

**计算机网络**

**课程实验报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 实验名称 | 计算机网络实验1——HTTP代理服务器 | | | | | |
| 姓名 | 许坤彬 | | 院系 | 计算学部 | | |
| 班级 | 2203103 | | 学号 | 2022113586 | | |
| 任课教师 | 聂兰顺 | | 指导教师 | 聂兰顺 | | |
| 实验地点 | G001 | | 实验时间 | 2024/10/10 | | |
| 实验课表现 | 出勤、表现得分(10) |  | 实验报告  得分(40) |  | 实验总分 |  |
| 操作结果得分(50) |  |
| 教师评语 | | | | | | |
|  | | | | | | |

****

|  |
| --- |
| 实验目的： |
| 熟悉并掌握 Socket 网络编程的过程与技术；  深入理解 HTTP 协议，掌握 HTTP 代理服务器的基本工作原理；  掌握 HTTP 代理服务器设计与编程实现的基本技能。 |
| 实验内容： |
| (1) 设计并实现一个基本 HTTP 代理服务器。 要求在指定端口（例如8080） 接收来自客户的 HTTP 请求并且根据其中的 URL 地址访问该地址所指向的 HTTP 服务器（原服务器）， 接收 HTTP 服务器的响应报文，并将响应报文转发给对应的客户进行浏览。  (2) 设计并实现一个支持 Cache 功能的 HTTP 代理服务器。 要求能缓存原服务器响应的对象，并能够通过修改请求报文（添加 if-modified-since头行），向原服务器确认缓存对象是否是最新版本。  (3) 扩展 HTTP 代理服务器，支持如下功能：  a) 网站过滤：允许/不允许访问某些网站；  b) 用户过滤：支持/不支持某些用户访问外部网站；  c) 网站引导：将用户对某个网站的访问引导至一个模拟网站（钓鱼）。 |
| 实验过程： |
| 1. Socket编程客户端和服务器端的主要步骤：  在实验过程中，我选择python作为完成实验的编程语言，调用了python的socket库完成了试验任务。所以下面的内容主要都是基于python的socket库而言的方法。Python的socket  库函数包含服务端函数和客户端函数两种。  使用socket进行编程首先需要使用socket(family, type,[protocol])方法完成套接字的创建，对于服务端而言，需要将创建的套接字通过bind()方法与指定的IP地址和端口号进行绑定，进而通过listen()方法对刚刚绑定的端口进行监听。随后不断通过accept()方法接收客户端请求，accept函数回返回一个新的套接字对象，用来接收、发送数据。使用这个由accept函数返回的套接字调用recv()方法就可以完成客户端请求、数据的接收。  对客户端而言，首先需要使用socket()方法创建套接字，随后调用connect()方法完成客户端程序与目标服务端的连接。随后就可以继续使用send()方法进行数据、请求的发送，使用recv()方法完成数据、请求的接收。  在最后分别使用close()方法关闭套接字。  2. HTTP代理服务器的基本原理：  代理服务器，俗称“翻墙软件”，允许一个网络终端（一般为客户端）通过这个服务与另一个网络终端（一般为服务器）进行非直接的连接代理服务器在指定端口（例如8080）监听浏览器的访问请求（需要在客户端浏览器进行相应的设置），接收到浏览器对远程网站的浏览请求时，代理服务器开始在代理服务器的缓存中检索URL对应的对象（网页、图像等对象），找到对象文件后，提取该对象文件的最新被修改时间；代理服务器程序在客户的请求报文首部插入<If-Modified-Since: 对象文件的最新被修改时间>，并向原Web 服务器转发修改后的请求报文。如果代理服务器没有该对象的缓存，则会直接向原服务器转发请求报文，并将原服务器返回的响应直接转发给客户端，同时将对象缓存到代理服务器中。代理服务器程序会根据缓存的时间、大小和提取记录等对缓存进行清理。  3. HTTP代理服务器的程序流程图    4. 实现HTTP代理服务器的关键技术及解决方案  4.1 基本HTTP代理服务器的实现  根据前面的socket编写服务端和客户端流程，以及HTTP代理服务器的基本原理，我们可以梳理出完成HTTP代理服务器的流程。首先我们需要创建一个socket，作为代理服务器监听本地端口的服务器端socket。使用bind()函数将其与宏变量指定的本地端口进行绑定，使用listen()方法进行监听，并使用使用accept()方法开始接收来自客户端的请求，并返回用于接收请求的client\_socket和对应的客户端地址。接下来为每一个客户端请求创建一个新的线程，调用线程处理函数完成接下来的逻辑。  在handle\_client()线程处理函数中，首先需要从client\_socket接收请求，随后对请求中的方法和目标主机进行提取，如果请求方法是get或者post，那么首先使用urlparse对请求头中的url进行解析，解析得到目标主机host和目标端口port。随后创建一个remote\_socket套接字，并使用connect()用于连接服务端。进而将原本请求通过send()转发给服务器，并调用recv()方法获取服务器的响应。接收到http的响应后通过响应解析分别得到响应状态行，响应头和响应体，用于后面的缓存。接下来通过send()方法将响应内容发送给客户端。完成请求的处理和响应的发送之后，关闭client\_socket和remote\_socket  4.2 实现支持cache的HTTP代理服务器  在完成4.1的基础上进一步完成4.2的需求，首先需要定义一种合适的数据结构用于缓存网页内容。在这里我们定义代理服务器的缓存条目cache\_entry结构分为三部分：头部、缓存内容和上次修改时间last-modified字段。缓存HTTP头部字段又包含method、url、host和port四个字段。缓存cache被声明为一个cache\_entry的列表。缓存的处理流程主要添加在handle\_client()线程处理函数中，在判断出请求方法是get或者post之后，首先通过解析出的请求头部检查缓存中是否有与之相同的缓存HTTP头部，如果有则返回index，否则返回-1。  在命中缓存的情况下，提取缓存中的上次修改时间字段，将其作为if\_modified\_since字段添加到将要发给服务端的请求中。服务端会依据请求中的if\_modified\_since字段判断自这个时间之后请求内容是否被修改，如果修改会返回200并返回新的请求内容，否则返回304且不会返回请求内容。那么我们只需要在接收到服务端的response之后，对状态行中的状态码进行判断，如果是304则只需要将缓存cache\_entry中的缓存内容返回给客户端即可。否则的话则将缓存中的缓存内容进行更新，并直接将服务端返回的新response发送给客户端。  在缓存没有命中的情况下，我们需要依据第一次接收到的response增加缓存信息。在接收到服务端的response之后，依据缓存结构将各项需要的数据进行更新，对于上次修改时间字段，需要直接从response中提取last-modified字段，并添加到缓存中。