# 概述

[NanoHttpd](https://github.com/NanoHttpd/nanohttpd), 纳米级的HTTP Daemon,是一个用Java写的HTTP服务器，堪称蝇级，但麻雀虽小，五脏俱全，在GitHub上有着两千多个Star

## 特点

• 只有一个Java文件，提供HTTP 1.1支持。

• 没有固定的配置文件，日志记录，授权等等（如果你需要它们，自己实现。但是错误传递给java.util.logging）。

• 支持HTTPS（SSL）。

• 对cookie的基本支持。

• 支持GET和POST方法的参数解析。

• 一些内置的HEAD，POST和DELETE请求支持。不过，您可以轻松实现/定制任何HTTP方法。

• 支持文件上传。为小型上传使用内存，为大型文件使用临时文件。

• 从不缓存任何东西。

• 默认情况下，不会限制带宽，请求时间或同时连接。

• 所有标题名称都被转换为小写字母，因此它们在浏览器/客户端之间不会改变。

• 持久连接（连接“保持活动”）支持允许通过单个套接字连接提供多个请求。

## Socket通信承载功能

Asdf

在线程或后台任务(非UI线程)中，实例化socket并设置监听端口；

开启 while (mIsRunning) {…}死循环体，监听请求

循环体内：

2.1 Socket socket = mServerSocket.accept();响应一次请求

2.2 mRequestHandler.handle(socket) 处理本次请求

2.2.1 获取请求体中的Path

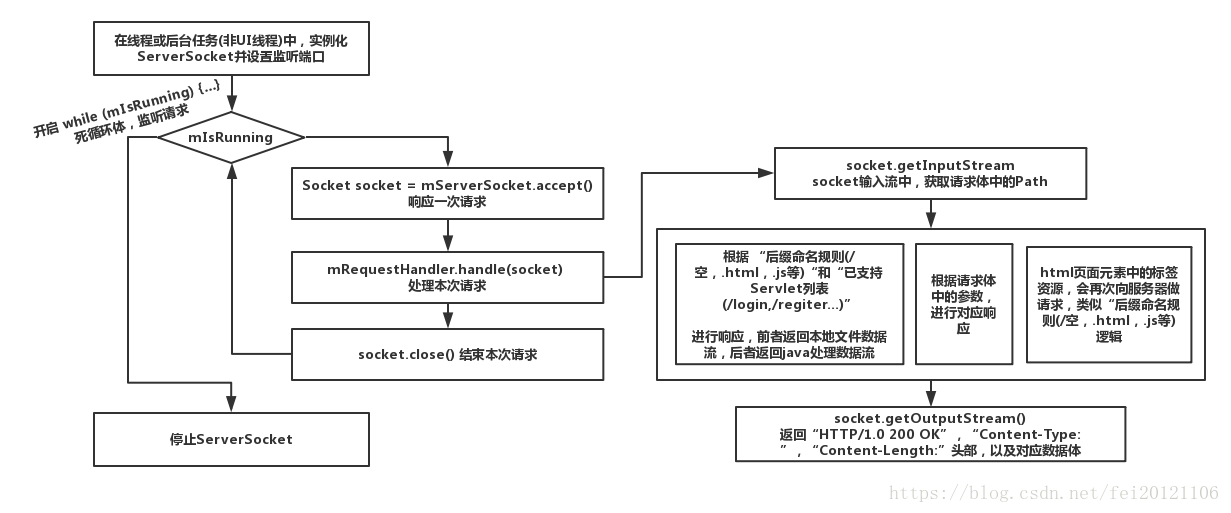
2.2.2 根据 “后缀命名规则(/空，.html，.js等)“和“已支持Servlet列表(/login,/regiter…)”进行响应，前者返回本地文件数据流，后者返回java处理数据流

