

模拟实现 HTTP 服务器

根据我们学习的 网络编程 以及 HTTP 协议 的相关知识, 我们可以自己模拟实现一个 HTTP 服务器. (类似于 Tomcat 的效果).

版本1

实现一个最简单的 HTTP 服务器.

在这个版本中, 我们只是简单解析 GET 请求, 并根据请求的路径来构造出不同的响应.

- 路径为 /200, 返回一个 "欢迎页面".
- 路径为 /404, 返回一个 "没有找到" 的页面.
- 路径为 /302, 重定向到其他的页面.

创建 HttpServer 类.

- 先初始化 ServerSocket 和 线程池
- 在主循环中循环调用 accept 获取连接. 一旦获取到连接就立刻构造一个任务加入到线程池中. 这个任务负责解析请求并构造响应.
- 在线程池任务中, 先读取请求数据, 按行读取出首行和 header 部分. body 暂时不处理.
- 根据请求的 URL 的路径, 分别构造 "欢迎页面", "没有找到页面", 和重定向响应.

1) 编写代码框架

```
public class HttpServer {
    // HTTP 底层要基于 TCP 来实现. 需要按照 TCP 的基本格式来先进行开发.
    private ServerSocket serverSocket = null;

    public HttpServer(int port) throws IOException {
        serverSocket = new ServerSocket(port);
    }

    public void start() throws IOException {
        System.out.println("服务器启动");
        ExecutorService executorService = Executors.newCachedThreadPool();
        while (true) {
            // 1. 获取连接
            Socket clientSocket = serverSocket.accept();
            // 2. 处理连接(使用短连接的方式实现)
            executorService.execute(new Runnable() {
                @Override
                public void run() {
                    process(clientSocket);
                }
            });
        }
    }

    private void process(Socket clientSocket) {
```

```

    }

    public static void main(String[] args) throws IOException {
        HttpServer server = new HttpServer(9090);
        server.start();
    }
}

```

2) 实现 process 方法

```

private void process(Socket clientSocket) {
    // 由于 HTTP 协议是文本协议，所以仍然使用字符流来处理。
    try (BufferedReader bufferedReader = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
        BufferedWriter bufferedWriter = new BufferedWriter(new
OutputStreamWriter(clientSocket.getOutputStream())) {
        // 下面的操作都要严格按照 HTTP 协议来进行操作。
        // 1. 读取请求并解析
        // a) 解析首行，三个部分使用空格切分
        String firstLine = bufferedReader.readLine();
        String[] firstLineTokens = firstLine.split(" ");
        String method = firstLineTokens[0];
        String url = firstLineTokens[1];
        String version = firstLineTokens[2];
        // b) 解析 header，按行读取，然后按照冒号空格来分割键值对
        Map<String, String> headers = new HashMap<>();
        String line = "";
        // readLine 读取的一行内容，是会自动去掉换行符的。对于空行来说，去掉了换行符，就变
        成空字符串
        while ((line = bufferedReader.readLine()) != null && !line.equals("")) {
            // 不能使用 : 来切分。像 referer 字段，里面的内容是可能包含 : .
            String[] headerTokens = line.split(": ");
            headers.put(headerTokens[0], headerTokens[1]);
        }
        // c) 解析 body (暂时先不考虑)
        // 请求解析完毕，加上一个日志，观察请求的内容是否正确。
        System.out.printf("%s %s %s\n", method, url, version);
        for (Map.Entry<String, String> entry : headers.entrySet()) {
            System.out.println(entry.getKey() + ": " + entry.getValue());
        }
        System.out.println();
        // 2. 根据请求计算响应
        // 不管是啥样的请求，咱们都返回一个 hello 这样的 html
        String body = "";
        if (url.equals("/200")) {
            bufferedWriter.write(version + " 200 OK\n");
            body = "<h1>hello</h1>";
        } else if (url.equals("/404")) {
            bufferedWriter.write(version + " 404 Not Found\n");
            body = "<h1>not found</h1>";
        } else if (url.equals("/302")) {
            bufferedWriter.write(version + " 302 Found\n");
            bufferedWriter.write("Location: http://www.sogou.com\n");
            body = "";
        } else {
            bufferedWriter.write(version + " 200 OK\n");