设定cache\_index字段用于记录需要写入到哪一个cache块，将其设定为全局变量，这样防止不同的线程之间出现冲突。cache\_index初始设定为0，每次完成更新之后加1。在完成更新后再将response发送给客户端。  4.3 扩展功能  4.3.1 网站过滤  设定一个列表invalid\_website作为禁止访问的网站列表，在handle\_client线程处理函数中，在接收到客户端请求并完成url提取后，判断是否出现在禁止列表中，如果出现则直接关闭这个client\_socket而不发送任何东西。    4.3.2 用户过滤  设定一个列表restrict\_host作为禁止访问的用户ip，根据代理服务器socker在监听本地端口时所接收到的addr，判断是否出现在列表中，如果出现则同样直接关闭client\_socket。  4.3.3 钓鱼网站  首先配置钓鱼网站的相关信息，设定源网站和目的网站，在handle\_client线程函数中，如果在接收到客户端请求后发现钓鱼源网站与头部的url相同，则修改头部字段中的host和url，将其修改为目的钓鱼网站中的对应信息，并且将修改结构整合回请求中，并接下来发送给服务端。  5. 带有详细注释的源代码  import socket  import threading  import select  import time  import os  from urllib.parse import urlparse  # 用于解析URL  # 可以用来测试缓存的网站  # http://www.arnc2024.cn/hello.html  # http://example.com/  # 禁止访问的网站列表  invalid\_website = ["http://jwes.hit.edu.cn/"]  # 钓鱼网站配置  fishing\_src = "http://today.hit.edu.cn"  fishing\_dest = "http://example.com/"  fishing\_dest\_host = "example.com"  # 限制访问的用户IP  restrict\_host = ["127.0.0.1"]  # 是否开启钓鱼、禁止和限制访问功能  func\_web\_block = True  func\_user\_block = False  func\_fish = True  # 设置代理服务器监听的端口  PROXY\_PORT = 10240  BUFFER\_SIZE = 4096  # 缓存字典，键值对格式(URL:缓存响应和响应时间)  CACHE\_EXPIRY = 60      # 设置缓存的过期时间，单位为秒  CACHE\_NUM = 50  cache\_index = 0  # 当前应将缓存放在哪个位置  # HTTP 重要头部数据 用来存储解析后的HTTP请求头信息  class HttpHeader:      def \_\_init\_\_(self):          self.method = ''  # GET、POST 或 CONNECT          self.url = ''     # 请求的 URL          self.host = ''    # 目标主机          self.port = 80    # 目标端口  # 缓存的 HTTP 头部（不包含 Cookie） 用来比较是否命中缓存  class CacheHttpHead:      def \_\_init\_\_(self):          self.method = ''          self.url = ''          self.host = ''          self.port = 80    # 代理服务器缓存结构  class CacheEntry:      def \_\_init\_\_(self):          self.httpHead = CacheHttpHead()          self.buffer = b''      # 存储返回内容          self.date = ''         # 缓存内容的最后修改时间    cache = [CacheEntry() for \_ in range(CACHE\_NUM)]  # 缓存列表    def receive\_full\_response(remote\_socket):      response = b''      while True:          part = remote\_socket.recv(BUFFER\_SIZE)          response += part          if not part:              break      return response  def parse\_http\_response(response):      # 解析 HTTP 响应，获取状态行、头部和正文      header\_end = response.find(b'\r\n\r\n')      if header\_end == -1:          return None, None, None      header\_bytes = response[:header\_end]      body = response[header\_end+4:]      header\_text = header\_bytes.decode('utf-8', errors='ignore')      lines = header\_text.split('\r\n')      status\_line = lines[0]      headers = {}      for line in lines[1:]:          if ': ' in line:              key, value = line.split(': ', 1)              headers[key.lower()] = value      return status\_line, headers, body  def http\_equal(http1, http2):      # 判断两个 HTTP 报文是否相同      return (http1.method == http2.method and http1.host == http2.host              and http1.url == http2.url and http1.port == http2.port)    def is\_in\_cache(cache, http\_header):      # 检查缓存中是否存在对应的条目      for index, entry in enumerate(cache):          # print(entry.httpHead.url)          if http\_equal(entry.httpHead, http\_header):              return index      return -1  def