§ 1.1　设计实验：简单的 Web 服务器及浏览器

1.1.1　实验要求

1）构建简单的 Web 服务器及客户端，客户端以命令行界面与用户进行交互；

2）Web服务程序在端口80创建一个Socket，等待接入客户程序发来的请求。无论是合法的还是不合法的请求，客户程序都应获得遵循RFC规范的响应。

1.1.2　实验步骤

(1)　构建一个命令行界面的简单 HTTP/1.0 客户程序

当Web客户程序启动后，它应具备以下功能：

1）从命令行参数中获取Web服务器的主机名（例如www.sysu.edu.cn）；

2）建立一个到该服务器80端口的TCP连接；

3）反馈给终端用户一个成功建立连接的提示（如果建立连接失败，亦应有相应的提示）；

4）接受终端用户在命令行的输入（通常是一个HTTP/1.0 的GET 请求，也可能是一些错误的命令）；

5）将用户的输入发送给Web服务器；

6）在接收Web服务器处理请求的响应后，向用户展示该响应的消息头部，并将响应的消息体（body）保存到一个文件中。

完成本步骤后，比较客户程序所获取的响应与本章 § 3.3节中使用ieHTTPHeaders工具手工获取的响应，这些响应的消息头部和消息体应该是相同的。

(2)　构建一个简单的 HTTP/1.0 服务程序

只要求服务程序成功地响应客户程序发来的GET命令，对其他命令可作为不识别的命令来处理。在任何情况下，服务程序都不应崩溃或中止。Web服务程序的根目录（root）应指向一个由命令行参数指定的目录。例如，如果启动服务程序时以命令行参数d:\www指定根目录，则服务程序响应请求GET /test/index.html的语义是将文件d:\www\test\index.html发送给客户程序，如果该文件不存在则回复一个出错消息。

如果GET命令提供的参数是一个目录而不是一个文件（即以“/”结尾），则视同获取一个该目录下的索引文件index.html（或index.htm）。例如，服务程序响应请求GET/的语义是将根目录下的文件index.html（或index.htm）发送给客户程序。

在成功返回的响应消息中，Web服务程序应至少为消息头部的Content-Type域指定两种文件的MIME 类型。本实验假设服务程序根据文件扩展名推断一个文件的MIME类型。例如，以.htm或.html为扩展名的文件其MIME类型被假设为text/html，以.jpg或.jpeg为扩展名的文件其MIME类型被假设为image/jpeg。

在本实验中，服务程序在任一时刻只需服务于一个连接，即服务程序可以在一个无限循环中读取请求并发送响应。在合成与发送响应时，服务程序将无法处理客户程序传来的其他请求。这一缺陷将在下一实验步骤中得到改进。

1.1.3 实验分析

为了更好地理解 HTTP 协议的通讯过程，本节将演示一个简单的 HTTP 客户端程序与服务程序。HTTP 协议最简单、最常见的消息是 GET url HTTP/1.1。该服务程序只处理这一种形式的客户请求，将 GET 命令中由 URL 指定的文档送回客户程序。HTTP 协议的另一重要消息是 POST 命令，用于向服务程序提交表单中的数据。

表 **Error! No text of specified style in document.**-1 HTTP客户端程序例程

SimpleHTTPClient.java	客户端程序	程序 Error! **No text of specified style in document.**-3
SimpleHTTPClientHelper.java	客户端帮助类	程序 Error! **No text of specified style in document.**-2
StreamSocket.java	流 Socket 类	程序 Error! **No text of specified style in document.**-1