```

```

        body = "<h1>default</h1>";
    }
    // 3. 把响应写回到客户端
    bufferedWriter.write("Content-Type: text/html\n");
    bufferedWriter.write("Content-Length: " + body.getBytes().length +
"\n"); // 此处的长度，不能写成 body.length()，得到的是字符的数目，而不是字节的数目
    bufferedWriter.write("\n");
    bufferedWriter.write(body);
    // 此处这个 flush 就算没有，问题也不大。紧接着
    // bufferedWriter 对象就要被关闭了。close 时就会自动触发刷新操作。
    bufferedWriter.flush();
    clientSocket.close();
} catch (IOException e) {
    e.printStackTrace();
}
}
}

```

运行程序, 通过 URL

- `http://127.0.0.1:9090/200`
- `http://127.0.0.1:9090/404`
- `http://127.0.0.1:9090/302`

分别访问服务器. 观察效果.

版本2

在这个版本中, 我们只是简单解析 GET 请求, 并根据请求的路径来构造出不同的响应.

在版本1 的基础上, 我们做出一些改进:

- 把解析请求和构造响应的代码提取成单独的类.
- 能够把 URL 中的 query string 解析成键值对.
- 能够给浏览器返回 Cookie.

1. 创建 HttpRequest 类

- 对照着 HTTP 请求的格式, 创建属性: method, url, version, headers.
- 创建 parameters, 用于存放 query string 的解析结果.
- 创建一个静态方法 build, 用来完成解析 HTTP 请求的过程.
- 从 socket 中读取数据的时候注意设置字符编码方式
- 创建一系列 getter 方法获取到请求中的属性.
- 单独写一个方法 parseKV 用来解析 query string

```

// 表示一个 HTTP 请求, 并负责解析.
public class HttpRequest {
    private String method;
    // /index.html?a=10&b=20
    private String url;
    private String version;
    private Map<String, String> headers = new HashMap<>();
    private Map<String, String> parameters = new HashMap<>();
}

```

```

// 请求的构造逻辑，也使用工厂模式来构造。
// 此处的参数，就是从 socket 中获取到的 InputStream 对象
// 这个过程本质上就是在 "反序列化"
public static HttpRequest build(InputStream inputStream) throws IOException
{
    HttpRequest request = new HttpRequest();
    // 此处的逻辑中，不能把 bufferedReader 写到 try ( ) 中。
    // 一旦写进去之后意味着 bufferedReader 就会被关闭，会影响到 clientSocket 的状态。
    // 等到最后整个请求处理完了，再统一关闭
    BufferedReader bufferedReader = new BufferedReader(new
InputStreamReader(inputStream, "UTF-8"));
    // 此处的 build 的过程就是解析请求的过程。
    // 1. 解析首行
    String firstLine = bufferedReader.readLine();
    String[] firstLineTokens = firstLine.split(" ");
    request.method = firstLineTokens[0];
    request.url = firstLineTokens[1];
    request.version = firstLineTokens[2];
    // 2. 解析 url 中的参数
    int pos = request.url.indexOf("?");
    if (pos != -1) {
        // 看看 url 中是否有 ? 。如果没有，就说明不带参数，也就不必解析了
        // 此处的 parameters 是希望包含整个 参数 部分的内容
        // pos 表示 ? 的下标
        // /index.html?a=10&b=20
        // parameters 的结果就相当于 a=10&b=20
        String parameters = request.url.substring(pos + 1);
        // 切分的最终结果，key a, value 10; key b, value 20;
        parseKV(parameters, request.parameters);
    }
    // 3. 解析 header
    String line = "";
    while ((line = bufferedReader.readLine()) != null && line.length() != 0)
    {
        String[] headerTokens = line.split(": ");
        request.headers.put(headerTokens[0], headerTokens[1]);
    }
    // 4. 解析 body (暂时先不考虑)
    return request;
}

private static void parseKV(String input, Map<String, String> output) {
    // 1. 先按照 & 切分成若干组键值对
    String[] kvTokens = input.split("&");
    // 2. 针对切分结果再分别进行按照 = 切分，就得到了键和值
    for (String kv : kvTokens) {
        String[] result = kv.split("=");
        output.put(result[0], result[1]);
    }
}

// 给这个类构造一些 getter 方法。(不要搞 setter)。
// 请求对象的内容应该是从网络上解析来的。用户不应该修改。
public String getMethod() {
    return method;
}

public String getUrl() {

