DNS 缓存系统系统设计说明书

v1.0

2011-6-25

修改记录

No	修改后版本号	修改内容简介	修改日期
1	v1.0	第一版	2011.06.23

目 录

1 引言

1.1 编写目的

本文档详细介绍了 DNS 缓存系统的系统设计，描述出一个具体的产品设计模型，为开发及测试人员提供下步工作的依据。

1.2 背景

本系统的开发满足需求设计中对本系统的要求。它提供 DNS 缓存服务，网络中的主机可以一次向本系统提出多个 DNS 请求，系统以尽快的速度予以响应。这样就避免了客户端频繁的 DNS 请求，加快了客户端的速度。另外，客户在一次请求中可以请求多个域名。本系统可以同时为多个客户端服务，采用多线程的方式来加快处理速度，初步设计为可以承受 5000reqs/s 的工作负担。采用比 LRU 效率更高的 LIRS[1] 技术来管理本地的缓存，可以提高系统的缓存命中率，减小了响应时间。

2 总体设计

2.1 软件体系结构

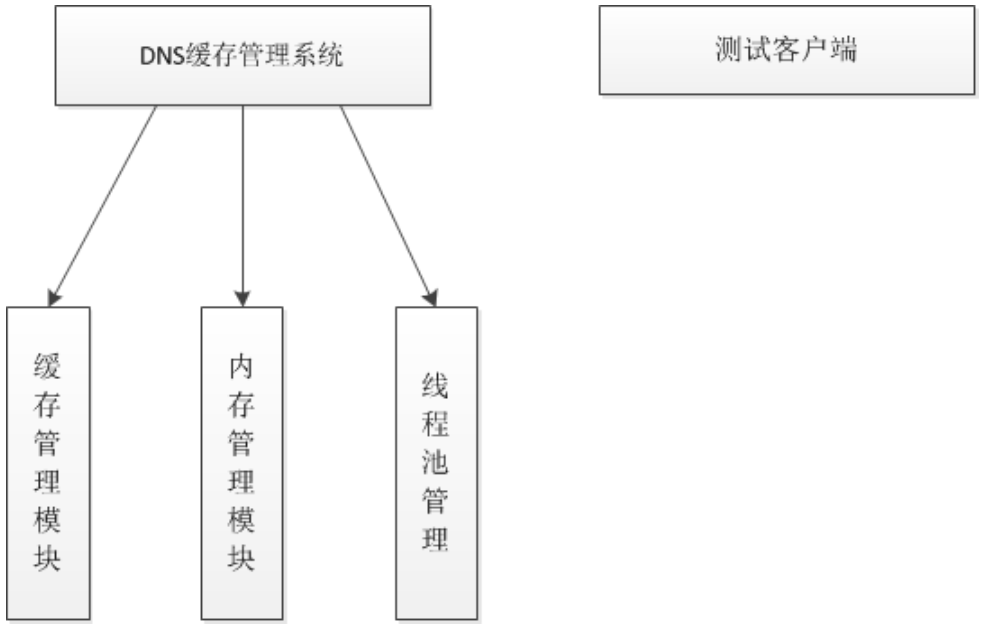

图 1: DNS 缓存系统子系统层次图

DNS 缓存系统是一个 DNS 的一个高速缓存服务器，提供 DNS 缓存服务。传统的 DNS 服务器一次只能查询一个域名，而且响应速度慢。本系统将为客户端代理 DNS 请求，把 DNS 查询结果返回给客户端，并且把结果缓存在本地。当客户下一次请求同样的域名时，就可以在本地的缓存中查找，如果缓存中存在未被替换的结果，则返回该结果；否则，系统重新请求 DNS 服务器，并将 DNS 服务器返回结果存入缓存中。客户可以在一次请求中查询多个域名的 IP 地址，本系统予以响应。我们把整个系统分为三个大的模块:

- **缓存管理模块：** 把从 DNS 处获得的 IP 地址在本地缓存。域名使用字典树 [2]在本地组织，从而加快对域名的检索速度。在缓存区满了以后，采用比传统 LRU 算法效率更高的 LIRS[1]算法，进行缓冲区的更新。

- **内存管理模块：**为了加快系统的响应速度，减少动态申请内存带来的时间损耗，为请求报文预分配一块内存区域，当客户有连接时，从用户报文的前四个字节中获取整个报文的长度，然后从内存管理模块中申请（不是动态申请）响应长度的内存区域。这一块内存还将被用于构建返回报文。
- **线程池管理模块：**为了减少创建和销毁线程所带来的系统开销，在系统中采用线程池技术。在系统启动时，创建一定数目的线程。主线程负责接受用户的请求，然后调用线程池中的线程为客户服务。线程池管理模块负责线程的创建，申请和回收。
- **测试客户端：**用于测试本 DNS 缓存系统是否正确工作，同时也测试它的并发性能。