modify\_request\_for\_fishing(request, http\_header):      # 修改请求，进行钓鱼网站重定向 返回新的请求url      lines = request.decode('utf-8').split('\r\n')      # 修改请求行和 Host 头部      new\_lines = []      for line in lines:          if line.startswith(http\_header.method):              # 修改请求行              parts = line.split(' ')              if len(parts) >= 3:                  parts[1] = http\_header.url  # 更新 URL                  new\_line = ' '.join(parts)                  new\_lines.append(new\_line)              else:                  new\_lines.append(line)          elif line.startswith('Host:'):              # 修改 Host 头部              new\_lines.append(f"Host: {http\_header.host}")          else:              new\_lines.append(line)      return '\r\n'.join(new\_lines).encode('utf-8')  # 处理客户端请求的线程处理函数  def handle\_client(client\_socket, addr):        global cache\_index        # 用户过滤      if func\_user\_block and (addr[0] in restrict\_host):          print(addr[0] + '已经被禁止访问')          client\_socket.close()   # 关闭socket          return              # 接收客户端请求      # recv(bufsize, [flag])：接受TCP套接字的数据，数据会以字符串的形式返回      # bufsize：是一个整数，表示要接收的最大字节数，接收实际数据量小于等于这个值      # flag(可选)：一般忽略      request = client\_socket.recv(BUFFER\_SIZE)     # decode用于解码        if not request.strip():  # 使用strip()去掉首尾空白字符          print("收到空请求，关闭连接。")          client\_socket.close()          return        # 提取请求中的方法和目标主机      first\_line = request.decode().split('\n')[0]     # 提取请求中的第一行      parts = first\_line.split()        if len(parts) < 3:          print(parts)          print(f"格式错误的请求行: '{first\_line}'")          client\_socket.close()          return        header = HttpHeader()   # 实例化HTTP头对象        header.method = parts[0]      header.url    = parts[1]        # 网站过滤      if func\_web\_block and any(site in header.url for site in invalid\_website):          print("网站已经被屏蔽")          client\_socket.close()          return        # 钓鱼网站      if func\_fish and fishing\_src in header.url:          print(f"从源网站{fishing\_src}转到目的网站{fishing\_dest}")          header.host = fishing\_dest\_host          header.url = fishing\_dest          # 重新将这些信息整合回到请求中          request = modify\_request\_for\_fishing(request, header)          # connect请求的格式为 arnc2024.cn:2024      if header.method == "CONNECT":          # 提取目标主机和端口          header.host = header.url.split(':')[0]          header.port = int(header.url.split(':')[1])          # 尝试连接目标服务器          try:              # 创建用于向目标IP发送请求的socket：使用IPv4和TCP相关参数              remote\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)              # connect()：客户端程序用来连接服务端              remote\_socket.connect((header.host, header.port))                # 向客户端发送 200 Connection Established 响应              client\_socket.send(b"HTTP/1.1 200 Connection Established\r\n\r\n")              # 确立HTTP连接之后开始转发数据              forward\_data(client\_socket, remote\_socket)          except Exception as e:              print(f"请求转发失败：{e}")              client\_socket.close()        # get/post请求格式是http://www.arnc2024.cn/      elif header.method == "GET" or header.method == "POST" :            # 由于格式和connect不一样，所以需要使用 urlparse 来解析 URL，提取目标主机和端口          