

南京大学本科生实验报告

课程名称：计算机网络

任课教师：田臣/李文中

助教：

学院	计算机科学与技术	专业（方向）	计算机科学与技术
学号	201220062	姓名	黄子睿
Email	201220062@smail.nju.edu.cn	开始/完成日期	5/21-5/22

1. 实验名称：

Lab7 Content Delivery Network

2. 实验目的：

通过设计建议的 DNS 服务器与 CacheServer 机制实现简单的 CDN 功能，从而加深对网络应用层的了解与掌握。

3. 实验内容：

Task2 DNS server

Step 1 : Load DNS Records Table

通过完善 `DNSServer.parse_dns_file()` 函数实现这一功能。这里要将条目信息存入一个 list 列表中，在匹配时遍历列表。每一条记录的信息可以分成三个部分，分别是用于匹配的正则表达式，记录类型，以及记录值。在文件中，匹配的正则表达式时以字符串形式存放的记录名字 (Record name)，但是考虑到该名字中包含有通配符 '*'，因此考虑采用正则表达式实现，这样在匹配记录项时，通过正则匹配实现即可。

下面给出这一过程的伪代码：

```
1. def DNSServer.parse_dns_file(self, file):
```

```

2.      open file
3.      foreach line in file do:
4.          line.strip('\n')    //去掉每行末尾的换行符
5.          entry := line.split(' ') //按空格分割该行，得到字符串列表
6.
7.          //处理 Record name, 将其转换为正则表达式
8.          Record_reg_name := Turn entry[0] into regular expression
9.          //Record type 即为 entry[1]
10.         Record_type := entry[1]
11.         //Record value 是从 entry[2]开始的字符串列表
12.         Record_value := entry[2: ]
13.
14.         self._dns_table.append(Record_name, Record_type, Record_value
    )

```

正则表达式转换分为三步, 将 Record name 中的 '.' 更换为 '\.'; 将 '*' 更换为 '+';

最后在末尾加上 '\.?' 即可, 如下:

```

1. pattern = dns_entry[0].replace(".", "\.")
2. pattern = pattern.replace("*", "+")
3. pattern += "\.?"

```

Step 2 : Reply Clients' DNS Request

这一功能在函数 `DNSHandler.get_response()` 中实现。首先需要遍历列表匹配对应的记录, 若匹配不到, 则返回 (None, None)。若匹配成功, 考察记录类型。若为 CNAME, 直接返回 (CNAME, Record value) 即可。若为类型为 A, 考察 Record Value (此时应当是一个字符串数组), 如果仅有仅有一条 Record Value, 直接返回 (A, Record value[0]) 即可; 否则根据距离选择最优的记录返回, 即 (A, best record value), 当然如果无法计算距离, 任意返回一个记录值即可。

计算距离的公式是通过地球经纬度坐标计算两点间距离, 公式如下:

$$S = 2 \arcsin \sqrt{\sin^2 \frac{a}{2} + \cos(\text{Lat1}) \times \cos(\text{Lat2}) \times \sin^2 \frac{b}{2}} \times 6378.137$$

其中 a, b 分别为弧度制之下维度差与经度差。

下面给出对应的伪代码：

```
1. def DNSHandler.get_response(self, name):
2.     foreach record in self.dns_table do:
3.         if name matches record.name then:
4.             if record.type == "CNAME" then:
5.                 return ("CNAME", record.value)
6.             else:
7.                 if len(record.value) == 1 then:
8.                     return ("A", record.value[0])
9.                 else:
10.                    return ("A", best record.value by cal_distance)
```

Task3 Caching server

Step 1 : HTTPRequestHandler

这一阶段需要完善三个函数，分别是

```
1. 1. Complete CachingServerHttpHandler.sendHeaders()
2. 2. Complete CachingServerHttpHandler.do_GET()
3. 3. Complete CachingServerHttpHandler.do_HEAD()
```