2.2 运行系统

2.2.1 运行体系图

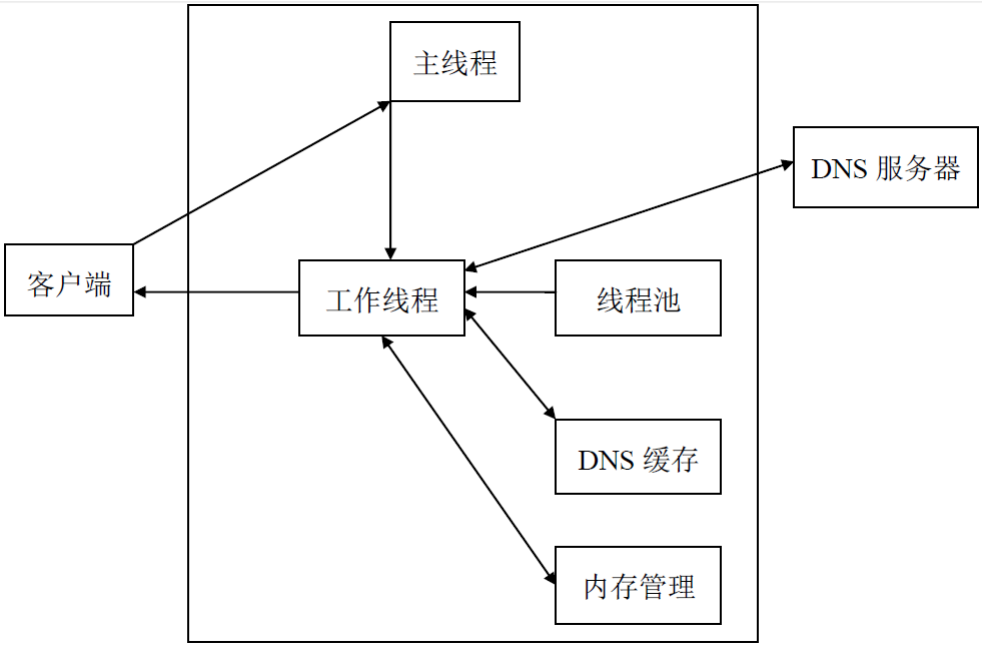

图 2: DNS 缓存系统子系统层次图

2.2.2 程序/模块对应表

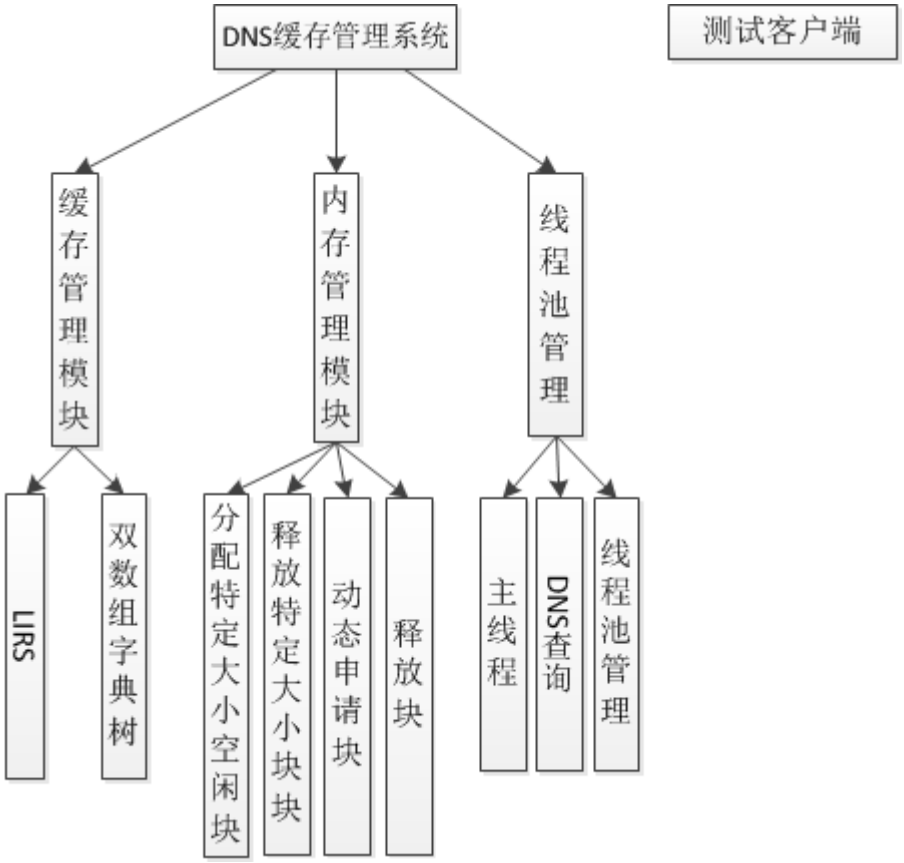

图 3: 程序/模块对应表

2.3 系统物理结构

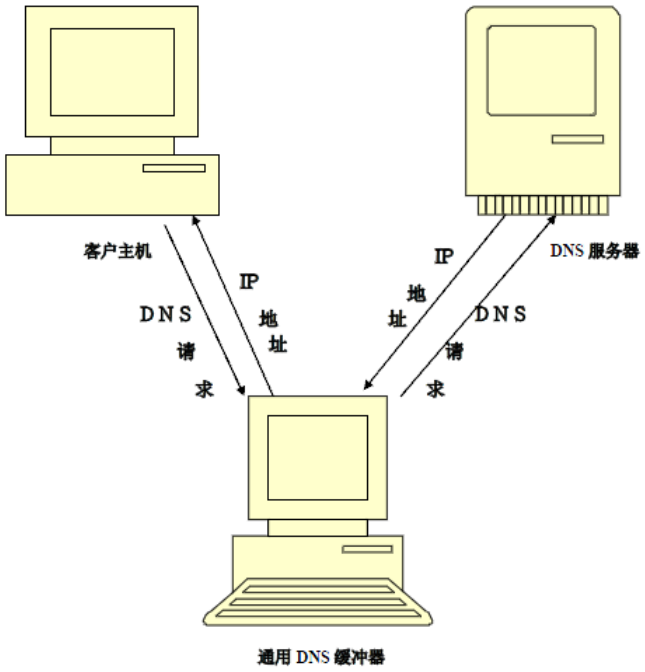

图 4: 系统物理结构图

3 系统环境

3.1 开发环境

环境	工具
操作系统	ubuntu11.04 x86_64
内核版本	2.6.38-8
编译器	gcc 4.5.2

表 1: 开发环境

4 设计思路及折衷

4.1 TCP OR UDP?

由于可以在一个请求中接受多个域名的查询，所以数据包可能会非常大。UDP 缺乏流控制和错误控制，在数据量比较小的时候使用 UDP 非常合适。我们一个请求和应答的数据包可能达到几十 K，考虑到客户端的网络流量，如果使用 UDP，肯定会发生错误或丢包，在应用程序中处理错误和重传会造成资源浪费，加重服务器的负担，也会造成网络的拥堵，同时也加大了客户的响应时间。

而在数据量比较小的时候，使用 UDP 就非常合适。所以我们初步设计为提供两个服务器，一个 UDP 服务器，和一个 TCP 服务器。客户端根据它的报文大小来选择使用 UDP 还是 TCP。

4.2 为什么是 LIRS 而不是 LRU?

根据资料显示，LIRS 的效率比 LRU 高，但是实现的代价比 LRU 高不了多少，所以使用 LIRS。

4.3 为什么是字典树，而不使用 HASH 函数?

