一、 系统功能设计

1. 基本任务与要求

设计一个 DNS 中继服务器程序，读入“域名-IP 地址”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，需要实现三种功能，即对于以下三种检索结果进行不同处理：

- **不良网站拦截功能**：检索结果为 IP 地址 0.0.0.0，则向客户端返回“域名不存在”的报错信息；
- **服务器功能**：检索结果为普通 IP 地址，则向客户返回这个地址；
- **中继器功能**：表中未检索到该域名，则向因特网 DNS 服务器发出查询，并将结果返回给客户端；

设计程序时需要考虑以下两个问题：

- **多客户端并发**：允许多个客户端（可能会位于不同计算机）的并发查询，即：允许第一个查询尚未得到答案前就启动处理另外一个客户端查询请求；
- **超时处理**：由于 UDP 的不可靠性，考虑求助外部 DNS 服务器时却不能得到应答或者收到迟到应答的情形如何处理。

2. 整体流程图

对于基本任务提到的三个功能，可以分为向表/向远程服务器获取 IP 两种方案，其主要流程如图一所示，具体实现参见后文。

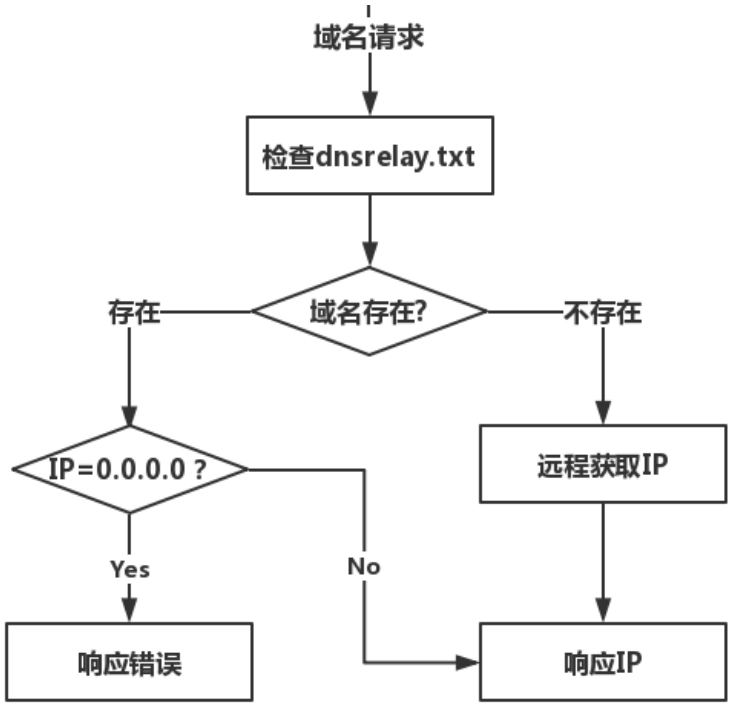

图 1 整体流程图

3. 问题考虑

针对高并发、超时处理两个问题，本次采取了以下解决方案：

- 多客户端并发：
  - 采用 UDP 无连接传输协议，较易支持服务端与多个客户端通信；
  - 解耦功能模块，采用多线程/多进程技术并发执行，由于 Python GIL 全局锁的原因，多线程效率比较低，本次采用多进程与管道通信技术；
- 超时处理：
  - 设置超时时间，降低远程查询对服务性能的影响；
  - 记录发送包 ID，持续监听远程回复，避免迟到回复/永久不回复的问题；

二、 软件设计与模块划分

1. 技术选型与环境依赖

本次 DNS 服务器开发技术主要是基于 struct 工具库的字节流序列化/反序列化的 socket 通信，DNS 协议封装参考 RFC1034 及 RFC1035，相关说明如下：

操作系统	Windows10 家庭版
开发语言	Python 3.6
主要依赖包	multiprocessing：多进程库
	socket：socket 通信库
	struct：字节流解析/序列化工具库

2. 模块划分

从文件上划分，服务由入口程序 start.py、服务器类 DNSRelayServer.py、协议解析工具函数 DNSProtocol.py 三部分构成，下面是具体介绍：