2.2.2 根据请求体中的参数，进行对应响应

2.2.2 html页面元素中的标签资源，会再次向服务器做请求，类似“后缀命名规则(/空，.html，.js等)逻辑

2.2.3 返回“HTTP/1.0 200 OK”，“Content-Type: ”，“Content-Length:”头部，以及对应数据体

2.3 socket.close() 结束本次请求



NanoHTTPD项目目前由四部分组成：

/core - 由一（1）个Java文件组成的全功能的HTTP（s）服务器，可随时为您自己的项目定制/继承。

/samples - 关于如何定制NanoHTTPD的简单例子。看到HelloServer.java是一款热情招呼你的杀手级应用！

/nanolets - 独立的nano应用服务器，在/core的基础上，为实现者提供一个类似于系统的servlet。 响应URL返回JAVA运算数据

/webserver - 独立的文件服务器。运行和享受。一种流行的用途似乎是从Android设备上提供文件。 响应URL返回本地资源文件数据

/websocket - Websocket的实现，也在一个Java文件中，依赖于core。

/fileupload - 整合了apache常用文件上传库

# 核心实现NanoHTTPD

Fg

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **final** String hostname;
2. **public** **final** **int** myPort;//端口
3. **private** **volatile** ServerSocket myServerSocket;//socketServer
4. **private** IFactoryThrowing<ServerSocket, IOException> serverSocketFactory = **new** DefaultServerSocketFactory();//socketServer工厂
5. **private** IHandler<IHTTPSession, Response> httpHandler;//请求处理器
6. **protected** List<IHandler<IHTTPSession, Response>> interceptors = **new** ArrayList<IHandler<IHTTPSession, Response>>(4);//拦截器
8. **protected** IAsyncRunner asyncRunner;//异步任务运行器
9. **private** IFactory<ITempFileManager> tempFileManagerFactory;//临时文件操作管理器的生成工厂
11. **public** NanoHTTPD(**int** port) {
12. **this**(**null**, port);
13. }
14. **public** NanoHTTPD(String hostname, **int** port) {
15. **this**.hostname = hostname;
16. **this**.myPort = port;
17. setTempFileManagerFactory(**new** DefaultTempFileManagerFactory());
18. setAsyncRunner(**new** DefaultAsyncRunner());
20. // creates a default handler that redirects to deprecated serve();
21. **this**.httpHandler = **new** IHandler<IHTTPSession, Response>() {
23. @Override
24. **public** Response handle(IHTTPSession input) {
25. **return** NanoHTTPD.**this**.serve(input);
26. }
27. };
28. }

## DefaultServerSocketFactory 服务SocketServer工厂

Sdfas

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. /\*\*
2. \* SocketServer工厂，用于生产ServerSocket
3. \*/
4. **public** **class** DefaultServerSocketFactory **implements** IFactoryThrowing<ServerSocket, IOException> {
6. @Override
7. **public** ServerSocket create() **throws** IOException {
8. **return** **new** ServerSocket();
9. }
10. }
12. /\*\*
13. \* 工厂能力抽象： 一个“可以在生产实体期间抛出异常”的生成函数
14. \*/
15. **public** **interface** IFactoryThrowing<T, E **extends** Throwable> {
17. T create() **throws** E;
18. }

nanohttpd下的sockets下还有一个SecureServerSocketFactory

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. /\*\*
2. \* Creates a new SSLServerSocket
3. \*/
4. **public** **class** SecureServerSocketFactory **implements** IFactoryThrowing<ServerSocket, IOException> {
5. **private** SSLServerSocketFactory sslServerSocketFactory;
6. **private** String[] sslProtocols;
7. **public** SecureServerSocketFactory(SSLServerSocketFactory sslServerSocketFactory, String[] sslProtocols) {
8. **this**.sslServerSocketFactory = sslServerSocketFactory;
9. **this**.sslProtocols = sslProtocols;
10. }
12. @Override
13. **public** ServerSocket create() **throws** IOException {
14. SSLServerSocket ss = **null**;
15. ss = (SSLServerSocket) **this**.sslServerSocketFactory.createServerSocket();
16. **if** (**this**.sslProtocols != **null**) {
17. ss.setEnabledProtocols(**this**.sslProtocols);
18. } **else** {
19. ss.setEnabledProtocols(ss.getSupportedProtocols());
20. }
21. ss.setUseClientMode(**false**);
22. ss.setWantClientAuth(**false**);
23. ss.setNeedClientAuth(**false**);
24. **return** ss;
25. }
26. }