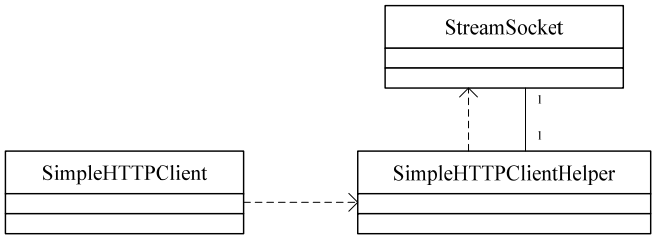

图 **Error! No text of specified style in document.**-1 HTTP客户端程序类图

程序 **Error! No text of specified style in document.**-1 流Socket类StreamSocket.java例程

```
// 自定义流连接套接字
import java.io.*;
import java.net.*;

public class StreamSocket {
        // 连接套接字
        private Socket socket;
        // 输入缓存
        private BufferedReader input;
        // 输出缓存
        private PrintWriter output;
        private OutputStream outStream;
        // 根据已有的套接字创建自定义流套接字
        public StreamSocket(Socket s) throws IOException {
                socket = s;
                setStreams();
        }
        // 根据主机名、监听端口建立流套接字
        public StreamSocket(String acceptorHost, int acceptorPort)
```

```
                                throws UnknownHostException, IOException {
                socket = new Socket(acceptorHost, acceptorPort);
                setStreams();
        }
        // 向服务端发送消息
        public void sendMessage(String message) {
                output.print(message);
                output.flush();
        }
        // 从服务端接收消息
        public String receiveMessage() throws IOException {
                String message = input.readLine();
                return message;
        }
        // 设置数据流
        private void setStreams() throws IOException {
                InputStream inStream = socket.getInputStream();
                input = new BufferedReader(new InputStreamReader(inStream));
                outStream = socket.getOutputStream();
                output = new PrintWriter(new OutputStreamWriter(outStream));
        }
        // 将文件输出到输出流中
        public void sendBytes(String filename) throws Exception {
                InputStream file = new FileInputStream(filename);
                byte[] data = new byte[file.available()];
                file.read(data);
                outStream.write(data);
                file.close();
        }
        // 关闭输入、输出缓存及套接字
        public void close() throws IOException {
                input.close();
                output.close();
                socket.close();
        }
        // 取套接字
        public Socket getSocket() {
                return socket;
        }
        // 设置套接字
        public void setSocket(Socket socket) {
                this.socket = socket;
        }
}
```

程序 Error! **No text of specified style in document.**-1 是流 StreamSocket 类，本实验的客户端与服务端都将使用此程序。StreamSocket 类封装有私有成员连接套接字 Socket、输入缓存 BufferedReader、输出缓存 PrintWriter，StreamSocket 类的构造方法根据主机名、监听端口建立流套接字，setStreams()方法完成设置数据流的工作，sendMessage()方法向服务端发送消息、receiveMessage()方法从服务端接收消息；sendBytes()方法将将文件输出到输出流中，实现从服务器端向客户传递数据文件。

程序 Error! No text of specified style in document.-2 客户端帮助类 SimpleHTTPClientHelper.java 例程

```
// HTTP 客户端程序帮助类
```

```
import java.io.*;
import java.net.*;

public class SimpleHTTPClientHelper {
	private static final String endMessage = ".";
	private StreamSocket streamSocket;
	private String serverHost;
	private int serverPort;
	// 构造函数
	public SimpleHTTPClientHelper(String hostName, String portNum)
			throws UnknownHostException, IOException {
		this.serverHost = hostName;
		this.serverPort = Integer.parseInt(portNum);
		// 反馈给用户连接情况的提示
		try {
			streamSocket = new StreamSocket(serverHost, serverPort);
			System.out.println("连接成功！\n");
		} catch (UnknownHostException u) {
			System.out.println("找不到 HTTP 服务器,连接失败！");
			throw u;
		} catch (IOException i) {
			System.out.println("Socket 连接失败！");
			throw i;
		}
	}
	// 从用户输入的命令行中取出命令
	public String getCommand(String message) {
		String cmd = "";
		// 找到第一个空格出现的位置
		int index = message.indexOf(" ");
		// 如果有空格则 index 之前就是命令,否则是一条没有参数的命令
		if (index >= 0)
			cmd = message.substring(0, index);
		else
			cmd = message;
		// 返回全部为大写字母的命令
		return cmd.toUpperCase();
	}
	// 从用户输入的命令行中取出 URI
	public String getUri(String message) {
		int index = message.indexOf(" ");
		// 得到一个去掉命令的参数行
		if (index >= 0)
			message = message.substring(index + 1, message.length()).trim();
		else
			message = "";
		return message;
	}
	// 客户端和服务端进行交互
	public void process(String req, String uri) {
		String request = req + " " + uri + " HTTP/1.0\n\n";
		try {
			// 发送请求
			streamSocket.sendMessage(request);
```

```
                        // 接收服务器端的响应
                        String response = streamSocket.receiveMessage();
                        // 显示头部信息
                        while (response.length() != 0) {
                                System.out.println(response);
                                response = streamSocket.receiveMessage();
                        }
                        // 将消息体保存到文件中去
                        FileOutputStream os = new FileOutputStream("MessageBody.txt");
                        response = streamSocket.receiveMessage();
                        while (response.length() != 0) {
                                response += "\r\n";
                                byte[] data = new byte[response.length()];
                                data = response.getBytes();
                                os.write(data, 0, data.length);
                                response = streamSocket.receiveMessage();
                        }
                        os.close();
                        System.out.println("消息体保存到 MessageBody.txt!\n");
                } catch (Exception ex) {
                        System.out.println("ERROR : " + ex);
                        ex.printStackTrace(System.out);
                }
        }
        // 关闭 Socket
        public void close() throws SocketException, IOException {
                streamSocket.sendMessage(endMessage);
                streamSocket.close();
        }
}
```