- **入口程序 start.py**：解析命令行参数并启动服务，-r 参数指定远程 DNS 服务器、-t 参数指定本地 IP-域名表、-b 参数指定日志开关。
- **服务器类 DNSRelayServer.py**
  - 成员变量
    - remote_dns：远程 DNS 服务器地址
    - local_dns_file：本地 IP-域名表数据
    - s_listener：负责与客户端通信的 socket（localhost:53）
    - id_addr：DNS 报文 ID 与客户端地址映射表
    - id_data：DNS 报文 ID 与客户端发送数据映射表
    - id_dname：DNS 报文 ID 与客户端请求域名映射表
    - printSwitch：全局日志输出开关，建议使用时关闭，开启会造成一定程度上的性

能损耗

- 成员函数
  - dns_load()：加载本地 IP-域名表数据至成员变量 local_dns_file
  - dns_update_buffer()：更新本地缓存表，到达一定量后自动清除
  - dns_server_listener(send_queue, help_queue)：持续运行进程函数之一，负责监听 53 端口里客户端发送的 DNS 域名解析请求，它首先会校验请求合法性（必须是查询报文 QR=0， 查询必须是 A 类型 QTYPE=1），对于合法请求会根据本地 IP-域名表/缓存检索结果来分发任务，如果在 IP-域名表/缓存中存在请求域名，则连带对应 IP 加入发送队列 send_queue，准备发回客户端（0.0.0.0 非法拦截延后至发送时处理）;如果不存在,则将请求报文加入查询队列 help_queue，准备向远程 DNS 查询
  - dns_server_sender(send_queue)：持续运行进程函数之一，负责通过 53 端口向客户端发送回包，它首先检查发送队列 send_queue 是否为空，若不为空则将队列内所有请求包重组为响应包，再不断发回客户端
  - dns_server_helper(send_queue, help_queue)：持续运行进程函数之一，负责向远程服务器查询请求解析未知域名，它首先检查查询队列 help_queue 是否为空，若不为空则将队列内所有客户端请求包重组后发送给远程 DNS 服务器（这里设计上的难点是 ID 的重新分配，本次实验中采取在不重复的约束下随机生成 ID 方案），同时不断监听远程服务器的响应，在获得正确响应后将相关数据加入发送队列 send_queue
  - start()：类运行统一入口，负责启动进程与 socket 初始化，在创建 DNSRelayServer 对象后，可通过此成员方法启动服务器

- **协议解析工具函数 DNSProtocol.py**
  - getDomainName(data)：解析请求包的查询域名
  - getPacketId(data)：解析 DNS 报文的 ID
  - getPacketIp(data)：解析响应包的 IP
  - createID(addr, mapping)：构造用于请求包的 ID，约束目前有效的 ID 相同
  - createResponsePacket(addr, data, ip)：构造 DNS 响应报文，用于发向客户端，如果发现给定的 ip 是 0.0.0.0，则构造错误码报文进行非法域名拦截
  - createQueryPacket(data, id, dname)：构造 DNS 请求报文，用于发向远程 DNS 服务器

在程序运行期间，主要依赖三大处理进程以及相关辅助函数、共享变量，三大处理进程分别指客户端请求处理分发进程 dns_server_listener、客户端回包进程 dns_server_sender 以及远程 DNS 查询进程 dns_server_helper，三进程架构保证了服务的高并发性能，其中 dns_server_listener 只负责快速收集客户端发送来的查询请求，分类（已有答案直接发回客户端或者需要向远程 DNS 服务器求助）后分发至下游任务，dns_server_sender 专门将发送队列 send_queue 中已有 ip 的相关数据重组为正确的响应报文后,持续发回客户端,dns_server_helper 则不断从请求队列 help_queue 中取出数据向远程 DNS 服务器查询，获得正确的响应后再放入发送队列 send_queue，交由 dns_server_sender 发回客户端；三大进程共享了相关功能函数、全局队列，功能函数诸如请求包/回包构造、DNS 报文解析等都是线程安全函数，十分方便地为进程提供功能支持，全局队列是指发送队列 send_queue 以及请求队列 help_queue，其中 send_queue 存储将要发回客户端的有关数据，help_queue 存储将要发向远程 DNS 服务器的查询报文有关数据，它们通过管道技术实现进程间安全、高效通信，服务功能架构如图 2 所示。