一般不调用的

**public void** makeSecure(SSLServerSocketFactory sslServerSocketFactory, String[] sslProtocols) {  
 **this**.**serverSocketFactory** = **new** SecureServerSocketFactory(sslServerSocketFactory, sslProtocols);  
}

## DefaultTempFileManagerFactory 临时文件的操作管理器的生成工厂

nanohttpd下的tempfiles全部

临时文件操作管理器的生成工厂

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. /\*\*
2. \* 创建和清除临时文件的默认策略生成器
3. \*/
4. **public** **class** DefaultTempFileManagerFactory **implements** IFactory<ITempFileManager> {
5. @Override
6. **public** ITempFileManager create() {
7. **return** **new** DefaultTempFileManager();
8. }
9. }
10. /\*\*
11. \* 工厂能力抽象
12. \*/
13. **public** **interface** IFactory<T> {
14. T create();
15. }

默认的临时文件操作管理器

1. **public** **interface** ITempFileManager {
2. **void** clear();
3. **public** ITempFile createTempFile(String filename\_hint) **throws** Exception;
4. }
6. /\*\*
7. \* 创建和清除临时文件的默认策略
8. \* -  文件被存储在默认路径 java.io.tmpdir所指向位置
9. \* - 临时文件的创建
10. \* - 提供清除功能
11. \*/
12. **public** **class** DefaultTempFileManager **implements** ITempFileManager {
13. **private** **final** File tmpdir;
14. **private** **final** List<ITempFile> tempFiles;
15. **public** DefaultTempFileManager() {
16. **this**.tmpdir = **new** File(System.getProperty("java.io.tmpdir"));
17. **if** (!tmpdir.exists()) {
18. tmpdir.mkdirs();
19. }
20. **this**.tempFiles = **new** ArrayList<ITempFile>();
21. }
23. @Override
24. **public** **void** clear() {
25. **for** (ITempFile file : **this**.tempFiles) {
26. **try** {
27. file.delete();
28. } **catch** (Exception ignored) {
29. NanoHTTPD.LOG.log(Level.WARNING, "could not delete file ", ignored);
30. }
31. }
32. **this**.tempFiles.clear();
33. }
35. @Override
36. **public** ITempFile createTempFile(String filename\_hint) **throws** Exception {
37. DefaultTempFile tempFile = **new** DefaultTempFile(**this**.tmpdir);
38. **this**.tempFiles.add(tempFile);
39. **return** tempFile;
40. }
41. }

H‘

临时文件：

’ **[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **interface** ITempFile {
2. **public** **void** delete() **throws** Exception;
3. **public** String getName();
4. **public** OutputStream open() **throws** Exception;
5. }
7. **public** **class** DefaultTempFile **implements** ITempFile {
9. **private** **final** File file;
10. **private** **final** OutputStream fstream;
11. **public** DefaultTempFile(File tempdir) **throws** IOException {
12. **this**.file = File.createTempFile("NanoHTTPD-", "", tempdir);
13. **this**.fstream = **new** FileOutputStream(**this**.file);
14. }
16. @Override
17. **public** **void** delete() **throws** Exception {
18. NanoHTTPD.safeClose(**this**.fstream);
19. **if** (!**this**.file.delete()) {
20. **throw** **new** Exception("could not delete temporary file: " + **this**.file.getAbsolutePath());
21. }
22. }
24. @Override
25. **public** String getName() {
26. **return** **this**.file.getAbsolutePath();
27. }
29. @Override
30. **public** OutputStream open() **throws** Exception {
31. **return** **this**.fstream;
32. }
33. }