parsed\_url = urlparse(header.url)          header.host = parsed\_url.hostname          header.port = parsed\_url.port if parsed\_url.port else 80  # 如果URL中没有端口，默认使用80端口，一般情况下后端服务器的服务进程默认开放在80            cache\_entry = None            # for index, entry in enumerate(cache):          #     print(entry.httpHead.url)            # 检查缓存          index = is\_in\_cache(cache, header)            # print(index)          if index > -1:              print(f"命中缓存{index}")              # 检查缓存              cache\_entry = cache[index]  # 初始化 cache\_entry              # 缓存命中，添加 If-Modified-Since 头部              print(cache\_entry.date)              request = add\_if\_modified\_since(request, cache\_entry.date)              print("修改的request:" + request.decode('iso-8859-1', errors='ignore'))              print("已修改request，增加了if-modified-since")              # print(request)          # if cache\_entry is None:  # 如果缓存没有命中                # 尝试连接目标服务器          try:              # 创建用于向目标IP发送请求的socket：使用IPv4和TCP相关参数              remote\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)              # print("socket连接")                # connect()：客户端程序用来连接服务端              remote\_socket.connect((header.host, header.port))              # print("connect")              # 在Get/Post请求中，直接将原本的请求转发就可以              remote\_socket.send(request)              # print("转发请求")                # 获取响应              response = remote\_socket.recv(BUFFER\_SIZE)              # response = receive\_full\_response(remote\_socket)                # content\_length = get\_content\_length(response)                #if content\_length and content\_length != len(response):                    # print(f"Warning: Content-Length ({content\_length}) does not match actual response length ({len(response)})")                    # print("获得响应")              # print(response.decode())                status\_line, headers, body = parse\_http\_response(response)                if not status\_line:                  # 无法解析响应，直接转发                  client\_socket.sendall(response)                  return                # 当index>-1，缓存命中              if index > -1 :                  if "304" in status\_line:                      # 没有在服务器被修改，所以可以直接返回                      print(status\_line)                      client\_socket.send(cache\_entry.buffer)                      print(f"缓存没有被修改，所以可以直接返回内容：{header.url}")                  else:                      # 更新缓存                      print("缓存被修改了")                      print(status\_line)                      cache\_entry.buffer = response                      # 修改最新的last-modified                      cache\_entry.date = headers.get('last-modified', '')                      client\_socket.send(response)              else:                  print("缓存没命中，增加信息")                  # 缓存没命中，选择index指向的cache进行更新                  print("写入的"+ str(cache\_index))                  cache\_entry = cache[cache\_index % CACHE\_NUM]                    # 修改各项记录                  cache\_entry.httpHead.method = header.method                  cache\_entry.httpHead.url = header.url                  cache\_entry.httpHead.host = header.host                  