其中后两者的逻辑几乎相同，而 sendHeaders 的函数则与框架代码中 sendBody 函数相对应，分别发送 HTTP 报文的报文头与报文体。

因此 do_GET 与 do_HEADER 的为代码逻辑是：

```
1. def do_GET():
2.     get headers and body from touchItem()
3.     if headers is None and body is None:
4.         send 404 not found
5.     else:
6.         sendHeaders(headers)
7.         sendBody(body)
8.
9. def do_GET():
10.    get headers and body from touchItem()
11.    if headers is None and body is None:
12.        send 404 not found
13.    else:
14.        sendHeaders(headers)
```

而 sendHeaders 的为代码逻辑则是首先通过 send_response 发送一个 200 的

OK 应答，再将队列 headers 中的每个元素通过 send_header 发送出去，最后发送一个 end_headers。

伪代码如下：

```
1. def sendHeaders(headers):
2.     send_response(HTTPStatus.OK)
3.     foreach hd in headers:
4.         send_header(hd.key, hd.value)
5.     end_headers()
```

Step2: Caching Server

这一步就是实现 touchItem 函数。总的思路是，首先在 Cache 中查询是否需要报文已经缓存并且没有过期，如果是，直接发送回去即可；否则请求主服务器，将记录调入 Cache 并且发送给报文请求方。注意到 requestMainServer 函数返回的是一个 HTTPResponse 类，因此通过其中的 getHeaders 读取报文头，read 读取报文体。

下面是伪代码：

```
1. def touchItem(path):
2.     if path is in cache and it's not expired then:
3.         return (cache[path].headers, cache[path].body)
4.
5.     resp := requestMainServer()
6.
7.     if resp is not None then:
8.         headers := resp.get_headers()
9.         body := resp.read()
10.        cache.append(headers, body)
11.        return (headers, body)
12.
13.    return (None, None) //The path is not in main server
```

Optional Step: Stream Forwarding

通过 readinto 函数与 generator 实现这一功能。主要的逻辑是每读 64byte 的

内容，就将其发送给用户，边读边发，直到全部读取报文体。

为了简化说明，仅讨论 cache miss 的情形。我采取的方法是将 touchItem 函数转换为一个 generator，首先通过 yield headers，再依次 yield 每次读取的 64Byte 内容，最后抛出 stopIteration 异常。

每次读取 64Byte 信息是通过函数 readinto 实现的，通过 yield 将这 64Byte 信息及时返回给 touchItem()，如果此时读取的字节数不足 64Byte，说明已经全部读取报文，抛出 StopIteration 异常已结束循环，表示 generator 已经结束。这一部分逻辑的代码为：

```
1. headers = resp.getheaders()
2. self.cacheTable.setHeaders(path, headers)
3. yield headers
4. buf = bytearray(BUFFER_SIZE)
5. ret = BUFFER_SIZE
6. while ret == BUFFER_SIZE:
7.     ret = resp.readinto(buf)
8.     yield buf
9.     self.cacheTable.appendBody(path, buf)
10. raise StopIteration
```

同时修改 do_GET 与 do_HEADER 函数，这里以前者为例，首先生成 touchItem 的 generator，再通过 next(gen) 读取第一个返回的 headers，然后通过 for 循环将每次返回的 64Byte 信息加到 body 上即可。