采用字典树与 HASH 函数相比，有以下几点优势：

1. 可以利用字符串的公共前缀来节约存储空间
2. 查找效率比 HASH 高 [5]
3. HASH 可能产生碰撞

4.4 为什么使用两个双数组字典树?

字典树的查询效率非常高，但是动态构造构造的过程非常慢，需要做大量的内存拷贝。在设计中使用两个双数组字典树。把访问频度最高的几百万（可配置的数目）个域名存放在文件中，在系统启动时，从文件中读入域名信息，用于构造主字典树（在已知大小时，字典树构造速度非常快，因为不需要内存的动态分配和拷贝操作）。在查询域名时，首先从主字典树中查询，如果未命中，就把相应域名加入副字典树中（副字典树比较小，

动态构造小的字典树代价很低)。这种方式，既保留了字典树的较高的查询速率，又避免了动态构造很大的字典树带来的开销。

5 模块设计

5.1 线程池设计

在系统开始运行时创建 MAXTHREAD 个线程，放入线程池中。主线程创建监听套接字，开启服务。每个线程各自调用 accept，采用互斥锁来保证在每一个时刻只有一个线程调用 accept。每个线程在 accept 后为客户端服务。

由于 TCP 内部为监听套接字维护两个队列：a. 已完成队列，b. 未完成队列。所以在主线程睡眠期间，新的客户连接会使这两个队列充满（两个队列之和不超过在 listen 时指定的 backlog)，而在这两个队列满了之后当一个客户的 SYN 到达时，TCP 就忽略该分节，也就时说，并不返回 RST。这样做是因为：这种情况是暂时的，客户将重发 SYN ，期望在不久就能在这些队列中找到可用空间 [3]。所以当这两个队列充满以后，客户只是重发 SYN ，而服务器不会接受客户端创建更多的连接。