## DefaultAsyncRunner 异步任务运行器

默认的，当有新的Request来的时候，创新一个新的线程来响应； 这个线程被设置为守护线程和对应的命名

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. /\*\*
2. \* 为requests的响应过程提供异步过程
3. \*/
4. **public** **interface** IAsyncRunner {
5. **void** closeAll();
6. **void** closed(ClientHandler clientHandler);
7. **void** exec(ClientHandler code);
8. }
10. **public** **class** DefaultAsyncRunner **implements** IAsyncRunner {
12. **protected** **long** requestCount;
13. **private** **final** List<ClientHandler> running = Collections.synchronizedList(**new** ArrayList<ClientHandler>());
15. /\*\*
16. \* @return a list with currently running clients.
17. \*/
18. **public** List<ClientHandler> getRunning() {
19. **return** running;
20. }
22. @Override
23. **public** **void** closeAll() {
24. // copy of the list for concurrency
25. **for** (ClientHandler clientHandler : **new** ArrayList<ClientHandler>(**this**.running)) {
26. clientHandler.close();
27. }
28. }
30. @Override
31. **public** **void** closed(ClientHandler clientHandler) {
32. **this**.running.remove(clientHandler);
33. }
35. @Override
36. **public** **void** exec(ClientHandler clientHandler) {
37. ++**this**.requestCount;
38. **this**.running.add(clientHandler);
39. createThread(clientHandler).start();
40. }
42. **protected** Thread createThread(ClientHandler clientHandler) {
43. Thread t = **new** Thread(clientHandler);
44. t.setDaemon(**true**);
45. t.setName("NanoHttpd Request Processor (#" + **this**.requestCount + ")");
46. **return** t;
47. }
48. }

## IHandler响应器

我们在使用NanoHTTPD时候会重写覆盖Response serve(IHTTPSession session)方法，用于真正的处理Request

这里给一个默认的实现，返回一个 NotFound异常Reponse

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. // creates a default handler that redirects to deprecated serve();
2. **this**.httpHandler = **new** IHandler<IHTTPSession, Response>() {
3. @Override
4. **public** Response handle(IHTTPSession input) {
5. **return** NanoHTTPD.**this**.serve(input);
6. }
7. };
9. @Deprecated
10. **protected** Response serve(IHTTPSession session) {
11. **return** Response.newFixedLengthResponse(Status.NOT\_FOUND, NanoHTTPD.MIME\_PLAINTEXT, "Not Found");
12. }

## 核心start函数

### 【步骤1】使用服务SocketServer工厂生成一个 ServerSocket

### 【步骤2】开启一个线程，并将当前NanoHTTPD传递过去

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **void** start(**final** **int** timeout) **throws** IOException {
2. start(timeout, **true**);
3. }
5. **public** **void** start(**final** **int** timeout, **boolean** daemon) **throws** IOException {
6. //【步骤1】使用服务SocketServer工厂生成一个 ServerSocket
7. **this**.myServerSocket = **this**.getServerSocketFactory().create();
8. **this**.myServerSocket.setReuseAddress(**true**);
10. //【步骤2】开启一个线程，并将当前NanoHTTPD传递过去
11. ServerRunnable serverRunnable = createServerRunnable(timeout);
12. **this**.myThread = **new** Thread(serverRunnable);
13. **this**.myThread.setDaemon(daemon);
14. **this**.myThread.setName("NanoHttpd Main Listener");
15. **this**.myThread.start();
16. //绑定端口失败，譬如端口被占用
17. **while** (!serverRunnable.hasBinded() && serverRunnable.getBindException() == **null**) {
18. **try** {
19. Thread.sleep(10L);
20. } **catch** (Throwable e) {
21. // on android this may not be allowed, that's why we
22. // catch throwable the wait should be very short because we are
23. // just waiting for the bind of the socket
24. }
25. }
26. **if** (serverRunnable.getBindException() != **null**) {
27. **throw** serverRunnable.getBindException();
28. }
29. }
31. **protected** ServerRunnable createServerRunnable(**final** **int** timeout) {
32. **return** **new** ServerRunnable(**this**, timeout);
33. }

