

**计算机网络**

**课程实验报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 实验名称 | HTTP代理服务器的设计与实现 | | | | | |
| 姓名 | 张志强 | | 院系 | 计算机科学与技术/计算机工程 | | |
| 班级 | 1903101 | | 学号 | 1190201717 | | |
| 任课教师 | 刘亚维 | | 指导教师 | 刘亚维 | | |
| 实验地点 | 格物207 | | 实验时间 | 2021.10.31 | | |
| 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。（选作内容，加分项目，可以当堂完成或课下完成）  (3) 扩展 HTTP 代理服务器，支持如下功能：（选作内容，加分项目，可以当堂完成或课下完成）  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： |
| **HTTP网络应用通信原理**  在HTTP网络应用中，通信的两个进程主要采用客户端/服务器模式（或浏览器/服务器模式），客户端向服务器发送请求，服务器接收到客户端请求后，向客户端提供相应的服务。通信过程如下：    服务器端：   1. 服务器端需要首先启动，并绑定一个本地主机端口，在端口上提供服务 2. 等待客户端请求 3. 接收到客户端请求时，建立起与客户端通信的套接字，开启新线程，将与客户端通信的套接字放入新线程处理 4. 返回第二步，主线程继续等待客户端请求。 5. 关闭服务器   客户端：   1. 根据服务器IP与端口，建立起与服务器通信的socket 2. 向服务器发送请求报文，并等待服务器应答 3. 请求结束后关闭socket   **HTTP代理服务器原理**  RFC 7230规定，代理在HTTP通信中扮演一个中间人的角色，对于连接来的客户端来说，它扮演一个服务器的角色；对于要连接的远程服务器，它扮演一个客户端的角色。代理服务器就负责在客户端和服务器之间转发报文。如下图所示：  图片包含 屏幕截图  描述已自动生成  代理服务器在指定端口监听浏览器的请求，在接收到浏览器的请求时，首先查看浏览器的IP地址，如果来自被限制的IP地址，就向客户端返回错误信息。否则，从请求头中解析出请求的host主机，如果属于不允许访问的主机，则向客户端返回错误信息，如果属于需要引导的host，则修改请求头内所有的地址字段为被引导的地址。之后，根据URL查找是否缓存中是否有该URL的缓存，如果存在，则从中取出Last-modified头部内容，并构造包含If-modified-since的请求头，向服务器发送确认最新版本的报文，并在返回的请求头第一行里确认是否有“Not Modified”，如果存在该字段，则说明本地缓存未过期，直接将本地缓存内容发送给客户端，否则缓存过期，将服务器的报文直接写回客户端。如果缓存中不存在，就直接将客户端请求转发到服务器，并将服务器返回内容缓存后，再返回给客户端。  代理服务器的拦截用户、拦截主机和钓鱼信息都预先配置在配置文件里，并在程序运行后读入程序中，以按照规则执行。  程序运行流程图如下：    **代码实现：**  （1）代理服务器启动并监听客户端连接：  public void start() throws IOException {  // 启动服务端  ServerSocket serverSocket = new ServerSocket(port);  startLog();  // 遇到客户端连接就创建一个任务，然后提交到线程池当中，接下来由该线程与客户端保持通信  while (true) {  // 监听客户端连接  Socket socket = serverSocket.accept();  System.*out*.println("接收到了"+ socket.getInetAddress() + " " + socket.getPort() + "的连接");  // 创建一个任务并提交给线程池处理  threadPool.execute(new ProxyTask(socket));  } }  （2）线程池：  static final class MixedTargetThreadPool {  // 首先从环境变量 mixed.thread.amount 中获取预先配置的线程数  // 如果没有对 mixed.thread.amount 做配置，则使用常量 MIXED\_MAX 作为线程数  private static final int *max* = (null != System.*getProperty*(*MIXED\_THREAD\_AMOUNT*)) ?  Integer.*parseInt*(System.*getProperty*(*MIXED\_THREAD\_AMOUNT*)) : *MIXED\_MAX*;  // 自定义线程池  private static final ThreadPoolExecutor *EXECUTOR* = new ThreadPoolExecutor(  *MIXED\_CORE*,  *max*,  *KEEP\_ALIVE\_SECONDS*,  TimeUnit.*SECONDS*,  new LinkedBlockingQueue(*QUEUE\_SIZE*),  new ThreadPoolExecutor.CallerRunsPolicy()  );  static {  *EXECUTOR*.allowCoreThreadTimeOut(true);  // 钩子函数，用来关闭线程池  Runtime.