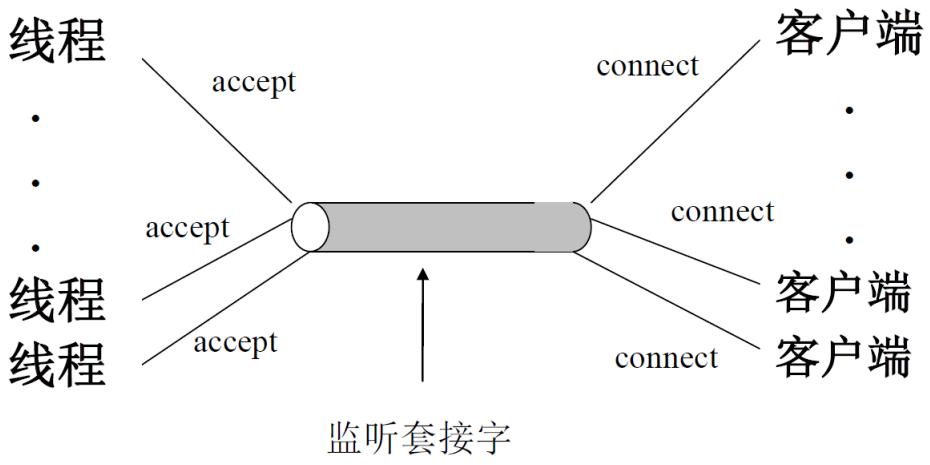

图 5: 多线程模型

5.2 工作线程设计

线程池中的工作线程首先调用 connect，接受一个客户端的连接，然后从报文的前 4 个字节中获取报文的长度，从内存管理模块申请响应大小的内存，用于存放请求报文。在收到请求后，逐次扫描每个域名。获得域名后，首先在本地的缓存中查找域名对应的 IP，如果有记录，就把 IP 地址填写到返回报文中，如果没有记录，则填入 127.0.0.1，表明本地没有缓存，将会进行 DNS 查询，结果在第二个报文中返回。然后把这个域名相应的序号和域名填入到下图所示的链表中。

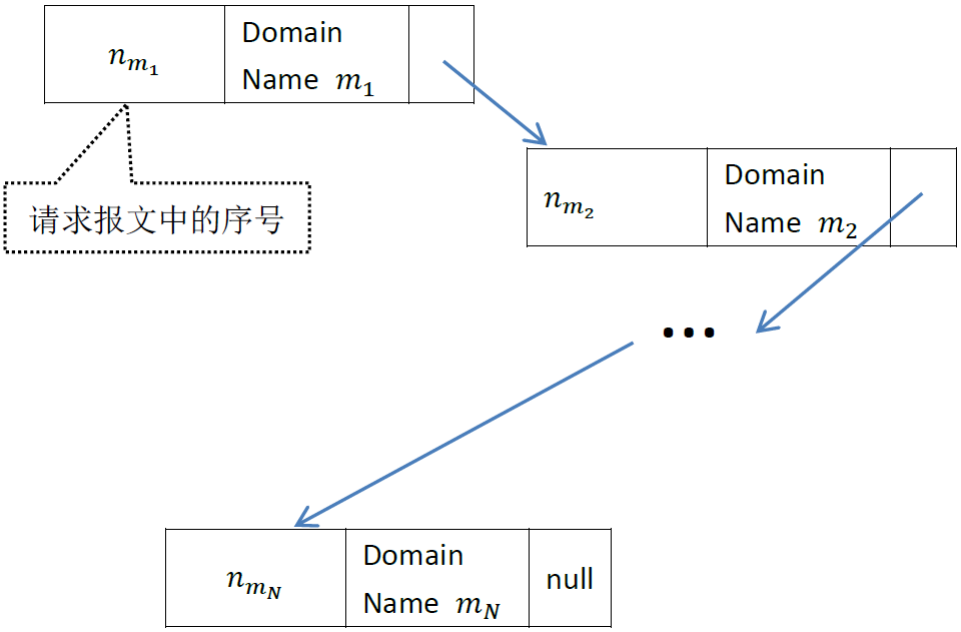

图 6: 本地缓存未命中的域名构成的链表

工作线程在把请求报文遍历完毕后，就把第一次返回报文发送给客户端，同时也把所有本地没有缓存的域名构造成了一个链表。如果链表为空，说明请求的域名在本地都有缓存，任务完毕，关闭连接。否则，根据链表中的数目，申请用于构造第二次返回报文的内存空间。对链表进行遍历，对链表中的每个域名调用一个 DNS 查询线程进行 DNS 查询，在最后一个 DNS 查询完成后，把第二个报文发送给客户端，关闭连接。