程序 Error! **No text of specified style in document.**-2 是客户端帮助类，根据 HTTP 协议的内容，SimpleHTTPClientHelper 类的构造方法实现与服务端建立连接，getCommand()方法从用户输入的命令行中取出命令，getUri()方法从用户输入的命令行中取出 URI；process()是主要方法，实现客户端与服务端的交互。

程序 Error! No text of specified style in document.-3 客户端程序 SimpleHTTPClient.java 例程

```
// 基于命令行界面的 HTTP 客户程序
import java.io.*;

public class SimpleHTTPClient {
        static final String endMessage = ".";
        public static void main(String[] args) {
                // 控制台的输入流
                BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
                try {
                        System.out.println("\n 请输入服务器主机名(默认主机名为 localhost):  ");
                        String hostName = br.readLine();
                        if (hostName.length() == 0)
                                hostName = "localhost";
                        System.out.println("\n 请输入服务器主机端口号(默认端口号为 8000):  ");
                        String portNum = br.readLine();
                        System.out.println(portNum);
                        if (portNum.length() == 0)
```

```
                portNum = "8000";
            SimpleHTTPClientHelper helper = new SimpleHTTPClientHelper(
                    hostName, portNum);
            boolean done = false;
            String message;
            System.out.println("\n 输入格式:  <HTTP method><space><Request-URI>\n"
                    + "示例:  GET /index.html\n" + "如要退出请输入 . \n");
            while (!done) {
                System.out.println("\n 请输入您的命令>");
                message = br.readLine();
                if ((message.trim()).equals(endMessage)) {
                    done = true;
                    helper.close();
                } else if ((helper.getCommand(message)).equals("GET")) {
                    String req = helper.getCommand(message);
                    String uri = helper.getUri(message);
                    helper.process(req, uri);
                } else {
                    System.out.println("您输入的命令有误，请重新输入！");
                }
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

程序 Error! **No text of specified style in document.**-3 是基于命令行界面的 HTTP 客户主程序，主要处理从控制台读入与处理输入流，读取命令内容，如果命令是“GET”，则向服务端发出请求。

表 Error! No text of specified style in document.-2 HTTP 服务端程序例程

SimpleHTTPServer.java	服务端程序	程序 Error! **No text of specified style in document.**-4
SimpleHTTPServerHelper.java	服务端帮助类	程序 **Error! No text of specified style in document.**-5
StreamSocket.java	流 Socket 类	程序 Error! **No text of specified style in document.**-1

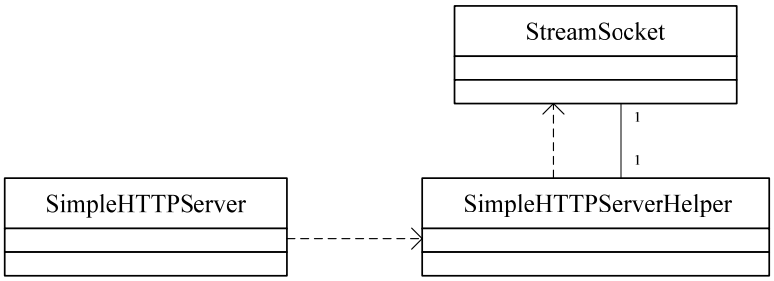

图 **Error! No text of specified style in document.**-2 HTTP服务端程序类图

程序 Error! **No text of specified style in document.**-4 是基于命令行界面的 HTTP 服务主程序，通过命令行读取服务端端口号，启动监听线程，等待客户端请求。当客户端请求到达后，为此请求产生 SimpleHTTPServerHelper 类的实例化对象，并由 SimpleHTTPServerHelper 对象的 processRequest()方法处理请求。

程序 **Error! No text of specified style in document.**-4 服务端程序 SimpleHTTPServer.java 例程

