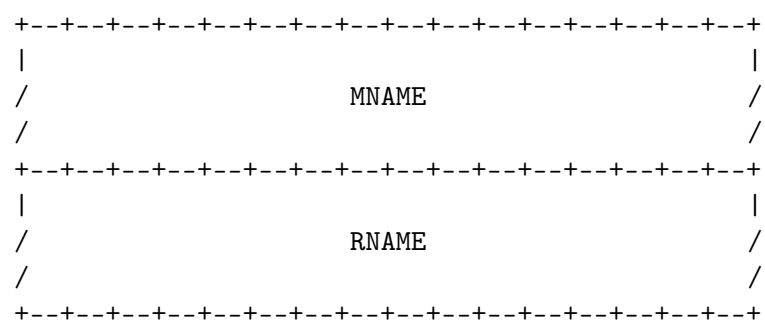
- NAME，一个域名。

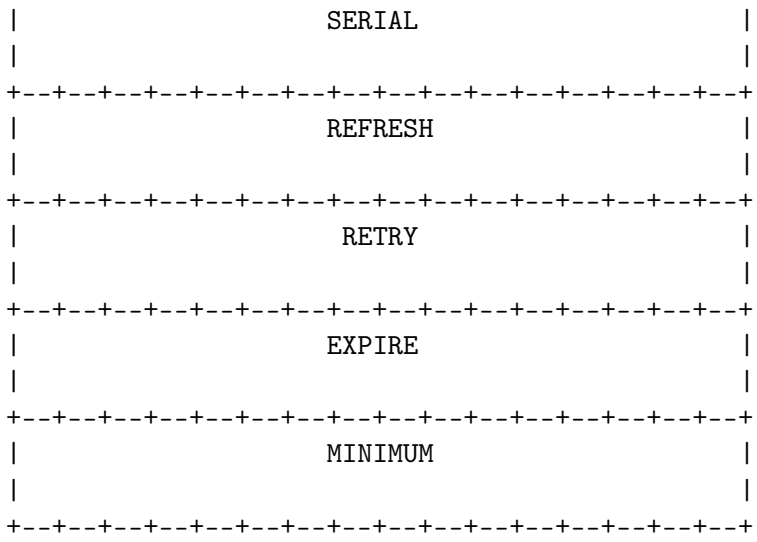
MX 类型 RDATA 格式 如下：



- PREFERENCE，16 位数字。
- EXCHANGE，一个域名。

SOA 类型 RDATA 格式 如下：





- 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，128 位 IPv6 地址。

3.1.2 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 1所示。

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：

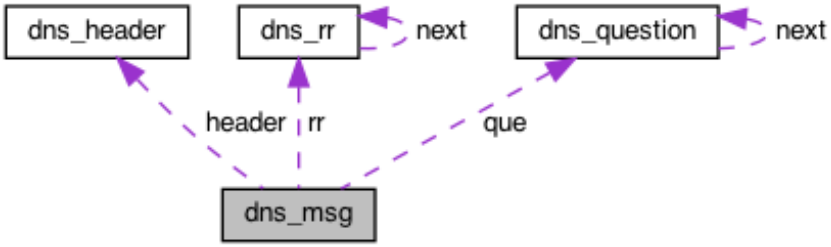


图 1: DNS 报文结构体的结构

```c
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;
```

```
41      Dns_RR * rr;
42  } Dns_Msg;
```