*getRuntime*().addShutdownHook(new Thread(new Runnable() {  @Override  public void run() {  *shutdownThreadPoolGracefully*(*EXECUTOR*);  }  }));  } }  （3）基于LRU的缓存：  public class LRUCache implements Cache {  */\*\*  \* 默认的代理服务器要缓存的请求的最大容量  \*/* private static final int *MAX\_CAPACITY* = 1024;   // 读写锁  private final ReentrantReadWriteLock lock = new ReentrantReadWriteLock();   // 读锁  private final WriteLock writeLock = lock.writeLock();   // 写锁  private final ReadLock readLock = lock.readLock();   private final LRU<String, byte[]> lru;    public LRUCache() {  lru = new LRU<>(*MAX\_CAPACITY*);  }   public LRUCache(int maxCapacity) {  lru = new LRU<>(maxCapacity);  }   @Override  public void addCache(String url, byte[] content) {  writeLock.lock();  try {  lru.put(url, content);  } finally {  writeLock.unlock();  }  }   @Override  public byte[] getContent(String url) {  readLock.lock();  try {  return lru.get(url);  } finally {  readLock.unlock();  }  }    */\*\*  \* 具体的LRU数据结构 <br/>  \* 不是线程安全的，需要自行解决线程安全问题  \*/* static class LRU<K, V> extends LinkedHashMap<K, V> {  // 最大的容量  private final int maxCapacity;   public LRU(int maxCapacity) {  // accessOrder参数为true时，当调用get和put方法时会将访问到的元素放到双向链表的尾部  super(16, 0.75f, true);  this.maxCapacity = maxCapacity;  }   // 实现LRU的关键方法，如果map里面的元素个数大于了缓存最大容量，则返回true，然后会删除链表的顶端元素eldest  @Override  public boolean removeEldestEntry(Map.Entry<K, V> eldest){  return size() > maxCapacity;  }  } }  （4）加载配置文件：  protected void initConfig() {  InputStream inputStream = null;  BufferedReader urlReader = null;  BufferedReader userReader = null;  BufferedReader fishingReader = null;   try {  // 读取主配置文件proxy.properties  inputStream = this.getClass().getClassLoader().getResourceAsStream("proxy.properties");  Properties properties = new Properties();  properties.load(inputStream);   // 加载主配置  SingletonFactory factory = SingletonFactory.*getInstance*();  ProxyConfig config = factory.getObject(ProxyConfig.class);  config.setUrlRule(Integer.*parseInt*(properties.getProperty("urlRule")));  config.setUserRule(Integer.*parseInt*(properties.getProperty("userRule")));   // 设置配置的url，文件里一行就是一个url  InputStream urlStream = this.getClass().getClassLoader().getResourceAsStream("url.txt");  if (urlStream != null) {  urlReader = new BufferedReader(new InputStreamReader(urlStream));  List<String> urls = new ArrayList<>();  String line;  while ((line = urlReader.readLine()) != null) {  urls.add(line);  }  config.setUrls(urls);  }   // 设置配置的User即主机地址，一行就是一个地址  InputStream userStream = this.getClass().getClassLoader().getResourceAsStream("user.txt");  if (userStream != null) {  userReader = new BufferedReader(new InputStreamReader(userStream));  List<String> users = new ArrayList<>();  String line;  while ((line = userReader.readLine()) != null) {  users.add(line);  }  config.setUsers(users);  }   // 设置要被钓鱼的用户，一行就是就是一个用户即主机地址  InputStream fishingStream = this.getClass().getClassLoader().getResourceAsStream("fishing.txt");  if (fishingStream != null) {  fishingReader = new BufferedReader(new InputStreamReader(fishingStream));  List<String> fishingUsers = new ArrayList<>();  String line;  while ((line = fishingReader.readLine()) != null) {  fishingUsers.add(line);  }  config.setFishingUsers(fishingUsers);  }   } catch (IOException e) {  e.printStackTrace();  System.