```
// 基于命令行界面的 HTTP 服务程序
import java.io.IOException;
import java.net.*;

public class SimpleHTTPServer {
        public static void main(String args[]) {
                int port;
                ServerSocket listenSocket;
                // 读取 Server 端口号
                try {
                        port = Integer.parseInt(args[0]);
                } catch (Exception e) {
                        port = 8000;
                }
                // 监听 Server 端口，等待连接请求，显示启动信息
                try {
                        listenSocket = new ServerSocket(port);
                        System.out.println("\nHttp 服务程序正在端口" + listenSocket.getLocalPort()
                                        + "处运行");
                        while (true) {
                                System.out.println("等待连接...");
                                Socket socket = listenSocket.accept();
                                System.out.println("\n 客户端： "
                                                + socket.getInetAddress().toString() + " 在端口： "
                                                + socket.getPort() + "的请求连接已经接受！ ");
                                try {
                                        SimpleHTTPServerHelper helper = new SimpleHTTPServerHelper(
                                                        socket);
                                        helper.processRequest();
                                } catch (Exception e) {
                                        System.out.println(e);
                                }
                        }
                } catch (IOException e) {
                        System.out.println(e);
                }
```

```
        }
}
```

程序 **Error! No text of specified style in document.**-5 是服务端帮助类，SimpleHTTPServerHelper 类的构造方法创建 streamSocket 类的实例化对象；SimpleHTTPServerHelper 类的 processRequest()方法处理 HTTP 客户端的请求，具体处理过程是：先读取并显示 HTTP 客户端提交的请求信息，根据实验要求从请求中取出文档的文件名，支持默认索引文件。如果文件存在，则读取文件中的内容并写到 socket 的输出流，并传递给客户端；如果文件不存在，则生成消息体，说明文件不存在，并将消息体传递到客户端。

程序 Error! No text of specified style in document.-5 服务端帮助类 SimpleHTTPServerHelper.java 例程

```
// HTTP 服务程序帮助类
import java.io.*;
import java.net.Socket;
import java.util.StringTokenizer;

public class SimpleHTTPServerHelper {
        private final static String CRLF = System.getProperty("line.separator");
        private StreamSocket streamSocket;

        public SimpleHTTPServerHelper(Socket socket) throws Exception {
                streamSocket = new StreamSocket(socket);
        }
        // 处理 HTTP 客户端的请求
        public void processRequest() throws Exception {
                String statusLine = null;
                String entityBody = null;
                // 读取并显示 HTTP 客户端提交的请求信息
                while (true) {
                        String headerLine = streamSocket.receiveMessage();
                        if (headerLine.trim().equals(".")) {
                                System.out.println("会话结束！");
                                break;
                        } else {
                                if (headerLine.length() != 0)
                                        System.out.println("客户端的请求是：" + headerLine);
                                StringTokenizer s = new StringTokenizer(headerLine);
                                if ((s.countTokens() >= 2) && s.nextToken().equals("GET")) {
                                        // 从请求中取出文档的文件名，支持默认索引文件
                                        String filename = s.nextToken();
                                        if (filename.startsWith("/"))
                                                filename = filename.substring(1);
                                        if (filename.endsWith("/"))
                                                filename += "index.html";
                                        if (filename.equals(""))
                                                filename += "index.html";
                                        // 打开所请求的文件
                                        FileInputStream fis = null;
                                        boolean fileExists = true;
                                        try {
                                                fis = new FileInputStream(filename);
                                                fis.close();
                                        } catch (FileNotFoundException e) {
```

