

Request&Response

今日目标

- 掌握Request对象的概念与使用
- 掌握Response对象的概念与使用
- 能够完成用户登录注册案例的实现
- 能够完成SqlSessionFactory工具类的抽取

1, Request和Response的概述

Request是请求对象，Response是响应对象。这两个对象在我们使用Servlet的时候有看到：

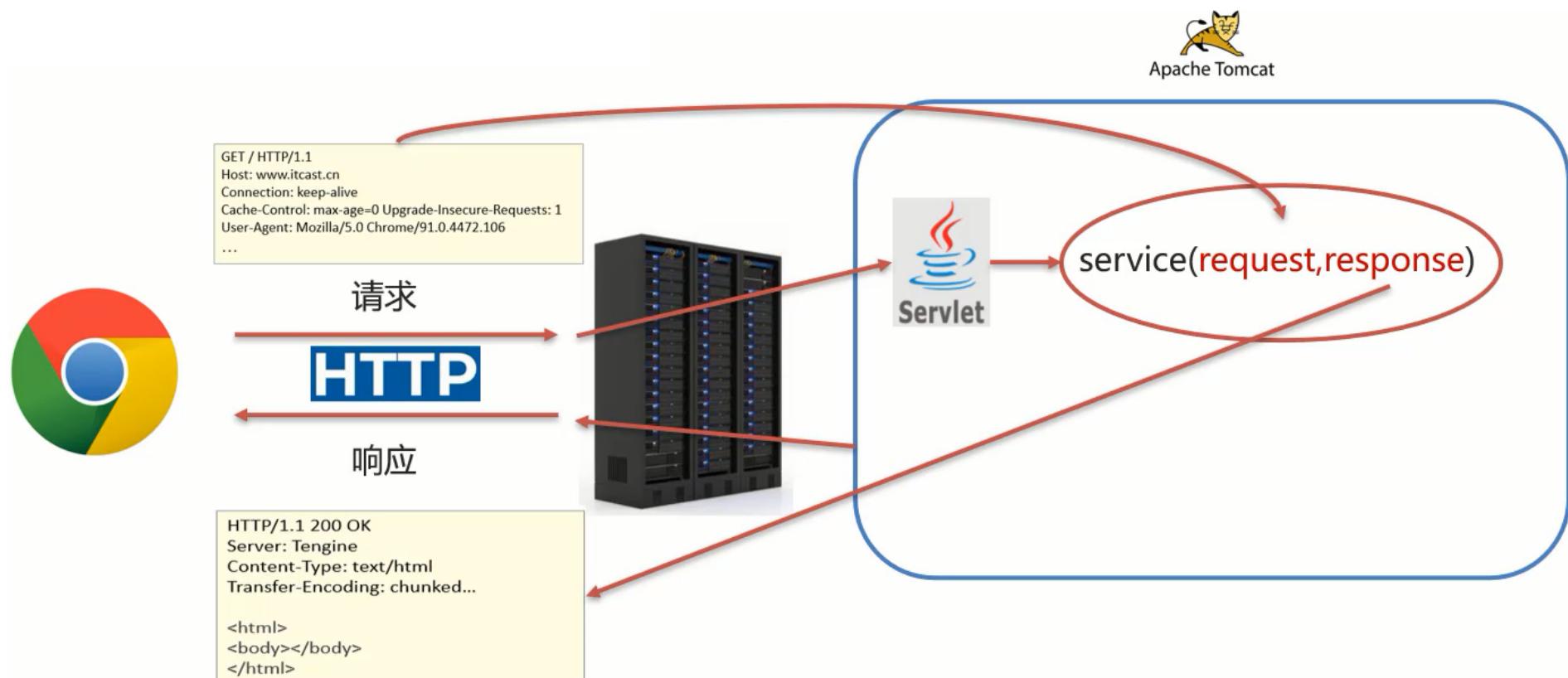
```
@WebServlet("/demo1")
public class ServletDemo1 implements Servlet{

    @Override
    public void service(ServletRequest request, ServletResponse response) throws ServletException, IOException {
        System.out.println("servlet hello ~");
    }

    @Override
    public void init(ServletConfig config) throws ServletException {

    }
}
```