cache\_entry.httpHead.port = header.port                  cache\_entry.buffer = response                  print(headers)                  # print(response)                    # 获取 Last-Modified 时间                  cache\_entry.date = headers.get('last-modified', '')                  print(cache\_entry.date)                  # cache\_entry.date = time.strftime("%a, %d %b %Y %H:%M:%S GMT", time.gmtime())                    # 更新索引                  cache\_index += 1                    client\_socket.send(response)            except Exception as e:              # print(cache)              print(f"请求转发失败1：{e}")              client\_socket.close()            finally:              client\_socket.close()              remote\_socket.close()    # 转发数据  def forward\_data(client\_socket, remote\_socket):      sockets = [client\_socket, remote\_socket]      while True:          read\_sockets, \_, error\_sockets = select.select(sockets, [], sockets)            if error\_sockets:              break          for sock in read\_sockets:              data = sock.recv(4096)              if not data:                  break              if sock == client\_socket:                  remote\_socket.send(data)              else:                  client\_socket.send(data)      # 关闭套接字      client\_socket.close()      remote\_socket.close()    # 启动代理服务器，作为服务端不断监听本地的请求  def start\_proxy():        # 创建监听套接字      # socket(family, type,[protocol])      # family参数：      #   AF\_UNIX：使用单一的Unix系统进程间通信      #   AF\_INET；Ipv4地址族，服务器之间网络通信      #   AF\_INET6：IPv6      # type参数：      #   SOCK\_STREAM：流式socket，使用TCP时选择此参数      #   SOCK\_DGRAM：数据报式socket，使用UDP的时候选择此参数      #   SOCK\_RAW：原始套接字，允许底层协议如IP、ICMP进行直接访问      # protocol：    可选项，指明接受的协议类型，通常为0或者不填      #   IPPROTO\_RAW：相当于protocol=255，此时socket只能用来发送IP包，而不能接收任何的数据。发送的数据需要自己填充IP包头，并且自己计算校验和      #   IPPROTO\_IP：相当于protocol=0，此时用于接收任何的IP数据包。其中的校验和和协议分析由程序自己完成。      proxy\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)        # 服务端函数        # bind()：将创建的套接字和指定的IP地址和端口进行绑定，使用(IP, 端口)的元组来指定      proxy\_socket.bind(('0.0.0.0', PROXY\_PORT))        # listen()：在使用TCP的服务端开启监听模式      proxy\_socket.listen(5)      # print(f"[\*] 代理服务器正在运行，监听端口 {PROXY\_PORT}")      # 不断接受客户端请求并创建新线程处理      while True:          # 调用accept函数接收客户端请求          # 返回的两个参数为(conn, address)          #   conn为新的套接字对象，用来接受收和发送数据          #   address是连接的客户端的地址          client\_socket, addr = proxy\_socket.accept()              # print(f"[\*] 接受到来自 {addr} 的连接")          # 为每个客户端请求创建一个新线程          # threading.Thread声明了一个线程对象          # threading.Thread(group=None, target=None, name=None, args=(), kwargs={}, \*, daemon=None)          #   group：应该设为None，即不用设置，使用默认值就好，因为这个参数是为了以后实现ThreadGroup类而保留的。          #   target：在run方法中调用的可调用对象，即需要开启线程的可调用对象，比如函数或方法。          #   name：线程名称，默认为“Thread-N”形式的名称，N为较小的十进制数。          #   args：在参数target中传入的可调用对象的参数元组，默认为空元组()。          #   kwargs：在参数target中传入的可调用对象的关键字参数字典，默认为空字典{}。          #   daemon：默认为None，即继承当前调用者线程（即开启线程的线程，一般就是主线程）的守护模式属性，如果不为None，则无论该线程是否为守护模式，都会被设置为“守护模式”。          