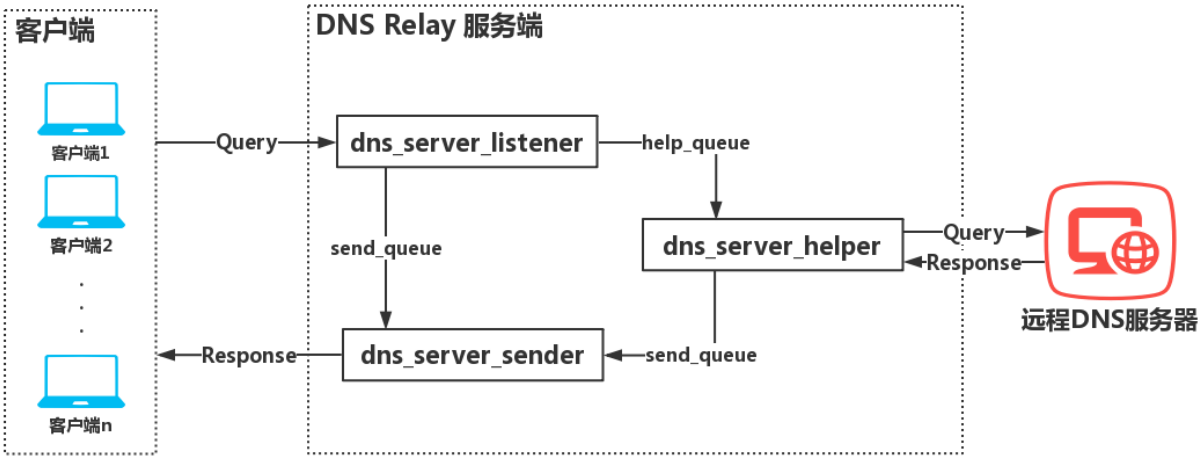

图 2 功能架构图

三、 软件流程图

1. 主流程图

如图 3 所示，服务主流程由读取用户参数→初始化(socket 以及进程)开始，之后三大进程（获取请求进程:dns_server_listener、响应回发进程:dns_server_sender、远程查询进程:dns_server_helper）持续运行直到外部给出中断指令。