此时，我们就需要思考一个问题request和response这两个参数的作用是什么？



- **request:** 获取请求数据
 - 浏览器会发送HTTP请求到后台服务器 [Tomcat]
 - HTTP的请求中会包含很多请求数据 [请求行+请求头+请求体]
 - 后台服务器 [Tomcat] 会对HTTP请求中的数据进行解析并把解析结果存入到一个对象中
 - 所存入的对象即为request对象，所以我们可以从request对象中获取请求的相关参数
 - 获取到数据后就可以继续后续的业务，比如获取用户名和密码就可以实现登录操作的相关业务
- **response:** 设置响应数据

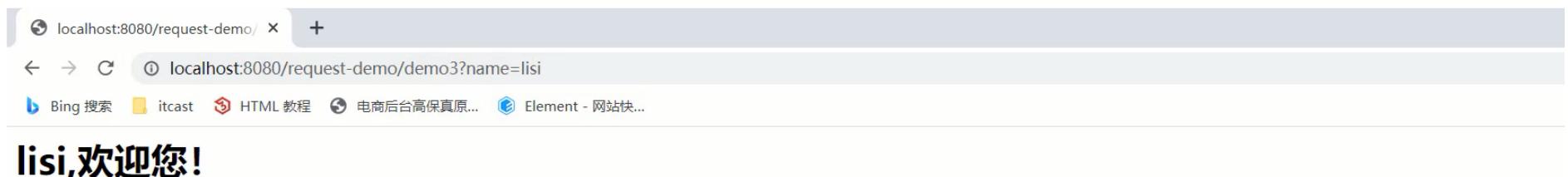
- 业务处理完后，后台就需要给前端返回业务处理的结果即响应数据
- 把响应数据封装到response对象中
- 后台服务器[Tomcat]会解析response对象，按照[响应行+响应头+响应体]格式拼接结果
- 浏览器最终解析结果，把内容展示在浏览器给用户浏览

对于上述所讲的内容，我们通过一个案例来初步体验下request和response对象的使用。

```

1 @WebServlet("/demo3")
2 public class ServletDemo3 extends HttpServlet {
3     @Override
4     protected void doGet(HttpServletRequest request, HttpServletResponse
5             response) throws ServletException, IOException {
6         //使用request对象 获取请求数据
7         String name = request.getParameter("name");//url?name=zhangsan
8
9         //使用response对象 设置响应数据
10        response.setHeader("Content-Type", "text/html; charset=utf-8");
11        response.getWriter().write("<h1>" + name + ",欢迎您! </h1>");
12    }
13
14    @Override
15    protected void doPost(HttpServletRequest request, HttpServletResponse
16            response) throws ServletException, IOException {
17        System.out.println("Post...");
18    }
19 }
```