```

```

        return url;
    }

    public String getVersion() {
        return version;
    }

    // 此处的 getter 手动写，自动生成的版本是直接得到整个 hash 表。
    // 而我们需要的是根据 key 来获取值。
    public String getHeader(String key) {
        return headers.get(key);
    }

    public String getParameter(String key) {
        return parameters.get(key);
    }

    @Override
    public String toString() {
        return "HttpRequest{" +
            "method='" + method + '\'' +
            ", url='" + url + '\'' +
            ", version='" + version + '\'' +
            ", headers=" + headers +
            ", parameters=" + parameters +
            '}';
    }
}

```

2. 创建 HttpResponse 类

- 根据 HTTP 响应, 创建属性: version, status, message, headers, body
- 另外创建一个 OutputStream, 用来关联到 Socket 的 OutputStream.
- 往 socket 中写入数据的时候注意指定字符编码方式.
- 创建一个静态方法 build, 用来构造 HttpResponse 对象.
- 创建一系列 setter 方法, 用来设置 HttpResponse 的属性.
- 创建一个 flush 方法, 用于最终把数据写入 OutputStream.

```

// 表示一个 HTTP 响应，负责构造
// 进行序列化操作
public class HttpResponse {
    private String version = "HTTP/1.1";
    private int status; // 状态码
    private String message; // 状态码的描述信息
    private Map<String, String> headers = new HashMap<>();
    private StringBuilder body = new StringBuilder(); // 方便一会进行拼接。
    // 当代码需要把响应写回给客户端的时候，就往这个 OutputStream 中写就好了。
    private OutputStream outputStream = null;

    public static HttpResponse build(OutputStream outputStream) {
        HttpResponse response = new HttpResponse();
        response.outputStream = outputStream;
        // 除了 outputStream 之外，其他的属性的内容，暂时都无法确定。要根据代码的具体业务

```

逻辑

```

        // 来确定。（服务器的“根据请求并计算响应”阶段来进行设置的）
        return response;
    }

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

    public void setStatus(int status) {
        this.status = status;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public void setHeader(String key, String value) {
        headers.put(key, value);
    }

    public void writeBody(String content) {
        body.append(content);
    }

    // 以上的设置属性的操作都是在内存中倒腾。
    // 还需要一个专门的方法，把这些属性 按照 HTTP 协议 都写到 socket 中。
    public void flush() throws IOException {
        BufferedWriter bufferedWriter = new BufferedWriter(new
        OutputStreamWriter(outputStream, "UTF-8"));
        bufferedWriter.write(version + " " + status + " " + message + "\n");
        headers.put("Content-Length", body.toString().getBytes().length + "");
        for (Map.Entry<String, String> entry : headers.entrySet()) {
            bufferedWriter.write(entry.getKey() + ": " + entry.getValue() +
            "\n");
        }
        bufferedWriter.write("\n");
        bufferedWriter.write(body.toString());
        bufferedWriter.flush();
    }
}

```

3. 创建 HttpServer 类

1) 构建代码框架

和之前版本一致。

```

public class HttpServer {
    private ServerSocket serverSocket = null;

    public HttpServer(int port) throws IOException {
        serverSocket = new ServerSocket(port);
    }

    public void start() throws IOException {

```

```

        System.out.println("服务器启动");
        ExecutorService executorService = Executors.newCachedThreadPool();
        while (true) {
            Socket clientSocket = serverSocket.accept();
            executorService.execute(new Runnable() {
                @Override
                public void run() {
                    process(clientSocket);
                }
            });
        }

        private void process(Socket clientSocket) {

        }

        public static void main(String[] args) throws IOException {
            HttpServer server = new HttpServer(9090);
            server.start();
        }
    }

```

2) 实现 process 方法

处理以下 path

- /200 返回一个 hello 页面.
- /add 通过 query string 传递 a 和 b 两个整数, 进行相加操作.
- /cookieUser 通过 Set-Cookie 字段给浏览器返回一个 user=zhangsan 的 Cookie
- /cookieTime 通过 Set-Cookie 字段给浏览器返回一个 time=[时间戳] 的 Cookie