### 【步骤3】异步线程的执行

【步骤3.1】绑定端口和地址，重置绑定标示和绑定异常

【步骤3.2】开启死循环，监听request: ServerSocket.accept()

【步骤3.3】根据请求soket及其输入流构建处理器ClientHandler

【步骤3.4】ClientHandler传入异步任务运行器中执行

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. @Override
2. **public** **void** run() {
3. **try** {
4. httpd.getMyServerSocket().bind(httpd.hostname != **null** ? **new** InetSocketAddress(httpd.hostname, httpd.myPort) : **new** InetSocketAddress(httpd.myPort));
5. hasBinded = **true**;
6. } **catch** (IOException e) {
7. **this**.bindException = e;
8. **return**;
9. }
10. **do** {
11. **try** {
12. **final** Socket finalAccept = httpd.getMyServerSocket().accept();
13. **if** (**this**.timeout > 0) {
14. finalAccept.setSoTimeout(**this**.timeout);
15. }
16. **final** InputStream inputStream = finalAccept.getInputStream();
17. httpd.asyncRunner.exec(httpd.createClientHandler(finalAccept, inputStream));
18. } **catch** (IOException e) {
19. NanoHTTPD.LOG.log(Level.FINE, "Communication with the client broken", e);
20. }
21. } **while** (!httpd.getMyServerSocket().isClosed());
22. }

### 【步骤4】分发请求

【步骤4.1】获取socket的输出流

【步骤4.2】获取 临时文件管理器

【步骤4.3】NanoHTTPD、临时文件管理器ITempFileManager、输入流、输出流、ip端口，共同构建HttpSession

【步骤4.3】socket未关闭时，死循环执行 session.execute();

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **class** ClientHandler **implements** Runnable {
2. **private** **final** NanoHTTPD httpd;
3. **private** **final** InputStream inputStream;
4. **private** **final** Socket acceptSocket;
5. **public** ClientHandler(NanoHTTPD httpd, InputStream inputStream, Socket acceptSocket) {
6. **this**.httpd = httpd;
7. **this**.inputStream = inputStream;
8. **this**.acceptSocket = acceptSocket;
9. }
11. **public** **void** close() {
12. NanoHTTPD.safeClose(**this**.inputStream);
13. NanoHTTPD.safeClose(**this**.acceptSocket);
14. }
16. @Override
17. **public** **void** run() {
18. OutputStream outputStream = **null**;
19. **try** {
20. outputStream = **this**.acceptSocket.getOutputStream();
21. ITempFileManager tempFileManager = httpd.getTempFileManagerFactory().create();
22. HTTPSession session = **new** HTTPSession(httpd, tempFileManager, **this**.inputStream, outputStream, **this**.acceptSocket.getInetAddress());
23. **while** (!**this**.acceptSocket.isClosed()) {
24. session.execute();
25. }
26. } **catch** (Exception e) {
27. // When the socket is closed by the client,
28. // we throw our own SocketException
29. // to break the "keep alive" loop above. If
30. // the exception was anything other
31. // than the expected SocketException OR a
32. // SocketTimeoutException, print the
33. // stacktrace
34. **if** (!(e **instanceof** SocketException && "NanoHttpd Shutdown".equals(e.getMessage())) && !(e **instanceof** SocketTimeoutException)) {
35. NanoHTTPD.LOG.log(Level.SEVERE, "Communication with the client broken, or an bug in the handler code", e);
36. }
37. } **finally** {
38. NanoHTTPD.safeClose(outputStream);
39. NanoHTTPD.safeClose(**this**.inputStream);
40. NanoHTTPD.safeClose(**this**.acceptSocket);
41. httpd.asyncRunner.closed(**this**);
42. }
43. }
44. }

## HttpSession

HttpSession是java里Session概念的实现，简单来说一个Session就是一次httpClient->httpServer的连接，当连接close后session就结束了，如果没结束则session会一直存在。这点从这里的代码也能看到：如果socket不close或者exec没有抛出异常（异常有可能是client段断开连接）session会一直执行exec方法