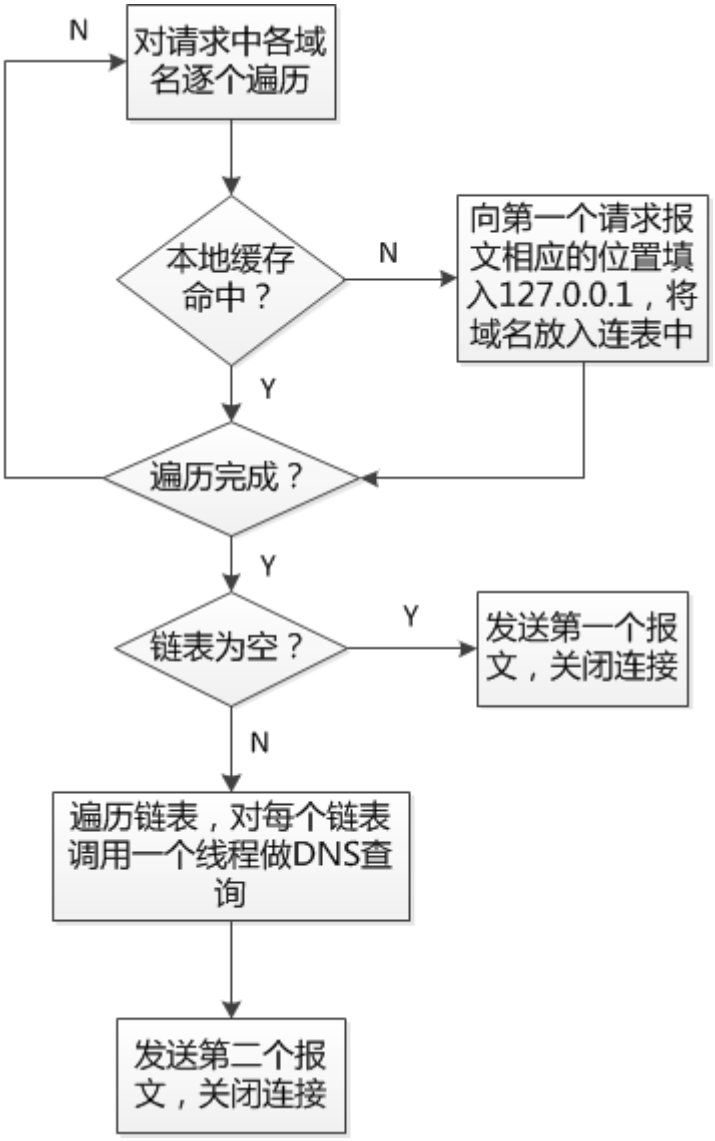

图 7: 线程工作流程图

5.3 内存管理模块

模块从系统预先申请 256B，512B，1KB，2K,...，2^nKB 等 2 幂次大小的块各若干。通过 mem_chunk 结构维护特定大小的内存块大小的幂级，预分配的数目，已被使用的块链表，空闲的块链表。指针数组维护所有大小块的 mem_chuck 结构指针。当线程申请使用 M KB($2^m < M < 2^n$) 空间时，

若 2^nKB 大小块仍有空闲块，则分配给该线程，并将该块移动至已使用块链表中；若上述大小的块不存在空闲或者不存在该大小的块，则模块动态申请若干数目该大小块的内存区域，分配之。当线程释放某大小块时，若检测到释放的块位于模块动态申请的内存中且这部分内存已没有线程使用，则释放该内存。