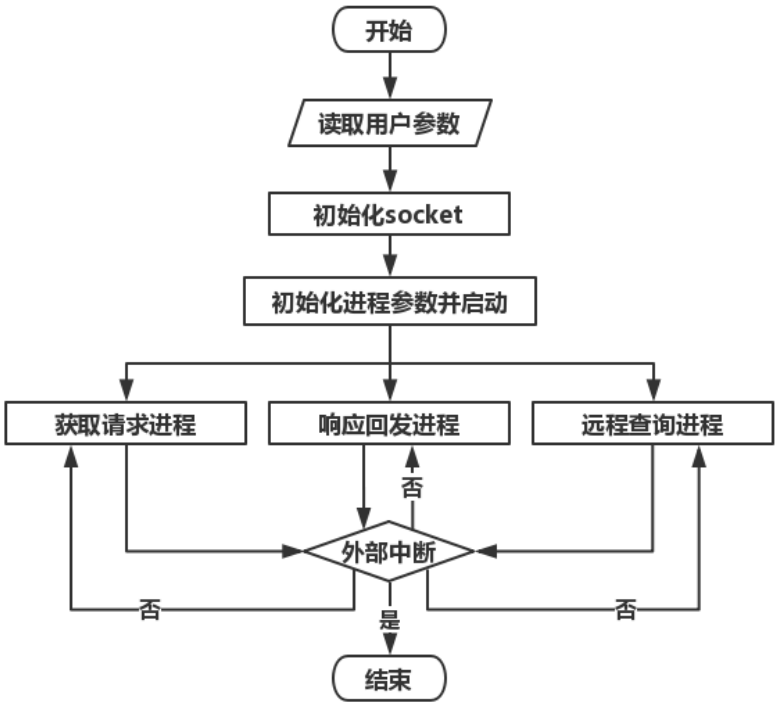

图 3 软件主流程图

2. 子流程图

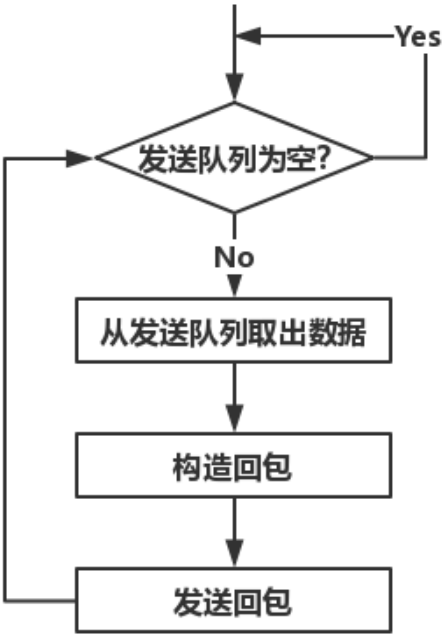

图 4 响应回发子流程

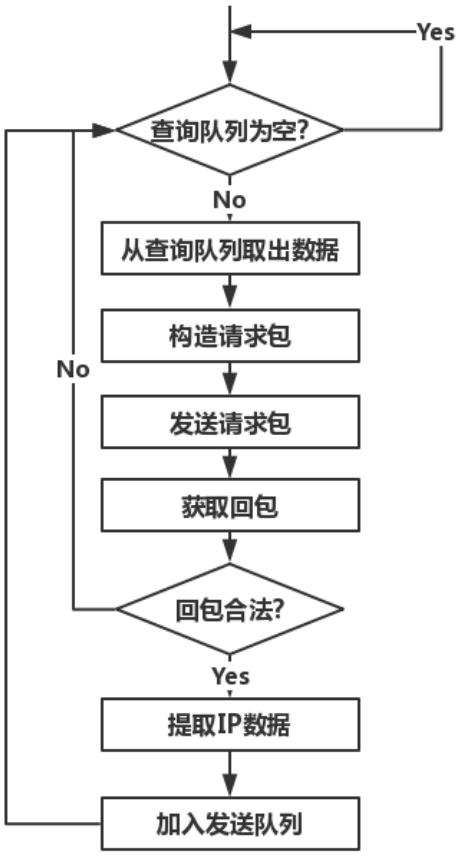

图 5 远程查询子流程

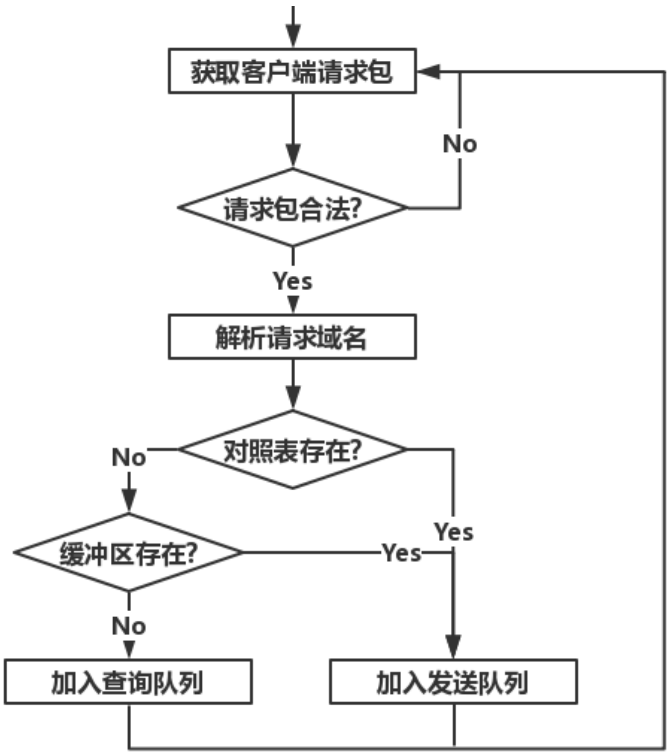

图 6 获取请求子流程

A. 获取请求:dns_server_listener

如图 6 所示，dns_server_listener 不断接收客户端请求包，在校验请求合法后（必须是查询类型、QTYPE 必须是 A 类型）取出其需要查询的域名，之后首先在本地 IP-域名对照表中进行检索，如果域名或者缓冲区存在该域名则将对应 IP 以及请求包、客户端信息写入发送队列 send_queue，准备进行写回，否则将请求包、客户端信息写入请求队列 help_queue，准备进行查询。