判定 URL path 的时候不能使用 equals 了, 需要使用 startsWith 判定(因为可能包含 query string)

```

private void process(Socket clientSocket) {
    try {
        // 1. 读取并解析请求
        HttpRequest request = HttpRequest.build(clientSocket.getInputStream());
        System.out.println("request: " + request);
        HttpResponse response =
            HttpResponse.build(clientSocket.getOutputStream());
        response.setHeader("Content-Type", "text/html; charset=utf-8");
        // 2. 根据请求计算响应
        if (request.getUrl().startsWith("/200")) {
            response.setStatus(200);
            response.setMessage("OK");
            response.writeBody("<h1>hello</h1>");
        } else if (request.getUrl().startsWith("/add")) {
            // 这个逻辑要根据参数的内容进行计算
            // 先获取到 a 和 b 两个参数的值
            String aStr = request.getParameter("a");
            String bStr = request.getParameter("b");
            // System.out.println("a: " + aStr + ", b: " + bStr);
            int a = Integer.parseInt(aStr);
            int b = Integer.parseInt(bStr);

```

```

        int result = a + b;
        response.setStatus(200);
        response.setMessage("OK");
        response.writeBody("<h1> result = " + result + "</h1>");
    } else if (request.getUrl().startsWith("/cookieUser")) {
        response.setStatus(200);
        response.setMessage("OK");
        // HTTP 的 header 中允许有多个 Set-Cookie 字段。但是
        // 此处 response 中使用 HashMap 来表示 header 的。此时相同的 key 就覆盖
        response.setHeader("Set-Cookie", "user=zhangsan");
        response.writeBody("<h1>set cookieUser</h1>");
    } else if (request.getUrl().startsWith("/cookieTime")) {
        response.setStatus(200);
        response.setMessage("OK");
        // HTTP 的 header 中允许有多个 Set-Cookie 字段。但是
        // 此处 response 中使用 HashMap 来表示 header 的。此时相同的 key 就覆盖
        response.setHeader("Set-Cookie", "time=" +
            (System.currentTimeMillis() / 1000));
        response.writeBody("<h1>set cookieTime</h1>");
    } else {
        response.setStatus(200);
        response.setMessage("OK");
        response.writeBody("<h1>default</h1>");
    }
    // 3. 把响应写回到客户端
    response.flush();
    // 4. 关闭 socket
    clientSocket.close();
} catch (IOException | NullPointerException e) {
    e.printStackTrace();
}
}
}

```

运行程序, 通过以下 URL 验证

1) `http://127.0.0.1:9090/200`

验证是否能显示欢迎页面

2) `http://127.0.0.1:9090/add?a=10&b=20`

验证能否计算出结果

3) `http://127.0.0.1:9090/cookieUser`

验证浏览器能否获取到 `user=zhangsan` 这个 Cookie

4) `http://127.0.0.1:9090/cookieTime`

验证浏览器能否获取到 `user=[时间戳]` 这个 Cookie

版本3

在版本 2 的基础上, 再做出进一步的改进.

- 解析请求中的 Cookie, 解析成键值对.

- 解析请求中的 body, 按照 x-www-form-urlencoded 的方式解析.
- 根据请求方法, 分别调用 doGet / doPost
- 能够返回指定的静态页面.
- 实现简单的会话机制.

1. 创建 HttpRequest 类

代码整体和 版本2 类似, 做出了以下改变

- 属性中新增了 cookies 和 body
- 新增一个方法 parseCookie, 在解析 header 完成后解析 cookie
- 新增了解析 body 的流程.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.HashMap;
import java.util.Map;

public class HttpRequest {
    private String method;
    private String url;
    private String version;
    private Map<String, String> headers = new HashMap<>();
    // url 中的参数和 body 中的参数都放到这个 parameters hash 表中.
    private Map<String, String> parameters = new HashMap<>();
    private Map<String, String> cookies = new HashMap<>();
    private String body;

    public static HttpRequest build(InputStream inputStream) throws IOException
    {
        HttpRequest request = new HttpRequest();
        BufferedReader bufferedReader = new BufferedReader(new
        InputStreamReader(inputStream));
        // 1. 处理首行
        String firstLine = bufferedReader.readLine();
        String[] firstLineTokens = firstLine.split(" ");
        request.method = firstLineTokens[0];
        request.url = firstLineTokens[1];
        request.version = firstLineTokens[2];
        // 2. 解析 url
        int pos = request.url.indexOf("?");
        if (pos != -1) {
            String queryString = request.url.substring(pos + 1);
            parseKV(queryString, request.parameters);
        }
        // 3. 循环处理 header 部分
        String line = "";
        while ((line = bufferedReader.readLine()) != null && line.length() != 0)
        {
            String[] headerTokens = line.split(": ");
            request.headers.put(headerTokens[0], headerTokens[1]);
        }
        // 4. 解析 cookie
```