一个HttpSession中存储了一次网络连接中server应该保存的信息，比如：URI，METHOD，PARAMS，HEADERS，COOKIES等

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. **public** **interface** IHTTPSession {
2. **void** execute() **throws** IOException;
3. **void** parseBody(Map<String, String> files) **throws** IOException, ResponseException;
4. CookieHandler getCookies();
5. Map<String, String> getHeaders();
6. InputStream getInputStream();
7. Method getMethod();
8. Map<String, List<String>> getParameters();
9. String getQueryParameterString();
10. String getUri();
11. String getRemoteIpAddress();
12. String getRemoteHostName();
13. }

HTTPSession

*//in usb0,this is a bug,timeout 20s!!  
// this.remoteIp = inetAddress.isLoopbackAddress() || inetAddress.isAnyLocalAddress() ? "127.0.0.1" : inetAddress.getHostAddress().toString();  
// this.remoteHostname = inetAddress.isLoopbackAddress() || inetAddress.isAnyLocalAddress() ? "localhost" : inetAddress.getHostName().toString();*

我们重点看下execute函数：

其实就是：解析Respons中的各类参数赋值到HttpSession中，把当前HttpSession传给自己复写的处理函数作为形参,调用并返回Response

【步骤1】读取socket数据流的前8192个字节，因为http协议中头部最长为8192

【步骤2】通过findHeaderEnd函数找到header数据的截止位置，并把位置保存到splitbyte内。

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. **byte**[] buf = **new** **byte**[BUFSIZE];
2. splitbyte = 0;
3. rlen = 0;
4. {
5. **int** read = -1;
6. **try** {
7. read = inputStream.read(buf, 0, BUFSIZE);
8. } **catch** (Exception e) {
9. safeClose(inputStream);
10. safeClose(outputStream);
11. **throw** **new** SocketException("NanoHttpd Shutdown");
12. }
13. **if** (read == -1) {
14. // socket was been closed
15. safeClose(inputStream);
16. safeClose(outputStream);
17. **throw** **new** SocketException("NanoHttpd Shutdown");
18. }
19. **while** (read > 0) {
20. rlen += read;
21. splitbyte = findHeaderEnd(buf, rlen);
22. **if** (splitbyte > 0)
23. **break**;
24. read = inputStream.read(buf, rlen, BUFSIZE - rlen);
25. }
26. }
28. //涉及函数
29. **private** **int** findHeaderEnd(**final** **byte**[] buf, **int** rlen) {
30. //Http协议规定header和body之间使用两个回车换行分割
31. **int** splitbyte = 0;
32. **while** (splitbyte + 3 < rlen) {
33. **if** (buf[splitbyte] == '\r' && buf[splitbyte + 1] == '\n' && buf[splitbyte + 2] == '\r' && buf[splitbyte + 3] == '\n') {
34. **return** splitbyte + 4;
35. }
36. splitbyte++;
37. }
38. **return** 0;
39. }

步骤3】使用unread函数将之前读出来的body pushback回去，这里使用了pushbackstream，用法比较巧妙，因为一旦读到了header的尾部就需要进入下面的逻辑来判断是否需要再读下去了，而不应该一直读，读到没有数据为止

步骤4】decodeHeader，将byte的header转换为java对象

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. **if** (splitbyte < rlen) {
2. inputStream.unread(buf, splitbyte, rlen - splitbyte);
3. }
5. parms = **new** HashMap<String, String>();
6. **if**(**null** == headers) {
7. headers = **new** HashMap<String, String>();
8. }
10. // Create a BufferedReader for parsing the header.
11. BufferedReader hin = **new** BufferedReader(**new** InputStreamReader(**new** ByteArrayInputStream(buf, 0, rlen)));
13. // Decode the header into parms and header java properties
14. Map<String, String> pre = **new** HashMap<String, String>();
15. decodeHeader(hin, pre, parms, headers);