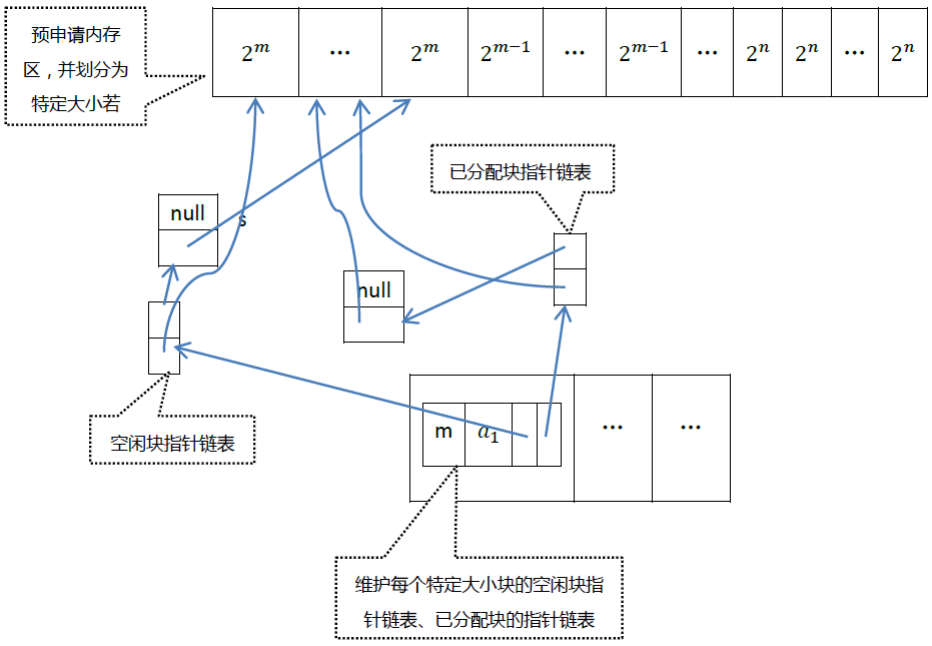

图 8: 内存管理示意图

5.4 缓存管理模块

缓存管理模块由 DATrie 和 LIRS 栈驱动。 **定义 1：**双数组 Trie（Double-Array Trie，DATrie）是 trie 树的一个简单而有效的实现，由两整数数组构成，一个是 base[]，另一个是 check[]。check[base[s] + c] = s base[s] + c = t.

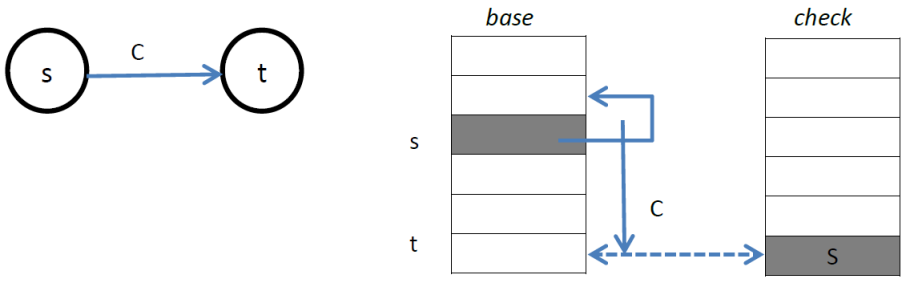

图 9: DATrie 的状态转换示意，base[s]+c=t, check[t]=s

对域名集合的索引是有两个双数组字典 (Double-Array Trie，DATrie) 驱动的。其中一个 DATrie 较大，存放所有已知域名集合，是静态的；另一个 DATrie 较小，实际应用中作为较大的 DATrie 的补充，是动态变化的。每当动态 DATrie 中记录达到一定数目时，就会离线补充静态 DATrie 中，失效域名从静态 DATrie 中剔除。当有请求查询缓存是首先访问静态的 DATrie，若没有命中则访问动态 DATrie，当两者均未命中时选择向动态 DATrie 中插入记录。

静态 DATrie 由离线生成，并且不断从动态 DATrie 中学习补充。

域名索引不仅是对缓存内容索引，由于 DATrie 查询字符串速度快，但插入或删除效率较低因此采用上述两个 DATrie 结合的缓存方法。 **定义 2**:最短最近使用间隔算法 (Low Inter-Reference Recency Set,LIRS):LIRS 算法是一中基于 LRU 算法弱点而改进的算法，使用页面的最近实用间隔 (Inter-Reference Recency,IRR) 来决定要替换的页面。IRR 用来表示一个页面的最近两次访问的间隔中的其他无重复页面的个数。LIRS 用来表示一个页面的最近两次访问的间隔中的其它无重复页面的个数。LIRS 还定义了一中不同的最近访问时间 R(Recency,R)，R 用来表示一个页面的最近访问至当前访问之间的其它无重复页面的个数。