```

String cookie = request.headers.get("Cookie");
if (cookie != null) {
    // 把 cookie 进行解析
    parseCookie(cookie, request.cookies);
}
// 5. 解析 body
if ("POST".equalsIgnoreCase(request.method)
    || "PUT".equalsIgnoreCase(request.method)) {
    // 这两个方法需要处理 body，其他方法暂时不考虑
    // 需要把 body 读取出来。
    // 需要先知道 body 的长度。Content-Length 就是干这个的。
    // 此处的长度单位是 "字节"
    int contentLength = Integer.parseInt(request.headers.get("Content-
Length"));
    // 注意体会此处的含义~~
    // 例如 contentLength 为 100，body 中有 100 个字节。
    // 下面创建的缓冲区长度是 100 个 char（相当于是 200 个字节）
    // 缓冲区不怕长。就怕不够用。这样创建的缓冲区才能保证长度管够~~
    char[] buffer = new char[contentLength];
    int len = bufferedReader.read(buffer);
    request.body = new String(buffer, 0, len);
    // body 中的格式形如：username=tanglaoshi&password=123
    parseKV(request.body, request.parameters);
}
return request;
}

private static void parseCookie(String cookie, Map<String, String> cookies)
{
    // 1. 按照 分号空格 拆分成多个键值对
    String[] kvTokens = cookie.split("; ");
    // 2. 按照 = 拆分每个键和值
    for (String kv : kvTokens) {
        String[] result = kv.split("=");
        cookies.put(result[0], result[1]);
    }
}

private static void parseKV(String queryString, Map<String, String>
parameters) {
    // 1. 按照 & 拆分成多个键值对
    String[] kvTokens = queryString.split("&");
    // 2. 按照 = 拆分每个键和值
    for (String kv : kvTokens) {
        String[] result = kv.split("=");
        parameters.put(result[0], result[1]);
    }
}

public String getMethod() {
    return method;
}

public String getUrl() {
    return url;
}

public String getVersion() {

```

```

        return version;
    }

    public String getBody() {
        return body;
    }

    public String getParameter(String key) {
        return parameters.get(key);
    }

    public String getHeader(String key) {
        return headers.get(key);
    }

    public String getCookie(String key) {
        return cookies.get(key);
    }
}

```

2. 创建 HttpResponse 类

代码和 版本2 完全一致.

```

public class HttpResponse {
    private String version = "HTTP/1.1";
    private int status;
    private String message;
    private Map<String, String> headers = new HashMap<>();
    private StringBuilder body = new StringBuilder();
    private OutputStream outputStream = null;

    public static HttpResponse build(OutputStream outputStream) {
        HttpResponse response = new HttpResponse();
        response.outputStream = outputStream;
        return response;
    }

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

    public void setStatus(int status) {
        this.status = status;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public void setHeader(String key, String value) {
        headers.put(key, value);
    }

    public void writeBody(String content) {

```

```

        body.append(content);
    }

    public void flush() throws IOException {
        BufferedWriter bufferedWriter = new BufferedWriter(new
OutputStreamWriter(outputStream));
        bufferedWriter.write(version + " " + status + " " + message + "\n");
        headers.put("Content-Length", body.toString().getBytes().length + "");
        for (Map.Entry<String, String> entry : headers.entrySet()) {
            bufferedWriter.write(entry.getKey() + ": " + entry.getValue() +
"\n");
        }
        bufferedWriter.write("\n");
        bufferedWriter.write(body.toString());
        bufferedWriter.flush();
    }
}

```

3. 创建 HttpServer 类

1) 实现代码框架

- 新增一个 sessions 成员, 是一个键值对结构, 用来管理会话. key 是一个字符串. value 是一个 User 对象.