*out*.println("配置文件不存在或格式不正确");  } finally {  // 关闭资源  if (inputStream != null) {  try {  inputStream.close();  } catch (IOException e) {  e.printStackTrace();  }  }  if (urlReader != null) {  try {  urlReader.close();  } catch (IOException e) {  e.printStackTrace();  }  }  if (userReader != null) {  try {  userReader.close();  } catch (IOException e) {  e.printStackTrace();  }  }  if (fishingReader != null) {  try {  fishingReader.close();  } catch (IOException e) {  e.printStackTrace();  }  }  } }  （5）提交给线程池的任务：  public void run() {  InputStream clientInputStream;  OutputStream clientOutputStream;  String url = null;  String host = null;  int port = 80;  StringBuffer buffer = new StringBuffer(); // HTTP请求头的字符形式   try {  clientInputStream = socket.getInputStream();  clientOutputStream = socket.getOutputStream();   // 解析HTTP请求头  String line;  while ((line = IOUtil.*readHttpLine*(clientInputStream)) != null) {  if (line.startsWith("GET")) {  // GET /index.html HTTP1.1  url = line.split(" ")[1];  } else if (line.startsWith("Host")) {  // Host: 127.0.0.1:80  host = line.split(" ")[1];  }  buffer.append(line).append("\r\n");  }  buffer.append("\r\n");   if (host == null) {  // *TODO：处理没有带host字段的请求* return;  }   // 解析地址和端口号，如果没有端口号则使用默认的80  String[] split = host.split(":");  host = split[0];  if (split.length != 1) {  port = Integer.*parseInt*(split[1]);  }   // 网站过滤  if (config.getUrlRule() == ProxyConstants.*ALLOW\_URL*) {  // 如果当前访问的网站没在配置文件中则拦截  if (!config.getUrls().contains(host)) {  clientOutputStream.write(refuseProxy().getBytes());  clientOutputStream.flush();  return;  }  } else if (config.getUrlRule() == ProxyConstants.*REFUSE\_URL*) {  // 如果要访问的网站存在于配置文件中则拦截  if (config.getUrls().contains(host)) {  clientOutputStream.write(refuseProxy().getBytes());  clientOutputStream.flush();  return;  }  } else { // 配置文件写错了  clientOutputStream.write(refuseProxy().getBytes());  clientOutputStream.flush();  return;  }   // 用户过滤  String clientHost = socket.getInetAddress().getHostAddress();  if (config.getUserRule() == ProxyConstants.*ALLOW\_USER*) {  // 如果客户端的Host不在配置文件里拦截  if (config.getUsers().contains(clientHost)) {  clientOutputStream.write(refuseProxy().getBytes());  clientOutputStream.flush();  return;  }  } else if (config.getUserRule() == ProxyConstants.*REFUSE\_USER*) {  // 如果客户端的Host在配置文件里拦截  if (config.getUsers().contains(clientHost)) {  clientOutputStream.write(refuseProxy().getBytes());  clientOutputStream.flush();  return;  }  } else { // 配置文件写错了  clientOutputStream.write(refuseProxy().getBytes());  clientOutputStream.flush();  return;  }   // 钓鱼  if (config.getFishingUsers().contains(clientHost)) {  // 构造发送给钓鱼网站的HTTP报文  StringBuffer fishingHTTP = new StringBuffer();  fishingHTTP.append("GET " + ProxyConstants.*fishingUrl* + " HTTP/1.1" + "\r\n");  fishingHTTP.append("Host: " + ProxyConstants.