启动成功后就可以通过浏览器来访问，并且根据传入参数的不同就可以在页面上展示不同的内容：



小结

在这节中，我们主要认识了下request对象和reponse对象：

- request对象是用来封装请求数据的对象
- response对象是用来封装响应数据的对象

目前我们只知道这两个对象是用来干什么的，那么它们具体是如何实现的，就需要我们继续深入的学习。接下来，就先从Request对象来学习，主要学习下面这些内容：

- request继承体系
- request获取请求参数
- request请求转发

2, Request对象

2.1 Request继承体系

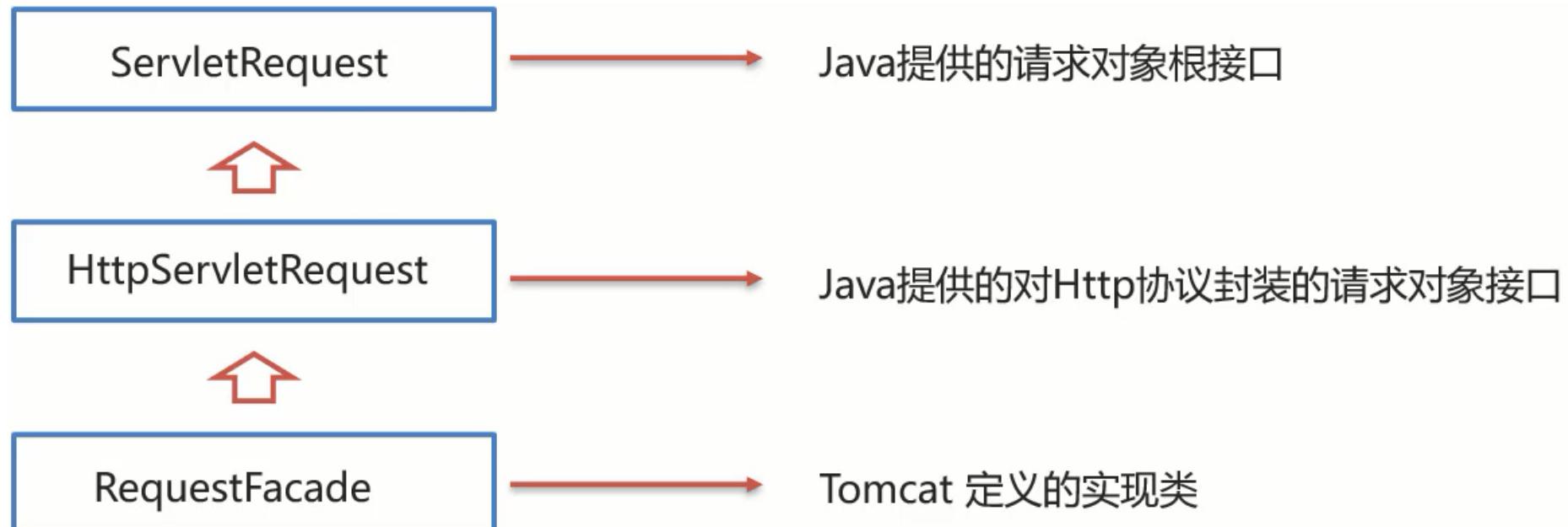
在学习这节内容之前，我们先思考一个问题，前面在介绍Request和Reponse对象的时候，比较细心的同学可能已经发现：

- 当我们的Servlet类实现的是Servlet接口的时候，service方法中的参数是ServletRequest和ServletResponse
- 当我们的Servlet类继承的是HttpServlet类的时候，doGet和doPost方法中的参数就变成HttpServletRequest和HttpServletResponse