```
                                        fileExists = false;
                                    }
                                    // 读取文件中的内容并写到 socket 的输出流
                                    if (fileExists) {
                                        statusLine = "HTTP/1.0 200 OK" + CRLF;
                                    } else {
                                        statusLine = "HTTP/1.0 404 Not Found" + CRLF;
                                        entityBody = "＜HTML＞" + "＜HEAD＞" + CRLF
                                                + "＜TITLE＞404 Not Found＜/TITLE＞"
+ CRLF
                                                + "＜/HEAD＞ ＜BODY＞" + CRLF +
"<h1>Not Found</h1>"
                                                + CRLF + "<p>The request URI " +
filename
                                                + " was not found on this server.</p>" +
CRLF
                                                + "＜/BODY＞ ＜/HTML＞" + CRLF;
                                    }
                                    streamSocket.sendMessage(statusLine);
                                    streamSocket.sendMessage(CRLF);
                                    if (fileExists) {
                                        streamSocket.sendBytes(filename);
                                        streamSocket.sendMessage(CRLF);
                                        streamSocket.sendMessage(CRLF);
                                    } else {
                                        streamSocket.sendMessage(entityBody);
                                        streamSocket.sendMessage(CRLF);
                                    }
                                }
                            }
                        }
                    }
}
```

§ 1.2　设计实验：改进的 Web 服务器框架

1.2.1　实验要求

1）改进单线程服务器为多线程服务器，实现并发处理请求；

2）综合应用设计模式（工厂方法、模板方法、策略模式、Facade 模式、责任链模式等），构建可扩展的 Web 服务器。

1.2.2　实验分析

(1)　改进服务器为多线程 Web 服务器

一个实用的 Web 服务器通常必须能够处理多个并发请求。所谓两个请求是并发的，是指在处理一个请求尚未结束之前就开始处理另一请求。分布式应用程序的服务端之所以必须具备处理并发请求的理由有许多，常见的原因有资源利用与客户延迟。资源利用，为响应一个 HTTP 请求，服务程序通常需要访问磁盘或数据库，而此时 CPU 和网络信道等资源并未使用。如果服务程序能处理多个并发请求，则可有效地减少这类资源浪费。客户延迟，如果一个服务程序在任一时刻仅能处理一个请求，将导致后续客户请求被不必要地延迟。

为使 Web 服务程序能够处理两个并发请求，每一请求必须在它自己的线程中被处理。然而，只有一个线程可以监听单个 TCP 端口并建立与客户程序之间的连接，因而需要将原来的 Web 服务程序代码划分为两部分：一部分作为监听器（listener），用于建立一个新的 TCP 连接；一部分作为处理器（handler），用于处理请求并将响应发送回客户程序。监听器应为它所建立的每一个 TCP 连接启动一个新的处理器线程。

为在一个独立的线程中执行一些 Java 代码，可定义一个实现了 Runnable 接口的 Java 类，并将这些要执行的代码放在该类的 run()方法中。注意 Runnable 接口的 run()方法不接受任何参数，因而在创建带有参数的线程时通常通过构造方法传递参数，将参数赋值给该类的私有变量。例如，以下源代码定义了一个处理器类 Handler：

```
class Handler implements Runnable {
        private Socket socket;
        // 其他变量
        ...
        public Handler(Socket s, ...) {
        Thread t;
        socket = s;
        ...
        t = new Thread(this, "Handler Thread");
        t.start();
        }
        // 其他方法
        ...
        public void run() {
        // 从输入流中读取请求
        // 处理请求并发送响应（可能使用到其他方法）
        }
}
```

如果要在监听器类**Listener**中调用它，可采用如下方式：