B. 响应回发:dns_server_sender

如图 4 所示，dns_server_sender 不断从发送队列中取得数据，根据这些数据（响应 IP、客户端信息）来构造响应包并发回对应客户端。

C. 远程查询:dns_server_helper

如图 5 所示，dns_server_helper 不断从请求队列中取得数据，根据这些数据来构造请求包向远端 DNS 服务器查询（ID 要重新构造且不能和未完成），之后便不断接受远端 DNS 服务器的响应，如果合法则取出对应 IP，与客户端信息一起写入发送队列，准备发回。

四、 软件测试

1. 测试用例设计

本次测试分为功能测试与性能测试，其中功能测试围绕运行场景、用例特殊性展开，两大运行场景分别为单机测试（本地使用）以及联机测试（远程机使用），用例特殊性考虑对照表中已有且正常的域名、对照表中已有但非法的域名以及对照表中不存在的域名，衡量标准分为 nslookup 请求解析正常与否以及浏览器能否正常使用。性能测试主要围绕并发量展开，编写脚本利用多线程的数量来模拟并发水平，观察 DNS 服务器的平均响应时间来评估其并发能力，整理如下:

功能测试	单机测试	表中正常域名
		表中非法域名
		表中不存在的域名
	联机测试	表中不存在的域名
性能测试	不同并发量	随机生成的域名

2. 测试结果分析

A. 功能测试

(1) 单机测试

- 表中正常域名：www.blogger.com
  - 调试信息（RECV 指接收到，SEND 指发回）

```
[RECV 2019-07-14 11:11:08]:  From:('127.0.0.1', 58414), Query:www.blogger.com
[RECV 2019-07-14 11:11:08]:  From:('127.0.0.1', 58907), Query:www.blogger.com
[SEND 2019-07-14 11:11:08]:  To: ('127.0.0.1', 58414), IP: 74.125.207.191, Query: www.blogger.com
[RECV 2019-07-14 11:11:08]:  From:('127.0.0.1', 58414), Query:www.blogger.com
[RECV 2019-07-14 11:11:08]:  From:('127.0.0.1', 58907), Query:www.blogger.com
[SEND 2019-07-14 11:11:08]:  To: ('127.0.0.1', 58414), IP: 74.125.207.191, Query: www.blogger.com
```

- 表中非法域名：008.cn
  - 调试信息

```
[RECV 2019-07-14 11:34:53]:  From:('127.0.0.1', 58626), Query:008.cn
[SEND 2019-07-14 11:34:53]:  To: ('127.0.0.1', 58626), IP: 0.0.0.0, Query: 008.cn
```

  - Nslookup 工具

```
PS C:\Users\45371> nslookup 008.cn
DNS request timed out.
    timeout was 2 seconds.
Server:  UnKnown
Address:  127.0.0.1

DNS request timed out.
    timeout was 2 seconds.
DNS request timed out.
    timeout was 2 seconds.
*** UnKnown can't find 008.cn: Non-existent domain
```

- 表中不存在的域名：www.baidu.com
  - 浏览器测试

  - Nslookup 工具

```
Non-authoritative answer:
DNS request timed out.
    timeout was 2 seconds.
Name:    www.a.shifen.com
Addresses:  39.156.66.14
          39.156.66.18
Aliases:  www.baidu.com
```

- 调试信息（HELP 是指向远程 DNS 服务器查询）

```
[RECV 2019-07-14 11:36:14]:  From:('127.0.0.1', 62919), Query:www.baidu.com
[RECV 2019-07-14 11:36:14]:  From:('127.0.0.1', 57189), Query:www.baidu.com
[HELP 2019-07-14 11:36:14]:  To:('10.3.9.5', 53), Query:www.baidu.com
[SEND 2019-07-14 11:36:14]:  To: ('127.0.0.1', 62919), Query: www.baidu.com
```