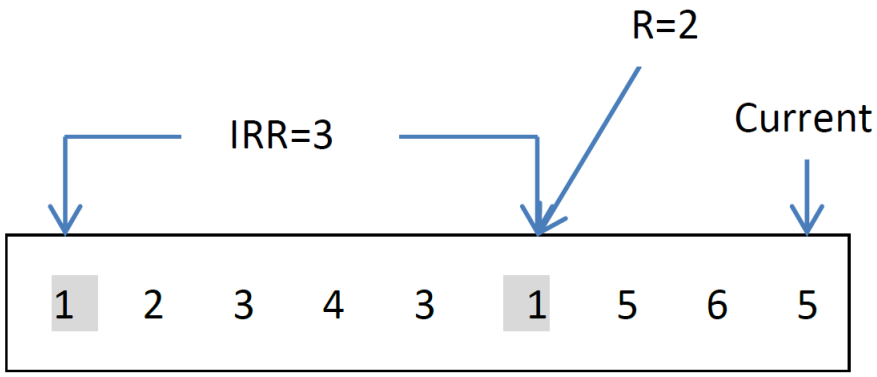

图 10: LIRS 算法的 IRR 和 R 的示意

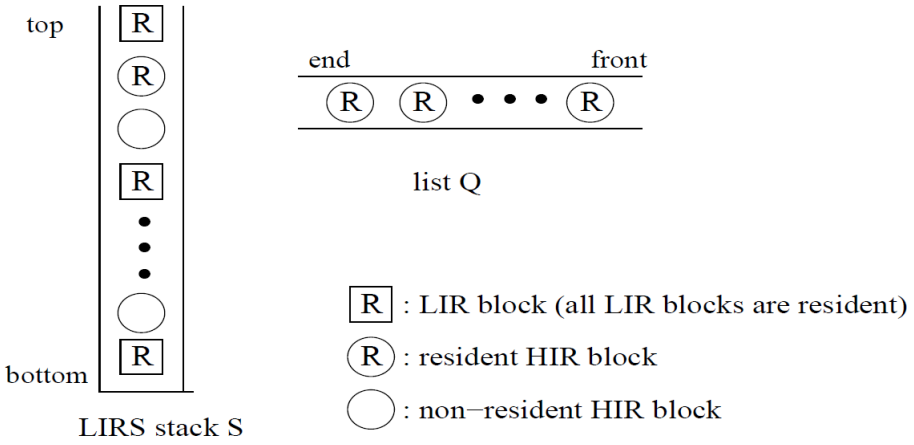

图 11: LIRS 栈 S 维护 LIR 块，同时也维护常驻或非常驻状态的 HIR 块，链表 Q 维护所有常驻的 HIR 块。

6 通信协议

6.1 请求报文

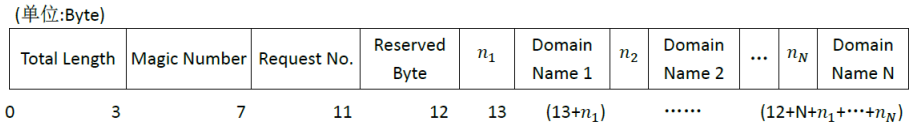

图 12: 请求报文格式

• 报文总长度 (Total Length) 字段: 占 4 字节。用来指定请求报文的总长度。

• 魔数 (Magic Number) 字段: 占 4 字节。用来区分 UDP 有效报文和垃圾报文。

• 请求序列号 (Request No.) 字段: 占 4 字节。用于标示同一台主机向系统请求 DNS 解析的请求编号。

• 保留字节 (Reserved Byte): 占 1 字节。用于返回报文的错误控制、返回顺序等扩展。

• 域名长度 (n) 字段: 占 1 字节。用于指定后续字节中域名的长度。

• 域名 (Domain Name) 字段: 不定长字节数。给出要解析的具体域名。

6.2 返回报文 1

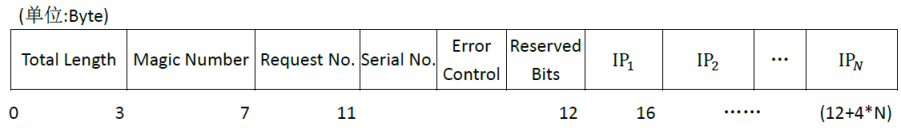

图 13: 返回报文 1

• 报文总长度 (Total Length) 字段: 占 4 字节。用来指定返回报文的总长度。

• 魔数 (Magic Number) 字段: 占 4 字节。用来区分 UDP 有效报文和垃圾报文。

• 请求序列号（Request No.）字段：占 4 字节。用于标示同一台主机向系统请求 DNS 解析的请求编号。

• 序列号（Serial No.）字段：占 1 比特。0 表示第一次向特定请求返回报文。

• 错误控制（Request No.）字段：占 1 比特。用于标示后续字节中存在域名解析失败的情况。

• 保留字位（Reserved Bits）：占 6 比特。用于返回报文将来可能需要的其他扩展。

• 网际协议地址（IP）字段：占 4 字节。用于给出请求报文中相应域名的对应 IP 地址，或者相应的错误代码。

6.3 返回报文 2

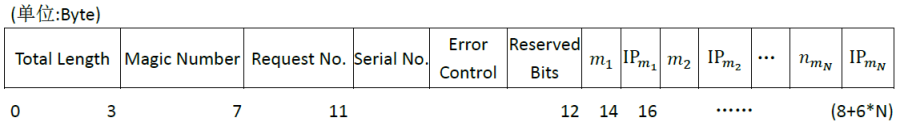

图 14：返回报文 2

• 报文总长度（Total Length）字段：占 4 字节。用来指定返回报文的总长度。

• 魔数（Magic Number）字段：占 4 字节。用来区分 UDP 有效报文和垃圾报文。

• 请求序列号（Request No.）字段：占 4 字节。用于标示同一台主机向系统请求 DNS 解析的请求编号。

• 序列号（Serial No.）字段：占 1 比特。1 表示第二次向特定请求返回报文。

• 错误控制（Request No.）字段：占 1 比特。用于标示后续字节中存在域名解析失败的情况。

• 保留字位（Reserved Bits）：占 6 比特。用于返回报文将来可能需要的其他扩展。

• 对应位置（m）字段：占 2 字节。用于给出后续 IP 地址在请求报文中对应域名的序号。

• 网际协议地址（IP）字段：占 4 字节。用于给出请求报文中相应域名的对应 IP 地址，或者相应的错误代码。

7 接口设计

7.1 公共接口

7.1.1 单链表

单链表提供一组创建链表，往链表中增加条目，删除条目，和删除链表的接口。链表的节点的数据结构为：

```
1 struct sl_node {
2         void * data;
3         struct sl_node * next;
4 };
```

链表的结构体为：

```
1 struct slist{
2         struct sl_node * head;
3         struct sl_node * end;
4         struct sl_node * blank;
5         size_t length;
6         size_t capacity;
7         struct sl_node * memlist;
8 };
```

创建链表

```
#include "slist.h"
struct slist * mk_slist(void * (*my_alloc)(size_t),
                                                size_t capacity)

                                                返回值：创建的链表
```