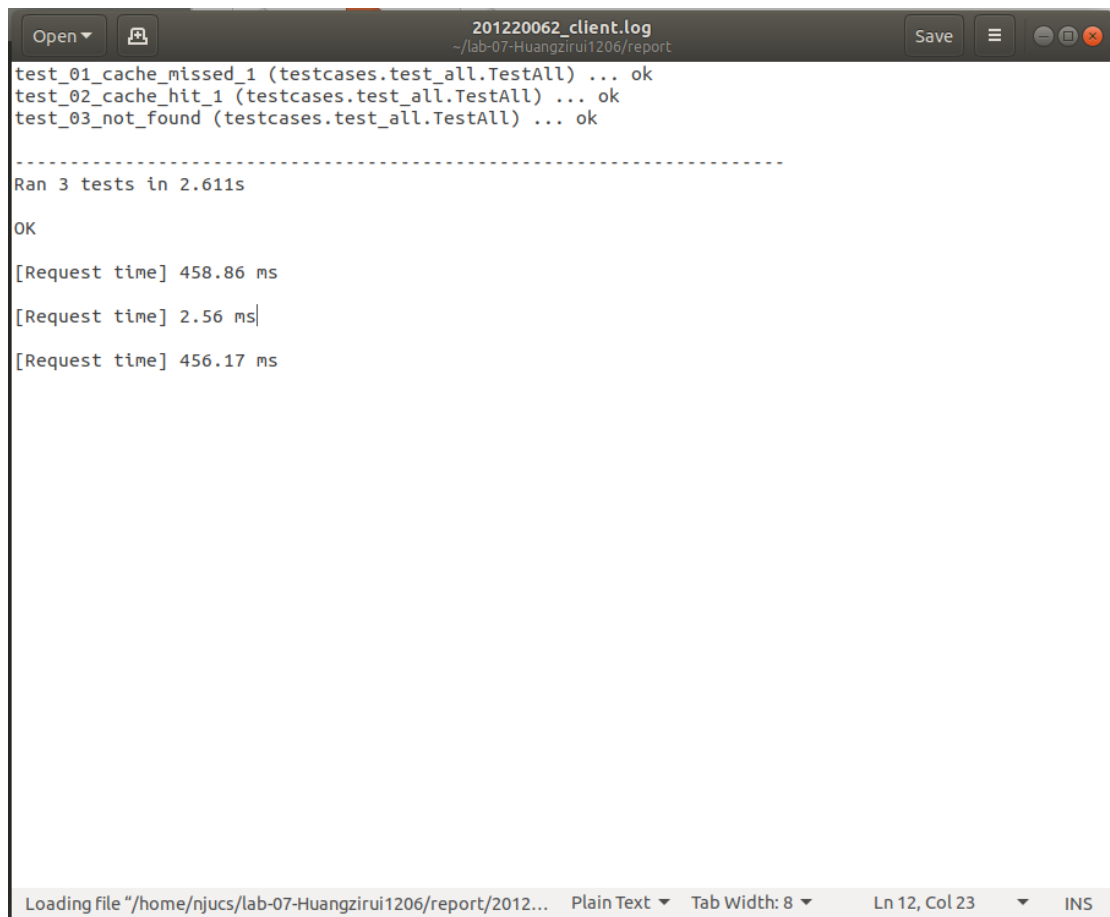
Task3 Caching server

首先贴出运行结果的截图。（见下页）

从下图不难看出，cache 命中与不命中的时间相差很大：hit_cache 时用时仅约 2 毫秒，而 miss_cache 时，用时到达了 458 毫秒，两者相差了 200 倍！如果有一个 HTTP 报文重复申请了很多次，这是 Cache 可以节省大量的时间，除了第一次调入 Cache 的过程，其余情况都可以从 Cache 中直接取报文，大大提高

了网络运行的效率。

下面对这一结果加以分析。考虑 Cache 的相关代码，do_GET 与 do_HEADER 时用户会首先调用 touchItem 向 cache server 申请报文，这时如果报文就在 Cache 中，那么直接返回即可，其用时几乎可以忽略不计；否则 server 需要向主服务器发送申请，主服务器再接受申请后从数据库中调出对应的报文再发送回来，这个报文在 Main server 与 Cache server 之间传递的过程将花费大量的时间，因此体现在结果上就是 cache_hit 时，用时显著缩短了。



The screenshot shows a text editor window titled "201220062_client.log" with the path "~/lab-07-Huangzirui1206/report". The window contains the following text:

```
test_01_cache_missed_1 (testcases.test_all.TestAll) ... ok
test_02_cache_hit_1 (testcases.test_all.TestAll) ... ok
test_03_not_found (testcases.test_all.TestAll) ... ok

-----
Ran 3 tests in 2.611s

OK

[Request time] 458.86 ms
[Request time] 2.56 ms|
[Request time] 456.17 ms
```

The status bar at the bottom indicates the file path, text encoding (Plain Text), tab width (8), and the current cursor position (Ln 12, Col 23).