(2) 联机测试（本机 ip：10.201.8.84，测试对象：10.28.210.202，宿舍校园网）

- 测试用例：www.bupt.edu.cn
  - 本机日志

```
[RECV 2019-07-14 11:54:46]:  From:('10.28.210.202', 56251), Query:www.bupt.edu.cn
[RECV 2019-07-14 11:54:46]:  From:('10.28.210.202', 49753), Query:nav.smartscreen.microsoft.com
[HELP 2019-07-14 11:54:46]:  To:('10.3.9.5', 53), Query:www.bupt.edu.cn
[HELP 2019-07-14 11:54:46]:  To:('10.3.9.5', 53), Query:nav.smartscreen.microsoft.com
[SEND 2019-07-14 11:54:46]:  To: ('10.28.210.202', 56251), Query: www.bupt.edu.cn
```

  - 测试机表现

- 测试用例：www.qq.com
  - 本机日志

```
[RECV 2019-07-14 11:56:08]:  From:('10.28.210.202', 55224), Query:www.qq.com
[HELP 2019-07-14 11:56:08]:  To:('10.3.9.5', 53), Query:www.qq.com
[SEND 2019-07-14 11:56:08]:  To: ('10.28.210.202', 55224), Query: www.qq.com
```

  - 测试机表现

B. 性能测试

为了模拟并发查询，本次实验专门开发了并发测试程序 test.py，其通过调整多线程值 t 以及每个线程内查询次数 m 实现不同的并发量 m*t。测试程序每次均匀增加并发量，计算该轮次内响应时间（得到回包时间-查询时间）的相关统计指标（均值、分位数等）来评估并发水平。下图是本次性能测试结果，可以发现尽管随着并发量提高，平均耗时也在提高，不过仍在接受范围内（因为测试代码超时时间 30000s，如果平均响应时间到达超时时间说明 DNS 服务器基本无法使用了）。

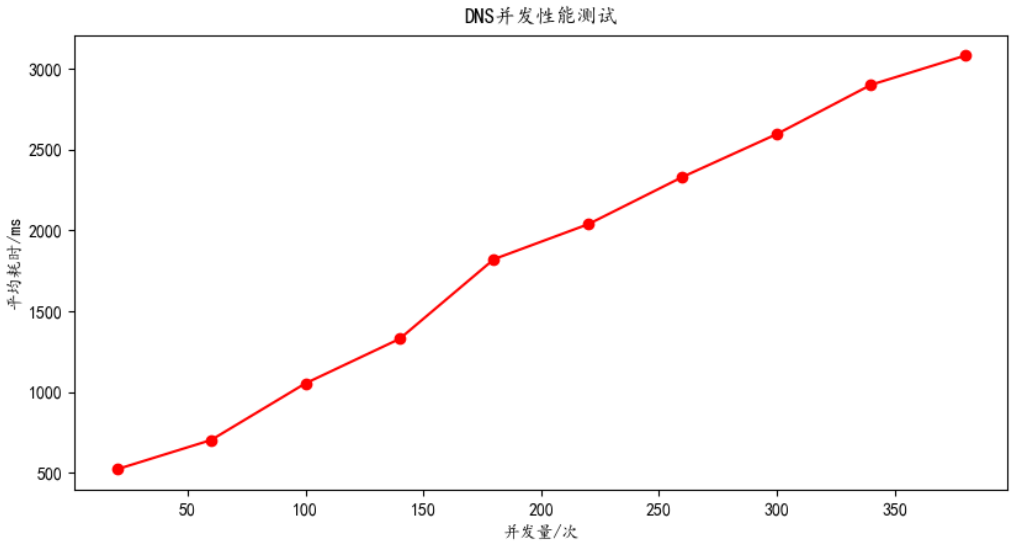

图 7 并发性能测试

五、 问题分析与解决

1．如何实现多客户端并发？如何评测服务器的并发水平？

服务器采用 Python 编写，虽然尝试使用多线程+消息队列+互斥锁的模式，但是效果不明显，这是由于 Python GIL 全局锁的存在，只能使用 cpu 一个核进行运算，同一时间只有一个线程运行，在查询相关资料后，最终采用多进程编程的方式，将发送回包、接收请求、远端查询分别用不同的进程，在不同的 cpu 核里执行，从而实现并发。另外一方面，socket 默认采取同步的方式，recvfrom/sendto 在执行过程中都会阻塞其他线程，十分影响效率，后采取 settimeout 的方式将其转变为非阻塞，超时后程序捕获异常继续执行即可。