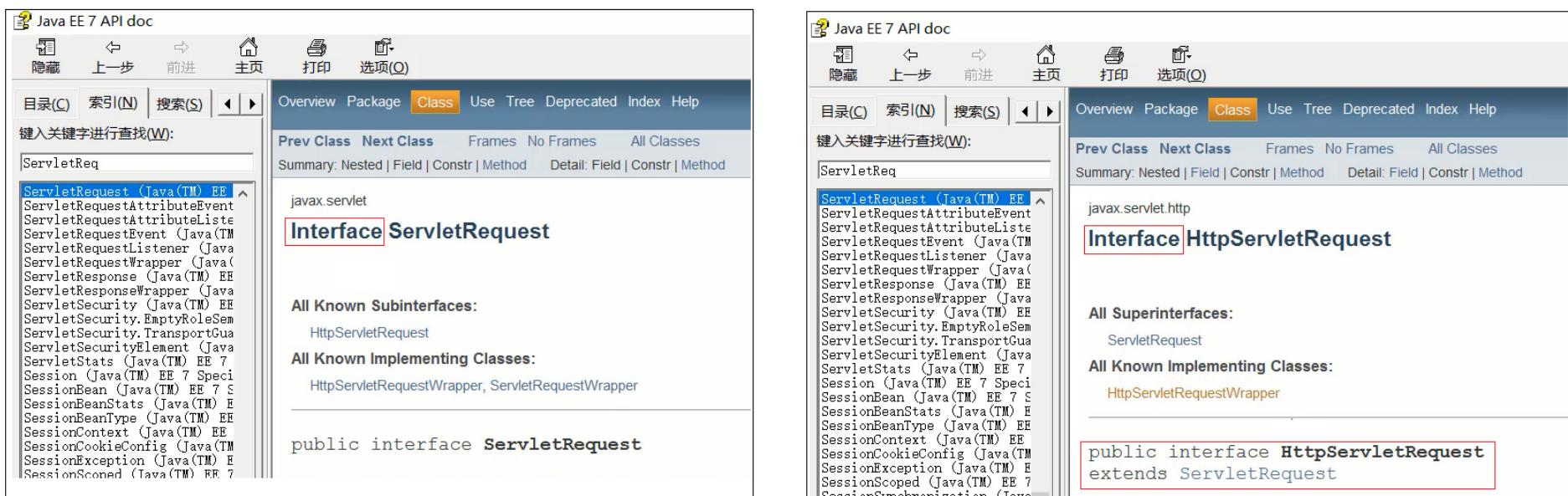
那么，

- ServletRequest和HttpServletRequest的关系是什么？
- request对象是谁来创建的？
- request提供了哪些API，这些API从哪里查？

首先，我们先来看下Request的继承体系：



从上图中可以看出，ServletRequest和HttpServletRequest都是Java提供的，所以我们可以打开JavaEE提供的API文档 [参考：资料/JavaEE7-api.chm]，打开后可以看到：



所以ServletRequest和HttpServletRequest是继承关系，并且两个都是接口，接口是无法创建对象，这个时候就引发了下面这个问题：

```

public class ServletDemo1 implements Servlet {

    @Override
    public void service(ServletRequest request, ServletResponse response) throws ServletException, IOException {
        System.out.println("servlet hello~");
    }
}

@WebServlet("/demo3")
public class ServletDemo3 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        //使用request对象 获取请求数据
        String name = request.getParameter(name: "name"); //url?name=zhangsan

        //使用response对象 设置响应数据
        response.setHeader(name: "content-type", value: "text/html;charset=utf-8");
        response.getWriter().write(s: "<h1>" + name + ", 欢迎您! </h1>");
    }
}

```

这些方法参数中的对象是由谁来创建的呢?

这个时候，我们就需要用到Request继承体系中的RequestFacade：

- 该类实现了HttpServletRequest接口，也间接实现了ServletRequest接口。
- Servlet类中的service方法、doGet方法或者是doPost方法最终都是由Web服务器[Tomcat]来调用的，所以Tomcat提供了方法参数接口的具体实现类，并完成了对象的创建
- 要想了解RequestFacade中都提供了哪些方法，我们可以直接查看JavaEE的API文档中关于ServletRequest和HttpServletRequest的接口文档，因为RequestFacade实现了其接口就需要重写接口中的方法

对于上述结论，要想验证，可以编写一个Servlet，在方法中把request对象打印下，就能看到最终的对象是不是RequestFacade，代码如下：

```

1 @WebServlet("/demo2")
2 public class ServletDemo2 extends HttpServlet {
3     @Override
4     protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
5         System.out.println(request);
6     }
7
8     @Override
9     protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
10    }
11 }

```

启动服务器，运行访问<http://localhost:8080/request-demo/demo2>，得到运行结果：



小结

- Request的继承体系为ServletRequest-->HttpServletRequest-->RequestFacade
- Tomcat需要解析请求数据，封装为request对象，并且创建request对象传递到service方法
- 使用request对象，可以查阅JavaEE API文档的HttpServletRequest接口中方法说明