功能： 创建一个新的链表。
参数： my_alloc() 用来分配内存的函数指针，capacity，链表的容量。
返回： 创建的链表

```
#include "slist.h"
struct slist * sl_expand(struct slist *sl,
                    void * (my_alloc) (size_t),size_t delta);

                                        返回值：指向链表的指针
```

功能： 当预分配的内存不够用的时候增加链表容量
参数： 链表指针，分配内存的函数和增加的大小
返回： 指向链表的指针

```
#include "slist.h"
int append(void * data, struct slist * sl);

                                返回值：若成功则为 0，若没有内存 1
```

功能： 向链表尾部添加一个节点
参数： data，要存放的数据指针。sl，操作的链表指针。
返回： 若成功则返回 0，返回 1 表示内存不足，函数执行失败。

```
#include "slist.h"
void * pop(struct slist * sl);

                                        返回值：无
```

功能： 从链表的头部弹出一个节点，并返回这个指向这个节点的指针。
参数： 链表指针
返回： 指向弹出节点的指针。

```
#include "slist.h"
int push(void * data, struct slist * sl);

                                返回值：若成功则为 0，若没有内存 1
```

功能： 在链表头部前面插入一个节点。

参数： data，插入节点的数据指针，sl，操作的链表。
返回： 若成功则返回 0，返回 1 表示内存不足，函数执行失败。

```
#include "slist.h"
void sl_free(void (* my_free)(void *),struct slist * sl);

                                返回值：若成功则为 0，若没有内存 1
```

功能： 在链表头部前面插入一个节点。
参数： data，插入节点的数据指针，sl，操作的链表。
返回： 若成功则返回 0，返回 1 表示内存不足，函数执行失败。

7.2 内存管理模块向外提供的接口

内存管理模块向外提供两个接口——内存申请函数和内存释放函数。

内存申请函数

```
#include "dc_mm.h"
void * dc_alloc(size_t size)

          返回值：若成功返回分配内存的首地址，若出错则返回 NULL
```

参数： size 的单位为字节，表示想要申请的字节数。
返回： 返回一个指向所分配空间的 void 类型指针。如果 size 为 0，则返回 NULL 或一个可以被 dc_free() 成功释放的一个特定的指针。

内存释放函数

```
#include "dc_mm.h"
void dc_free(void * ptr)

                                返回值：无返回值
```

参数： 将要被释放的内存的首地址指针 ptr。
返回： 无
说明： dc_free() 释放由 dc_alloc() 分配的内存。如果先前已经调用了 dc_free(ptr)，则产生未定义操作，可能产生严重后果。如果 ptr 为 NULL，就什么也不做。

7.3 缓存接口对外接口

参考文献

[1] Song Jiang, Xiaodong Zhang , LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance.ACM SIGMETRICS Performance Evaluation , 2002 -portal.acm.org. 1, 2

[2] Theppitak Karoonboonyanan, An Implementation of Double-ArrayTrie. 2

[3] W.Richard Stevens,Bill Fenner, Andrew M.Rudoff.《UNIX 网络编程 (卷 1): 套接字联网 API(第 3 版)》. 人民邮电出版社. 7

[4] W.Richard Stevens.《UNIX 网络编程 (卷 2): 进程间通信 (第 2 版)》. 人民邮电出版社.

[5] 王思力, 张华平, 王斌. 双数组 Trie 树算法优化及其应用研究 6