18. //涉及函数
19. //1.Http协议第一行是Method URI HTTP\_VERSION
20. //2.后面每行都是KEY:VALUE格式的header
21. //3.uri需要经过URIDecode处理后才能使用
22. //4.uri中如果包含？则表示有param，httprequest的param一般表现为:/index.jsp?username=xiaoming&id=2
23. **private** **void** decodeHeader(BufferedReader in, Map<String, String> pre, Map<String, String> parms, Map<String, String> headers)
24. **throws** ResponseException {
25. **try** {
26. // Read the request line
27. String inLine = in.readLine();
28. **if** (inLine == **null**) {
29. **return**;
30. }
32. StringTokenizer st = **new** StringTokenizer(inLine);
33. **if** (!st.hasMoreTokens()) {
34. **throw** **new** ResponseException(Response.Status.BAD\_REQUEST, "BAD REQUEST: Syntax error. Usage: GET /example/file.html");
35. }
37. pre.put("method", st.nextToken());
39. **if** (!st.hasMoreTokens()) {
40. **throw** **new** ResponseException(Response.Status.BAD\_REQUEST, "BAD REQUEST: Missing URI. Usage: GET /example/file.html");
41. }
43. String uri = st.nextToken();
45. // Decode parameters from the URI
46. **int** qmi = uri.indexOf('?');
47. **if** (qmi >= 0) {
48. decodeParms(uri.substring(qmi + 1), parms);
49. uri = decodePercent(uri.substring(0, qmi));
50. } **else** {
51. uri = decodePercent(uri);
52. }
54. // If there's another token, it's protocol version,
55. // followed by HTTP headers. Ignore version but parse headers.
56. // NOTE: this now forces header names lowercase since they are
57. // case insensitive and vary by client.
58. **if** (st.hasMoreTokens()) {
59. String line = in.readLine();
60. **while** (line != **null** && line.trim().length() > 0) {
61. **int** p = line.indexOf(':');
62. **if** (p >= 0)
63. headers.put(line.substring(0, p).trim().toLowerCase(Locale.US), line.substring(p + 1).trim());
64. line = in.readLine();
65. }
66. }
68. pre.put("uri", uri);
69. } **catch** (IOException ioe) {
70. **throw** **new** ResponseException(Response.Status.INTERNAL\_ERROR, "SERVER INTERNAL ERROR: IOException: " + ioe.getMessage(), ioe);
71. }
72. }

* 【步骤5】处理cookie

this.cookies = new CookieHandler(this.headers);

* 【步骤6】**响应处理，并生成Respons**
  + 过滤器调用
  + 自己重写的处理函数响应

r = httpd.handle(this);

//涉及NanoHTTPD的函数

public Response handle(IHTTPSession session) {

for (IHandler<IHTTPSession, Response> interceptor : interceptors) {

Response response = interceptor.handle(session);

if (response != null)

return response;

}

return httpHandler.handle(session);

}

//涉及NanoHTTPD的变量

this.httpHandler = new IHandler<IHTTPSession, Response>() {

@Override

public Response handle(IHTTPSession input) {

return NanoHTTPD.this.serve(input);

}

};

//涉及NanoHTTPD的函数

@Deprecated

protected Response serve(IHTTPSession session) {

return Response.newFixedLengthResponse(Status.NOT\_FOUND, NanoHTTPD.MIME\_PLAINTEXT, "Not Found");

}

* 【步骤7】Response返回，将返回数据写入输出流中

r.send(this.outputStream);

## Response响应

发送response的步骤如下： 1.设置mimeType和Time等内容。

2.创建一个PrintWriter，按照HTTP协议依次开始写入内容

3.第一行是HTTP的返回码

4.然后是content-Type

5.然后是Date时间

6.之后是其他的HTTP Header

7.设置Keep-Alive的Header，Keep-Alive是Http1.1的新特性，作用是让客户端和服务器端之间保持一个长链接。

8.如果客户端指定了ChunkedEncoding则分块发送response，Chunked Encoding是Http1.1的又一新特性。一般在response的body比较大的时候使用，server端会首先发送response的HEADER，然后分块发送response的body，每个分块都由chunk length\r\n和chunk data\r\n组成，最后由一个0\r\n结束。