```
        new Handler(s, ...);
```

注意在run()方法中只需要处理单个请求。构建多线程HTTP服务端程序类结构图如图 Error! **No text of specified style in document.**-3 所示。

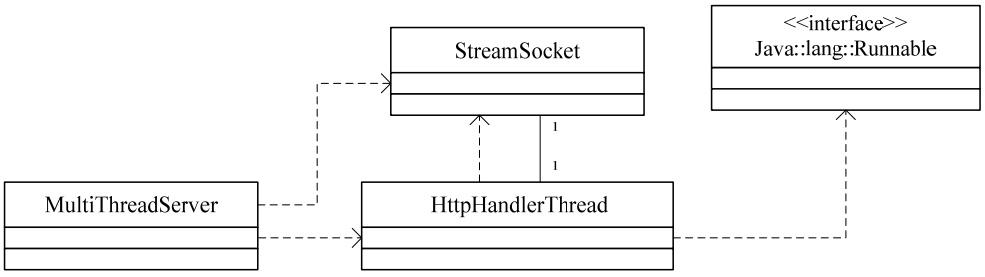

图 Error! No text of specified style in document.-3 多线程HTTP服务端程序类图

(2) **构建 Web 服务器构架，提高重用性**

为了使服务器端能更好的随着协议的发展而进行扩展，搭建了一个简单的框架。主要使用了工厂方法、模板方法、Façade、策略和责任链的设计模式来提高重用性。

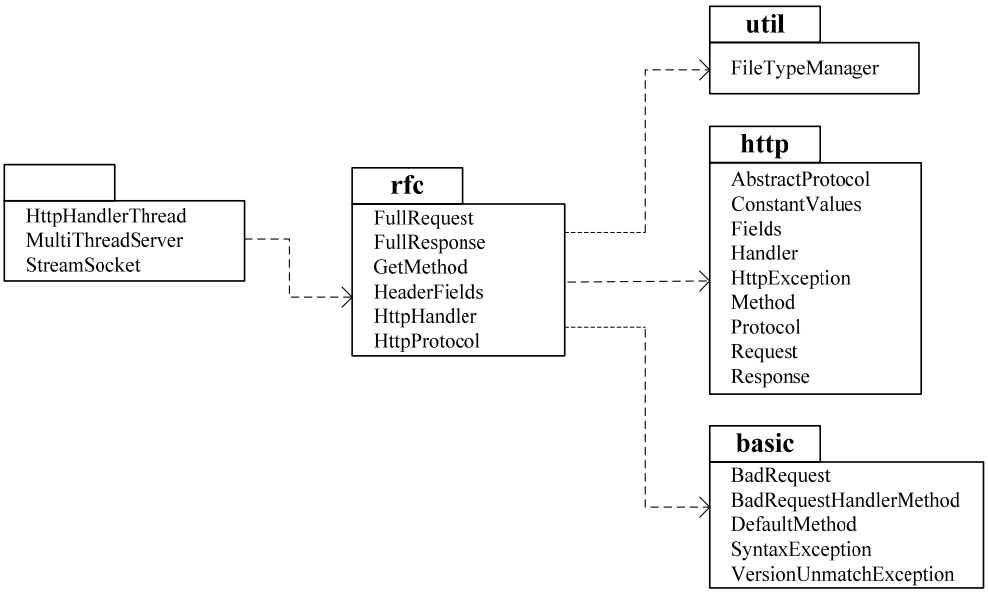

图 **Error! No text of specified style in document.**-4 HTTP服务端程序包部署图

框架中各包的功能如下（HTTP 服务端程序包部署图 如 图 Error! **No text of specified style in document.**-4 所示）：

- http 包：框架中的接口和抽象类
- basic 包：框架提供的 http 包中抽象类的基本实现类。
- rfc1945 包：对 rfc1945，即 HTTP1.0 的部分实现。

Web 服务器构架的工作顺序图如下图所示：

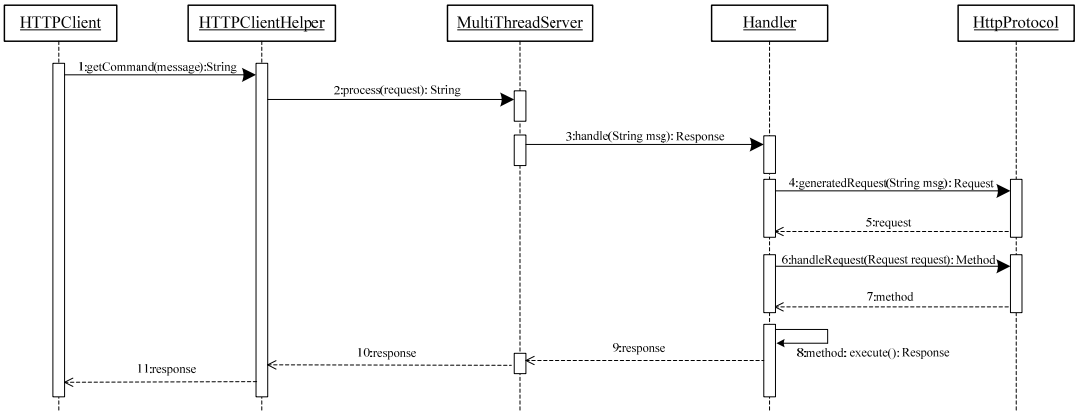

图 **Error! No text of specified style in document.**-5 服务端处理客户端请求时序图

服务器从客户端接收到一条字符串的消息，随后把消息交由 Handler 进行处理，以 Response 响应作为处理结果，再把响应回传给客户。

Handler 收到一条消息后，交给它所了解、持有的协议 Protocol 对消息的格式、参数等进行解析。解析后产生对服务的请求。Handler 获得请求后，调用 Protocol 的 handleRequest 方法，要求协议返回处理相应请求的方法 Method。最后，Handler 执行从 Protocol 处获取的方法并执行该方法，取得响应，返回给 Server。

程序 **Error! No text of specified style in document.**-6 Protocol.java 例程