```

public class HttpServer {
    private ServerSocket serverSocket = null;
    // session 会话. 指的就是同一个用户的一组访问服务器的操作, 归类到一起, 就是一个会话.
    // 记者来采访你, 记者问的问题就是一个请求, 你回答的内容, 就是一个响应. 一次采访过程中
    // 涉及到很多问题和回答(请求和响应), 这一组问题和回答, 就可以称为是一个 "会话" (整个采访
    的过程)
    // sessions 中就包含很多会话. (每个键值对就是一个会话)
    private HashMap<String, User> sessions = new HashMap<>();

    public HttpServer(int port) throws IOException {
        serverSocket = new ServerSocket(port);
    }

    public void start() throws IOException {
        System.out.println("服务器启动");
        ExecutorService executorService = Executors.newCachedThreadPool();
        while (true) {
            Socket clientSocket = serverSocket.accept();
            executorService.execute(new Runnable() {
                @Override
                public void run() {
                    process(clientSocket);
                }
            });
        }
    }

    public static void main(String[] args) throws IOException {
        HttpServer server = new HttpServer(9090);
        server.start();
    }
}

```

```
}  
}
```

- User 对象的定义: 为了简单, 直接把属性都定义成 public.

```
public class User {  
    // 保存用户的相关信息  
    public String userName;  
    public int age;  
    public String school;  
}
```

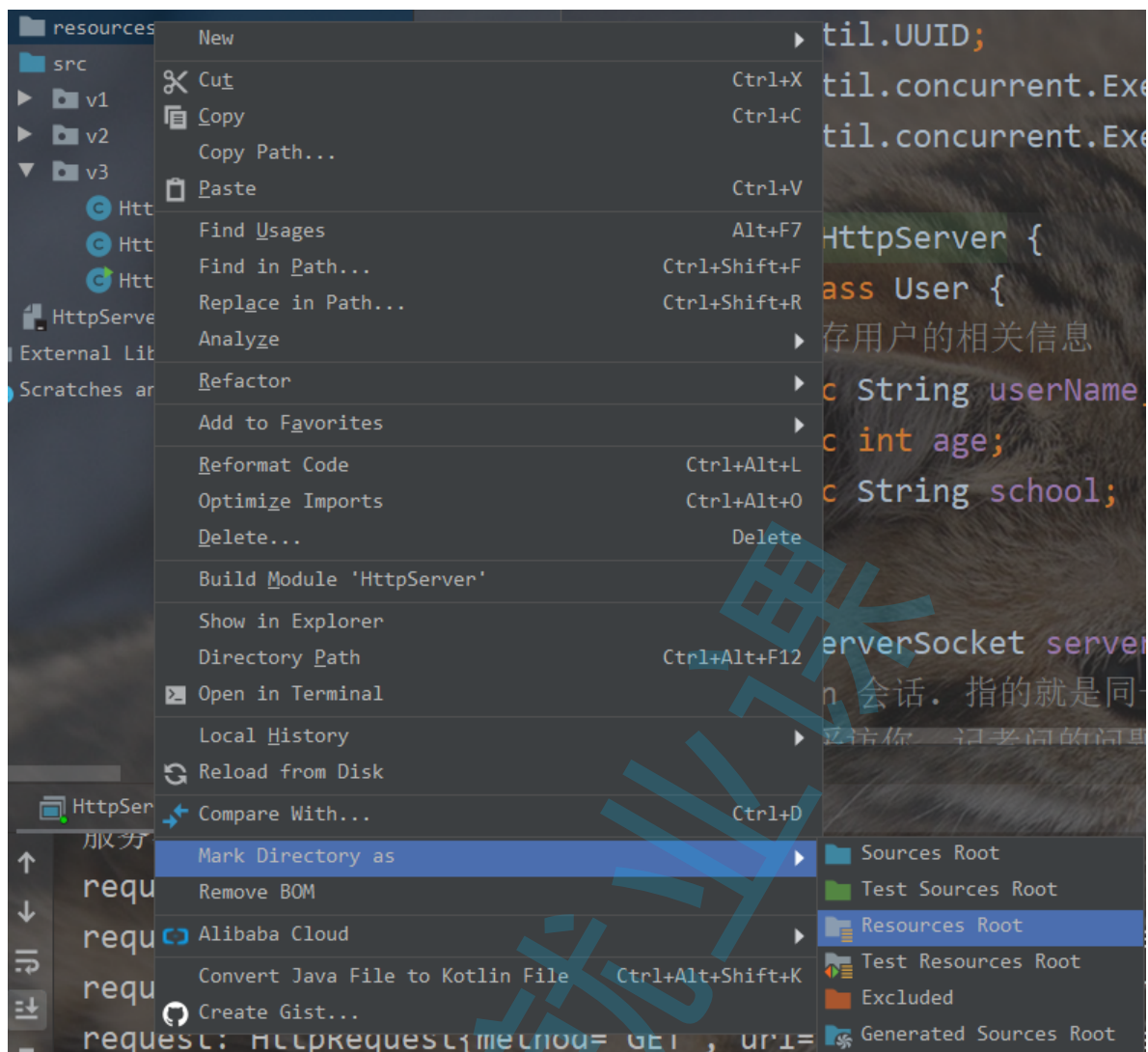
2) 实现 process 方法

- 根据请求方法的不同, 分别调用 doGet 和 doPost