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79346153) [copy](https://blog.csdn.net/qq1028850792/article/details/79346153)

1. /\*\*
2. \* Sends given response to the socket.
3. \*/
4. **protected** **void** send(OutputStream outputStream) {
5. String mime = mimeType;
6. SimpleDateFormat gmtFrmt = **new** SimpleDateFormat("E, d MMM yyyy HH:mm:ss 'GMT'", Locale.US);
7. gmtFrmt.setTimeZone(TimeZone.getTimeZone("GMT"));
9. **try** {
10. **if** (status == **null**) {
11. **throw** **new** Error("sendResponse(): Status can't be null.");
12. }
13. PrintWriter pw = **new** PrintWriter(outputStream);
14. pw.print("HTTP/1.1 " + status.getDescription() + " \r\n");
16. **if** (mime != **null**) {
17. pw.print("Content-Type: " + mime + "\r\n");
18. }
20. **if** (header == **null** || header.get("Date") == **null**) {
21. pw.print("Date: " + gmtFrmt.format(**new** Date()) + "\r\n");
22. }
24. **if** (header != **null**) {
25. **for** (String key : header.keySet()) {
26. String value = header.get(key);
27. pw.print(key + ": " + value + "\r\n");
28. }
29. }
31. sendConnectionHeaderIfNotAlreadyPresent(pw, header);
33. **if** (requestMethod != Method.HEAD && chunkedTransfer) {
34. sendAsChunked(outputStream, pw);
35. } **else** {
36. **int** pending = data != **null** ? data.available() : 0;
37. sendContentLengthHeaderIfNotAlreadyPresent(pw, header, pending);
38. pw.print("\r\n");
39. pw.flush();
40. sendAsFixedLength(outputStream, pending);
41. }
42. outputStream.flush();
43. safeClose(data);
44. } **catch** (IOException ioe) {
45. // Couldn't write? No can do.
46. }
47. }

Sd 9.如果没指定ChunkedEncoding则需要指定Content-Length来让客户端指定response的body的size，然后再一直写body直到写完为止

1. **private** **void** sendAsChunked(OutputStream outputStream, PrintWriter pw) **throws** IOException {
2. pw.print("Transfer-Encoding: chunked\r\n");
3. pw.print("\r\n");
4. pw.flush();
5. **int** BUFFER\_SIZE = 16 \* 1024;
6. **byte**[] CRLF = "\r\n".getBytes();
7. **byte**[] buff = **new** **byte**[BUFFER\_SIZE];
8. **int** read;
9. **while** ((read = data.read(buff)) > 0) {
10. outputStream.write(String.format("%x\r\n", read).getBytes());
11. outputStream.write(buff, 0, read);
12. outputStream.write(CRLF);
13. }
14. outputStream.write(String.format("0\r\n\r\n").getBytes());
15. }
17. **private** **void** sendAsFixedLength(OutputStream outputStream, **int** pending) **throws** IOException {
18. **if** (requestMethod != Method.HEAD && data != **null**) {
19. **int** BUFFER\_SIZE = 16 \* 1024;
20. **byte**[] buff = **new** **byte**[BUFFER\_SIZE];
21. **while** (pending > 0) {
22. **int** read = data.read(buff, 0, ((pending > BUFFER\_SIZE) ? BUFFER\_SIZE : pending));
23. **if** (read <= 0) {
24. **break**;
25. }
26. outputStream.write(buff, 0, read);
27. pending -= read;
28. }
29. }
30. }

<http://www.itboth.com/d/UbMFj2/android-nanohttpd-http>

<http://xuyushi.github.io/2017/05/04/NanoHttpd/>

http://ericcenblog.com/2017/05/16/nanohttpdyuan-ma-pou-xi/

# REF

180427: [使用手机搭建移动服务器：NanoHttpd手机服务器源码分析](https://juejin.im/entry/5ae1bfccf265da0b8d41a299)

[NanoHTTPD源码剖析](http://ericcenblog.com/2017/05/16/nanohttpdyuan-ma-pou-xi/)