如何评测服务器的并发水平呢？由于经济问题，没法寻找大量节点去实际模拟 DNS 请求，后来选择编写脚本使用多线程技术模拟大量并发请求，test.py 文件中实现了增量性能测试，即通过均匀增加线程数去发送 DNS 请求，统计发送接收的平均耗时时间。使用该脚本，我测试了单线程、多线程、多进程三种架构下的服务，耗时时间逐渐下降，并发能力逐渐提升，当然，这只是一个简单的模拟测试，仅供参考。

2．如何解析/封装 DNS 报文?

DNS 报文属于二进制字节流，Python 的 struct 库提供了丰富序列化/反序列化函数，通过 unpack 函数，能够将字节流(Bytes 数组)转换为字符串 str 方便程序逻辑的判断；通过 pack 函数，能够将 str 字符串转换为 Bytes 数组；

有了解析工具库后，我参考了 RFC1034/RFC1035 以及 Wireshark 的实际报文封装了解析/包装函数集，能够方便获取域名/获取 IP/获取 ID 以及包装请求包、包装查询包；

在程序运行期间，遇到了一些问题。

首先是 ID 映射问题，DNS 协议要求请求/查询报文 ID 得一一对应，在实际运行中，发现客户端会从不同端口请求相同的域名，如果单纯的将该请求包转发给远程 DNS，会出现 ID 相同冲突的现象，因此在转发前需要将 ID 和客户端请求存储在哈希表中（ID 作为 key，客户端请求作为 value)，直到确认收到远程 DNS 的回包后再清除该 ID-key。

其次是 IP 解析问题，此前默认 Response 报文后四字节是 IP，这是建立在 Authority/Additional 为空的基础上，还有一个隐患是如果某个域名对应多个 IP，那么这种做法只可以取到一个 IP，后来的做法是只有在 Answer、Authority、Additional 的数量分别是 1、0、0 时用这种方法将 IP 取出并存入缓存，其余情况直接做转发（大多数情况解析 IP 较复杂)。

六、 心得体会

本次 DNS 中继服务器课程设计于我而言，收益颇丰。最大的收获体现在两个方面：软件设计相关技术与计算机网络 DNS 协议。

软件设计方面，首先是编程语言的选择，在 C++与 Python 中，我最终选择了后者。C++的优势是效率高、稳定；Python 的优势是快速开发，可用第三方库多。由于此前熟悉的 C++开发是在 Linux 环境下，多线程可以轻松的使用 pthread 编写，然而本次实验 Linux 不方便 DNS 的调试，不熟悉 Windows 下的多线程编程，故采用 Python。尽管 Python 的 GIL 全局锁在多线程上令人失望，但是 multiprocessing 库提供的多进程编程也令人一喜，一定程度上弥补了并发上的劣势；在技术选型上，为了使课程设计更具意义，我抛弃了 dnspython/dnslib 等已经封装好的 DNS 服务/解析库，转而利用 struct 手动解析/包装报文+socket 实现 UDP 传输通信；软件编写上，我采取了面向对象编程+命令行参数控制的方式，

计算机网络协议方面，尽管开发大多数精力投放在性能提升、并发测试上，但是 DNS 协议的一些琐碎细节带来的问题也不可忽略。学习 DNS 协议，我采用 RFC1035+中文教程的方式熟悉大体报文格式，之后利用 Wireshark 捕获实际 DNS 报文+python 字节流输出的方式深入理解报文字段含义，在此基础上，我将 DNS 解析/包装相关的函数统一整合在 DNSProtocol.py 文件中供其他模块使用。

课程设计相对于基础课程学习具有诸多意义，在原先的计算机网络课程上，我们更多着眼于理论知识的普及，对它们在实践中的应用以及更多的难点及其考虑有所疏漏，经过本次课程设计，我对 DNS 协议的设计原理、使用有了更深的理解。