```
public void process(Socket clientSocket) {  
    // 处理核心逻辑  
    try {  
        // 1. 读取请求并解析  
        HttpRequest request = HttpRequest.build(clientSocket.getInputStream());  
        HttpResponse response =  
        HttpResponse.build(clientSocket.getOutputStream());  
        // 2. 根据请求计算响应  
        // 此处按照不同的 HTTP 方法, 拆分成多个不同的逻辑  
        if ("GET".equalsIgnoreCase(request.getMethod())) {  
            doGet(request, response);  
        } else if ("POST".equalsIgnoreCase(request.getMethod())) {  
            doPost(request, response);  
        } else {  
            // 其他方法, 返回一个 405 这样的状态码  
            response.setStatus(405);  
            response.setMessage("Method Not Allowed");  
        }  
        // 3. 把响应写回到客户端  
        response.flush();  
        // 4. 关闭 socket  
        clientSocket.close();  
    } catch (IOException | NullPointerException e) {  
        e.printStackTrace();  
    }  
}
```

3) 实现 doGet

- 如果请求的路径为 /index.html, 则判定用户是否登陆.
- 登陆状态的判定: 先看 Cookie 中是否存在 sessionId, 再看该 sessionId 是否在 sessions 中存在.
- 如果未登陆, 则返回一个静态页面 index.html. 这个页面. index.html 放到 resources 目录中, 并
mark as Resources Root



- 通过 `HttpServer.class.getClassLoader().getResourceAsStream("index.html")` 能够打开该文件, 并读取文件内容.

```
private void doGet(HttpServletRequest request, HttpServletResponse response) throws
IOException {
    // 1. 能够支持返回一个 html 文件.
    if (request.getUrl().startsWith("/index.html")) {
        String sessionId = request.getCookie("sessionId");
        User user = sessions.get(sessionId);
        if (sessionId == null || user == null) {
            // 说明当前用户尚未登陆, 就返回一个登陆页面即可.

            // 这种情况下, 就让代码读取一个 index.html 这样的文件.
            // 要想读文件, 需要知道文件路径. 而现在只知道一个 文件名 index.html
            // 此时这个 html 文件所属的路径, 可以自己来约定(约定某个 d:/...) 专门放 html
            .

            // 把文件内容写入到响应的 body 中
            response.setStatus(200);
            response.setMessage("OK");
            response.setHeader("Content-Type", "text/html; charset=utf-8");
            InputStream inputStream =
                HttpServer.class.getClassLoader().getResourceAsStream("index.html");
            BufferedReader bufferedReader = new BufferedReader(new
                InputStreamReader(inputStream));
            // 按行读取内容, 把数据写入到 response 中
            String line = null;
```

```

        while ((line = bufferedReader.readLine()) != null) {
            response.writeBody(line + "\n");
        }
        bufferedReader.close();
    } else {
        // 用户已经登陆，无需再登陆了。
        // TODO
    }
}
}

```

4) 实现 index.html

- 通过 form 表单, 通过 POST 提交 username 和 password

```

<form action="login" method="POST">
    <input type="text" name="username">
    <input type="text" name="password">
    <input type="submit" value="提交">
</form>

```

5) 实现 doPost

处理 form 表单提交的请求.

- 判定路径是否为 /login
- 前面 HttpRequest 中已经对 body 进行解析了. 通过 getParameter 解析其中的 username 和 password, 并校验.
- 如果用户名密码匹配, 则返回一个登陆成功的页面.
- 登陆成功同时, 构造一个 SessionId 和一个 User 对象, 把这个键值对放到 sessions 中, 并把 sessionId 通过 cookie 返回给浏览器.
- 其中 sessionId 通过 UUID.randomUUID().toString() 来生成一个唯一字符串.
- 如果用户登陆失败, 返回一个登陆失败的页面.