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

3.1.3 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 2。

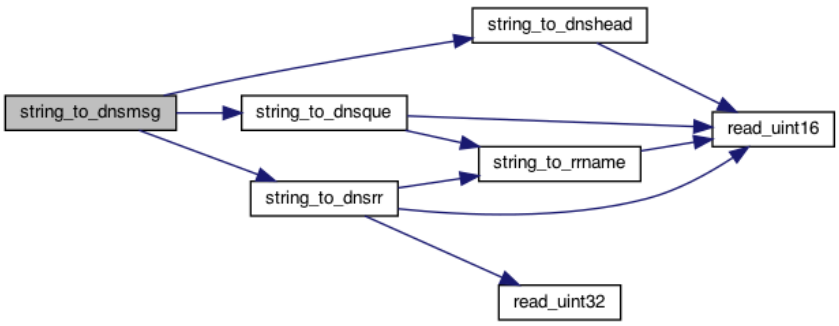


图 2: string_to_dnsmsg 函数调用图

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 3。

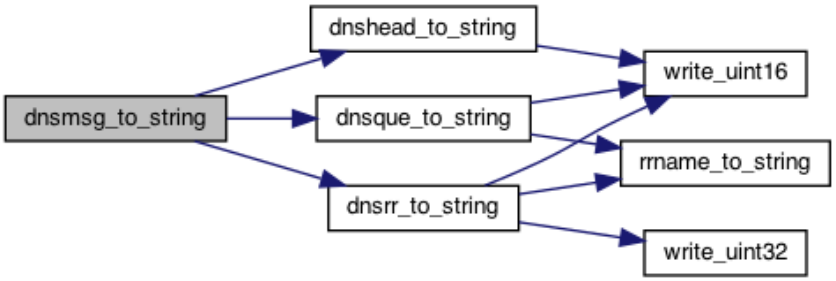


图 3: `dnsmsg_to_string` 函数调用图

`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。

3.1.4 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 4。

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用于输出不同类型的 RDATA 字段。

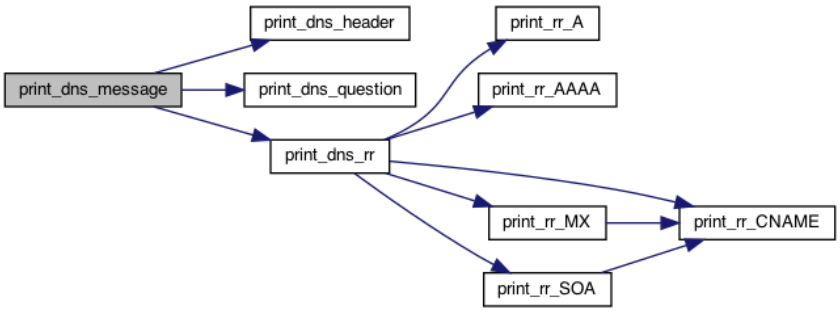


图 4: `print_dns_message` 函数调用图

输出的效果如下：

```
=======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。

3.2 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。

3.2.1 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 5所示。

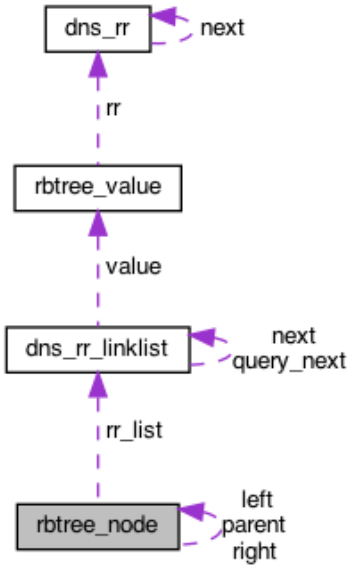


图 5: 红黑树和 RR 链表结构体的关系

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。

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 6**所示。

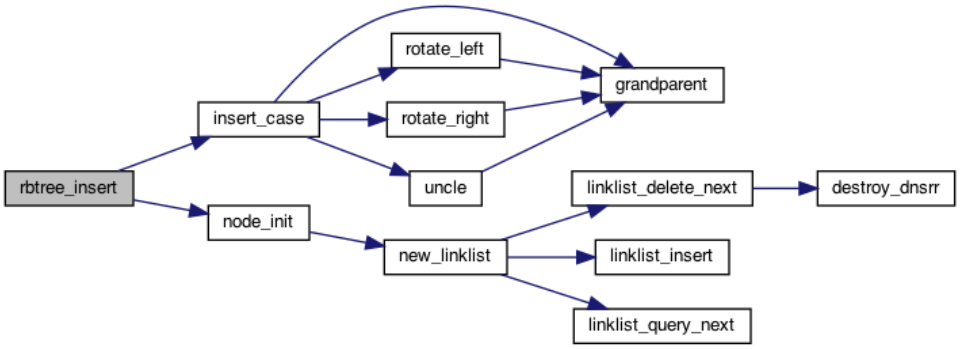


图 6: `rbtree_insert` 函数的调用图

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 7**所示。

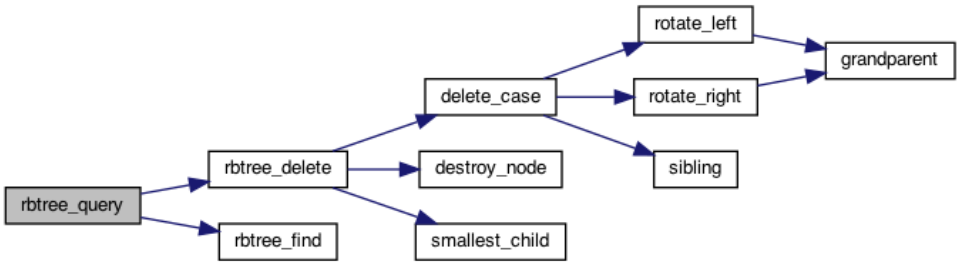


图 7: `rbtree_query` 函数的调用图

上面提到的函数接口的的文档见rbtree.c 文件参考。

3.2.2 LRU 高速缓存

缓存的插入操作

缓存的查询操作