```
// HTTP 协议是对一串请求消息进行处理,并把处理结果作为响应返回
package http;

public interface Protocol {
	// 根据消息，产生请求
	public Request generateRequest(String message);
	// 根据请求，返回处理请求的命令方法
	public Method handleRequest(Request request);
}
```

程序 **Error! No text of specified style in document.**-6 中的 gererateRequest()方法对所接收到的消息进行预处理，生成请求对象（包括客户端所请求的方法名）。如果只是消息的格式发生了改变，则只需要修改这部分的函数来解析新格式，不用修改执行方法生成响应或是其它地方的代码。handleRequest()是工厂方法，根据所请求的方法名（Request 中的 methodName 属性），返回相应的方法，供进一步执行。由于使用了工厂方法模式，当需要添加一个方法，只需要在继承 Method 抽象类后，把方法添加到该工厂方法中即可，无需改动其它地方对方法的调用。

【提示】工厂方法(Factory Method)提供创建对象的接口，是常用的设计模式。定义一个用于创建对象的接口，让子类决定实例化哪一个类；该模式允许一个类将实例化工作推迟到其子类中完成。与工厂模式密切相关的是抽象工厂(Abstract Factory)，区别在于需要创建对象的复杂程度，抽象工厂模式提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

程序 **Error! No text of specified style in document.**-7 Handler 类的 handle()方法，提供了 handle()模板方法。它通过组织接口 Protocol 中的两个函数，来处理 HTTP 消息。

【提示】模板方法模式（Template Method）：在一个操作中仅定义算法的框架，而将算法中的某些步骤推迟到子类中定义，从而使子类不必改变算法结构即可重定义该算法的某些步骤。

程序 **Error! No text of specified style in document.**-7 Handler.java 例程

```
// HTTP 头部域 HeaderField=FieldName:FieldValue
package http;

public abstract class Handler {
	// 处理器所依赖的协议
	protected Protocol protocol;

	public Handler(Protocol protocol) {
		setProtocol(protocol);
	}
	// 对请求消息进行处理并返回响应
	public Response handle(String message) {
		// 根据消息产生请求
		Request request = protocol.generateRequest(message);
		// 根据请求返回相应的处理方法
		Method method = protocol.handleRequest(request);
	   // 执行方法处理请求，生成响应做为处理结果，并返回该响应
		return method.execute(request);
	}
	public Protocol getProtocol() {
		return protocol;
	}
```

```
        public void setProtocol(Protocol protocol) {
                this.protocol = protocol;
        }
}
```

当服务器接收一个来自客户端的字符串消息，然后把消息作为参数传递给处理器 Handler 类（程序 **Error! No text of specified style in document.**-7）。在 Handler 类中，通过模板方法 handle()，处理消息后生成 Response 回复返回给服务端，再由服务器把 Response 回复给相应的客户。

程序 Error! **No text of specified style in document.**-8 采用策略模式的方法，通过为构造函数传递不同的 Protocol 实现类或通过 setProtocol()方法，就可以动态的替换不同版本的实现类，灵活的提供对不同版本的支持。

【提示】策略模式（Strategy）：定义一系列算法并逐一封装起来，并让它们可互换，从而使算法可独立于其客户程序而变化。策略模式属于设计模式中对象行为型模式，主要是定义一系列的算法，把这些算法一个个封装成单独的类。

程序 Error! No text of specified style in document.-8 Request.java 例程

```
//描述用户的请求
package http;

public abstract class Request {
        //请求要执行的方法的名字
        String methodName;
        //执行方法时所需要的资源位置
        String uri;
        //执行方法时需要的其它条件
        Fields fields;

        public String getMethodName() {
                return methodName;
        }
        public void setMethodName(String methodName) {
                this.methodName = methodName;
        }
        public String getUri() {
                return uri;
        }
        public void setUri(String uri) {
                this.uri = uri;
        }
        public Fields getFields() {
                return fields;
        }
        public void setFields(Fields fields) {
                this.fields = fields;
        }
}
```

程序 Error! **No text of specified style in document.**-8 Request 抽象类描述来自客户端的请求。它包括所请求要执行的方法、方法所需资源的 uri 以及其他一些条件属性（fields）。

程序 **Error! No text of specified style in document.**-9 Fields.java 例程