client\_handler = threading.Thread(target=handle\_client, args=(client\_socket,addr), daemon=True)          # 开启线程活动          client\_handler.start()  # 缓存相关  # 添加If-Modified-Since头  def add\_if\_modified\_since(request, last\_modified):      header\_end = request.find(b"\r\n\r\n")  # 找到头部结束的位置      if header\_end != -1:          headers = request[:header\_end].decode('iso-8859-1').split('\r\n')          body = request[header\_end:]            # 移除现有的 If-Modified-Since 头部（如果存在）          headers = [line for line in headers if not line.lower().startswith('if-modified-since:')]            # 添加新的 If-Modified-Since 头部          headers.append(f"If-Modified-Since: {last\_modified}")            # 重新组合头部          modified\_header = '\r\n'.join(headers).encode('iso-8859-1')            # 返回新的请求（头部 + 空行 + 原请求的主体）          return modified\_header + body      return request  # 如果没有找到头部结束位置，返回原始请求  def get\_last\_modified(response):      # 从响应头中提取Last-Modified字段      headers = response.decode('utf-8').split("\r\n")      for header in headers:          if "Last-Modified" in header:              return header.split(": ", 1)[1]      return None  # 在main函数的情况下，开启代理  if \_\_name\_\_ == "\_\_main\_\_":      start\_proxy() |
| 实验结果： |
| 3.2 用户屏蔽  将127.0.0.1加入到限制列表中，运行程序，发现无法访问任何互联网资源。      3.3 钓鱼网站  设定源网站为 <http://today.hit.edu.cn> ，目的网站为 <http://example.com/> 。 |
| 问题讨论： |
| 为什么程序无法处理https开头的相关网站？  经过请求的打印，https网站在访问时主要使用了connect请求，而非实验内容要求的get或者是post请求。所以如果想要让程序成为真正的代理服务器，我们需要能够处理connect请求。connect请求的url格式与get/post不太一样，所以不适用urlparse进行url解析，而是可以直接依据‘:’的位置分割除目标主机的host和端口号，进而使用socket进行连接，连接完成后向客户端发送一个response，告诉客户端完成了连接，response内容为HTTP/1.1 200 Connection Established\r\n\r\n。接下来再进行数据的传输，利用select模块完成正确socket的识别（client或者是remote），随后将数据发送给对应的的相对模块。  对应的handle\_client函数修改部分   # connect请求的格式为 arnc2024.cn:2024      if header.method == "CONNECT":          # 提取目标主机和端口          header.host = header.url.split(':')[0]          header.port = int(header.url.split(':')[1])          # 尝试连接目标服务器          try:              # 创建用于向目标IP发送请求的socket：使用IPv4和TCP相关参数              remote\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)              # connect()：客户端程序用来连接服务端              remote\_socket.connect((header.host, header.port))                # 向客户端发送 200 Connection Established 响应              client\_socket.send(b"HTTP/1.1 200 Connection Established\r\n\r\n")              # 确立HTTP连接之后开始转发数据              forward\_data(client\_socket, remote\_socket)          except Exception as e:              print(f"请求转发失败：{e}")              client\_socket.close()  对应的forward\_data函数：  def forward\_data(client\_socket, remote\_socket):      sockets = [client\_socket, remote\_socket]      while True:          read\_sockets, \_, error\_sockets = select.select(sockets, [], sockets)            if error\_sockets:              break          for sock in read\_sockets:              data = sock.recv(4096)              if not data:                  break              if sock == client\_socket:                  remote\_socket.send(data)              else:                  client\_socket.send(data)      # 关闭套接字      client\_socket.close()      remote\_socket.close() |
| 心得体会： |
| 对使用python进行socket编程有了一定的认识和了解，对HTTP代理服务器有了一定的了解，对于HTTP请求、响应的格式、结构和过程有了体验，对一些网站引导、网站屏蔽和用户屏蔽有了认识，对计算机网络有了更多的兴趣。 |