```

private void doPost(HttpRequest request, HttpResponse response) {
    // 2. 实现 /login 的处理
    if (request.getUrl().startsWith("/login")) {
        // 读取用户提交的用户名和密码
        String userName = request.getParameter("username");
        String password = request.getParameter("password");
        // System.out.println("userName: " + userName);
        // System.out.println("password: " + password);
        // 登陆逻辑就需要验证用户名密码是否正确.
        // 此处为了简单, 咱们把用户名和密码在代码中写死了.
        // 更科学的处理方式, 应该是从数据库中读取用户名对应的密码, 校验密码是否一致.
        if ("zhangsan".equals(userName) && "123".equals(password)) {
            // 登陆成功
            response.setStatus(200);
            response.setMessage("OK");
            response.setHeader("Content-Type", "text/html; charset=utf-8");
            // 原来登陆成功, 是给浏览器写了一个 cookie, cookie 中保存的是用户的用户名.
            // response.setHeader("Set-Cookie", "userName=" + userName);

```

```

// 现有的对于登陆成功的处理。给这次登陆的用户分配了一个 session
// (在 hash 中新增了一个键值对), key 是随机生成的。value 就是用户的身份信息
// 身份信息保存在服务器中, 此时也就不再有泄露的问题了
// 给浏览器返回的 Cookie 中只需要包含 sessionId 即可
String sessionId = UUID.randomUUID().toString();
User user = new User();
user.userName = "zhangsan";
user.age = 20;
user.school = "陕科大";
sessions.put(sessionId, user);
response.setHeader("Set-Cookie", "sessionId=" + sessionId);

response.writeBody("<html>");
response.writeBody("<div>欢迎您! " + userName + "</div>");
response.writeBody("</html>");
} else {
    // 登陆失败
    response.setStatus(403);
    response.setMessage("Forbidden");
    response.setHeader("Content-Type", "text/html; charset=utf-8");
    response.writeBody("<html>");
    response.writeBody("<div>登陆失败</div>");
    response.writeBody("</html>");
}
}
}

```

6) 修改 doGet

实现通过 session 获取到用户身份.

```

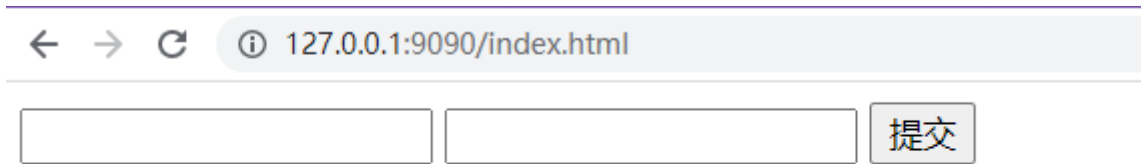
private void doGet(HttpServletRequest request, HttpServletResponse response) throws
IOException {
    // 1. 能够支持返回一个 html 文件.
    if (request.getUrl().startsWith("/index.html")) {
        String sessionId = request.getCookie("sessionId");
        User user = sessions.get(sessionId);
        if (sessionId == null || user == null) {
            // ... 此处代码不变.
        } else {
            // 用户已经登陆, 无需再登陆了.
            response.setStatus(200);
            response.setMessage("OK");
            response.setHeader("Content-Type", "text/html; charset=utf-8");
            response.writeBody("<html>");
            response.writeBody("<div>" + "您已经登陆了! 无需再次登陆! 用户名: " +
user.userName + "</div>");
            response.writeBody(+ user.age + "</div>");
            response.writeBody("<div>" + user.school + "</div>");
            response.writeBody("</html>");
        }
    }
}
}

```

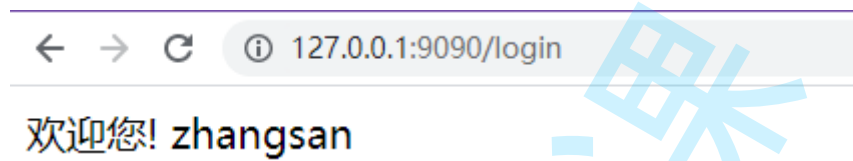

运行程序. 通过以下 URL 验证:

1) `http://127.0.0.1:9090/index.html`

首次访问, 当前未登录, 会看到 index.html 这个登陆页面.



2) 输入用户名密码之后, 如果登陆成功, 预期看到



3) 后续再访问 `http://127.0.0.1:9090/index.html` 时, 由于已经登陆过, 不必重新登陆