```
// 用于存储 HTTP 头部域和其它执行请求时需要的参数
// 其中 HTTP 头部域 HeaderField=FieldName:FieldValue
package http;

import java.util.Map;
import java.util.TreeMap;

public abstract class Fields {
        Map<String, String> fields = new TreeMap<String, String>();
        // 加入一个头部域
        public void addFieldValue(String fieldName, String value) {
                if (validateFieldValue(fieldName, value))
                        fields.put(fieldName, value);
        }
        // 获得一个头部域的值
        public String getValue(String fieldName) {
                return fields.get(fieldName);
        }
        // 校验一个头部域及其值是否合法
        protected abstract boolean validateFieldValue(String fieldName, String value);
}
```

程序 **Error! No text of specified style in document.**-9 的 validateFieldValue()方法可以根据实际需要，为不同的头部域提供统一的接口进行校验。它根据头部域名，调用不同的校验函数，对头部域值进行校验。若校验合法则返回 true。由于采用了刻面模式（Facade），可以不必理会具体的校验函数，只需把头部域名和头部域值传入该函数，即可进行相应的校验。

【提示】刻面模式（Facade）：为子系统中的一组接口提供一个统一的接口，或提供一个一致的界面。刻面模式定义了一个高层接口，使子系统更易于使用。facade实际上是个理顺系统间关系，降低系统间耦合度的一个常用的办法。

程序 **Error! No text of specified style in document.**-10 Method.java 例程

```
// 处理请求的方法
package http;

public abstract class Method {
        // 处理请求，生成响应
        public abstract Response execute(Request request);
}
```

程序 **Error! No text of specified style in document.**-10 的 execute()方法执行用户的请求，并生成一个响应返回。

程序 **Error! No text of specified style in document.**-11 Response.java 例程

```
package http;
import java.io.InputStream;

public abstract class Response {
        // 请求的头部
        StringBuffer header = new StringBuffer();
        // WEB 对象，这里简化成以流的形式来表达
        InputStream input;
```

```
        // 为头部添加一行
        public void addHeader(String line) {
                header.append(line);
        }
        public StringBuffer getHeader() {
                return header;
        }
        public void setHeader(StringBuffer header) {
                this.header = header;
        }
        public InputStream getInput() {
                return input;
        }
        public void setInput(InputStream input) {
                this.input = input;
        }
}
```

程序 **Error! No text of specified style in document.**-11 描述服务端响应结果包括一个 HTTP 响应消息头，如果需要传输数据给客户端，则将数据转成流，存放在输入流 InputStream 中，服务器再转发给客户端。

程序 **Error! No text of specified style in document.**-12 AbstractProtocol.java 例程

```
// 为提高协议的各版本的兼容性，框架所提供的协议抽象类/
package http;

public abstract class AbstractProtocol implements Protocol {
        // 当前协议的版本号
        protected String version;
        // 旧的协议
        protected Protocol oldProtocol;

        abstract public Request generateRequest(String message);

        abstract public Method handleRequest(Request request);

        public String getVersion() {
                return version;
        }
        public void setVersion(String version) {
                this.version = version;
        }

        public Protocol getOldProtocol() {
                return oldProtocol;
        }
        public void setOldProtocol(Protocol oldProtocol) {
                this.oldProtocol = oldProtocol;
        }
}
```

为了增加对旧有协议的兼容性，程序 **Error! No text of specified style in document.**-12 AbstractProtocol 持有一个旧的协议，当遇到无法处理的消息、请求

时，只需要转发这些消息请求到旧的协议即可。通过这种方式，可以递归地形成一条协议链（责任链模式），来处理接收到的消息。

【提示】责任链模式（Chain of Responsibility）：使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间紧耦合。将接收对象组织成一条链，并沿该链传递请求，直至有一个对象处理它。

程序 **Error! No text of specified style in document.**-13 是具体类 HttpProtocol 的应用示例：

程序 Error! No text of specified style in document.-13 HttpProtocol.java 例程

```
public Method handleRequest(Request request) {
        try {
                if (request instanceof BadRequest) {
                        return new BadRequestHandlerMethod();
                }
        // 处理请求
                if (request.getMethodName().equals("GET")) {
                        return new GetMethod();
                }
//若执行到此，表示当前协议没有与方法名对应的方法，尝试使用旧协议
                if (getOldProtocol() != null) {
                        return getOldProtocol().handleRequest(request);
                }
        } catch (Exception e) {
        }
        return new DefaultMethod();
}
```