*fishingHost* + "\r\n");  fishingHTTP.append("\r\n");  String fishingHTTPStr = fishingHTTP.toString();   // 建立连接然后发送数据  String[] hostAndPort = ProxyConstants.*fishingHost*.split(":");  targetSocket = new Socket(hostAndPort[0], Integer.*parseInt*(hostAndPort[1]));  OutputStream outputStream = targetSocket.getOutputStream();  outputStream.write(fishingHTTPStr.getBytes());  waitTargetServerAndTransfer(clientOutputStream, targetSocket.getInputStream());  return;  }   // 对于非GET请求的方法，直接转发给目的服务器  if (url == null) {  transfer(host, port, buffer, clientInputStream, clientOutputStream);  return;  }   String uri = url;  byte[] content = cache.getContent(uri);  // 对于GET方法，如果缓存中存在则向目的服务器发送条件GET  if (content != null) {  // 从缓存中提取Last-Modified值  String lastModified = parseLastModified(content);  // 构造条件GET请求  StringBuffer ifGetReqBuffer = new StringBuffer();  ifGetReqBuffer.append("GET " + url + " HTTP/1.1\r\n");  ifGetReqBuffer.append("Host: " + host + ":" + port + "\r\n");  ifGetReqBuffer.append("If-modified-since: " + lastModified + "\r\n");  ifGetReqBuffer.append("\r\n");  String ifGetReq = ifGetReqBuffer.toString();  // 向目的服务器发送  targetSocket = new Socket(host, port);  OutputStream outputStream = targetSocket.getOutputStream();  InputStream inputStream = targetSocket.getInputStream();  outputStream.write(ifGetReq.getBytes());  outputStream.flush();  // 阻塞式监听目的服务器的返回值  String respFirstLine = IOUtil.*readHttpLine*(inputStream);  int code = Integer.*parseInt*(respFirstLine.split(" ")[1]);  // 缓存过期  if (code != 304) {  System.*out*.println("代理服务器对" + uri + "的缓存过期");  // 将报文转发至客户端  byte[] firstLine = (respFirstLine + "\r\n").getBytes();  clientOutputStream.write(firstLine);  byte[] resp = waitTargetServerAndTransfer(clientOutputStream, inputStream);  // 将响应结果进行缓存，只缓存具有Last-Modified首部行的响应结果  if (new String(resp).contains("Last-Modified")) {  cache.addCache(uri, ArrayUtils.*addAll*(firstLine, resp));  System.*out*.println("对" + uri + "的响应结果进行了缓存");  }  } else { // 缓存命中  System.*out*.println("对" + uri + "的访问命中缓存");  // 直接返回缓存中的值  clientOutputStream.write(content);  clientOutputStream.flush();  }  } else { // 缓存不存在，则直接请求目的服务器，然后转发给客户端，并在代理服务器进行缓存  System.*out*.println("代理服务器没有对" + uri + "请求的缓存");  byte[] resp = transfer(host, port, buffer, null, clientOutputStream);  // 将响应结果进行缓存，只缓存具有Last-Modified首部行的响应结果  if (new String(resp).contains("Last-Modified")) {  cache.addCache(uri, resp);  System.*out*.println("对" + uri + "的响应结果进行了缓存");  }  }  } catch (Exception e) {  e.printStackTrace();  } finally {  try {  if (socket != null) {  socket.close();  }  if (targetSocket != null) {  targetSocket.close();  }  } catch (IOException e) {  e.printStackTrace();  }  } } |
| 实验结果： |
| （1）基础转发功能：  请求个人服务器：8.140.166.139：9999/hh    （2）缓存：  上次请求时代理服务器打印日志如下：    （3）网站过滤：      （4）用户过滤：      （5）钓鱼： |
| 问题讨论： |
| 无 |
| 心得体会： |
| 本次实验实现了一个简易的代理服务器，加深了我们对socket编程的理解，让我们了解到了“神秘”的翻墙软件的实现，受益匪浅。 |