2.2 Request获取请求数据

HTTP请求数据总共分为三部分内容，分别是**请求行、请求头、请求体**，对于这三部分内容的数据，分别该如何获取，首先我们先来学习请求行数据如何获取？

2.2.1 获取请求行数据

请求行包含三块内容，分别是请求方式、请求资源路径、HTTP协议及版本

GET	/request-demo/req1?username=zhangsan	HTTP/1.1
请求方式	请求资源路径	HTTP协议及版本

对于这三部分内容，`request`对象都提供了对应的API方法来获取，具体如下：

- 获取请求方式：`GET`

```
1 string getMethod()
```

- 获取虚拟目录(项目访问路径)：`/request-demo`

```
1 string getContextPath()
```

- 获取URL(统一资源定位符)：`http://localhost:8080/request-demo/req1`

```
1 StringBuffer getRequestURL()
```

- 获取URI(统一资源标识符)：`/request-demo/req1`

```
1 string getRequestURI()
```

- 获取请求参数(GET方式)：`username=zhangsan&password=123`

```
1 string getQueryString()
```

介绍完上述方法后，咱们通过代码把上述方法都使用下：

```
1 /**
2  * request 获取请求数据
3 */
4 @WebServlet("/req1")
5 public class RequestDemo1 extends HttpServlet {
6     @Override
7     protected void doGet(HttpServletRequest req, HttpServletResponse resp)
8     throws ServletException, IOException {
9         // String getMethod(): 获取请求方式: GET
10        String method = req.getMethod();
11        System.out.println(method); // GET
12        // String getContextPath(): 获取虚拟目录(项目访问路径): /request-demo
```

```

12     String contextPath = req.getContextPath();
13     System.out.println(contextPath);
14     // StringBuffer getRequestURL(): 获取URL(统一资源定位符):
15     // http://localhost:8080/request-demo/req1
16     StringBuffer url = req.getRequestURL();
17     System.out.println(url.toString());
18     // String getRequestURI(): 获取URI(统一资源标识符): /request-demo/req1
19     String uri = req.getRequestURI();
20     System.out.println(uri);
21     // String getQueryString(): 获取请求参数 (GET方式) : username=zhangsan
22     String queryString = req.getQueryString();
23     System.out.println(queryString);
24 }
25 @Override
26 protected void doPost(HttpServletRequest req, HttpServletResponse resp)
27 throws ServletException, IOException {
28 }
29 }
```

启动服务器，访问 `http://localhost:8080/request-demo/req1?username=zhangsan&password=123`，获取的结果如下：

```

GET
/request-demo
http://localhost:8080/request-demo/req1
/request-demo/req1
username=zhangsan&password=123
```

2.2.2 获取请求头数据

对于请求头的数据，格式为 `key: value` 如下：

User-Agent: Mozilla/5.0 Chrome/91.0.4472.106

所以根据请求头名称获取对应值的方法为：

```
1 string getHeader(String name)
```

接下来，在代码中如果想要获取客户端浏览器的版本信息，则可以使用

```

1 /**
2  * request 获取请求数据
3 */
4 @WebServlet("/req1")
5 public class RequestDemo1 extends HttpServlet {
6     @Override
7     protected void doGet(HttpServletRequest req, HttpServletResponse resp)
8     throws ServletException, IOException {
```

```
8     //获取请求头: user-agent: 浏览器的版本信息
9     String agent = req.getHeader("user-agent");
10    System.out.println(agent);
11 }
12 @Override
13 protected void doPost(HttpServletRequest req, HttpServletResponse resp)
14 throws ServletException, IOException {
15 }
16 }
```

重新启动服务器后, `http://localhost:8080/request-demo/req1?`

`username=zhangsan&password=123`, 获取的结果如下:

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.131 Safari/537.36
```

2.2.3 获取请求体数据

浏览器在发送GET请求的时候是没有请求体的, 所以需要把请求方式变更为POST, 请求体中的数据格式如下:

```
username=superbaby&password=123
```

对于请求体中的数据, Request对象提供了如下两种方式来获取其中的数据, 分别是:

- 获取字节输入流, 如果前端发送的是字节数据, 比如传递的是文件数据, 则使用该方法

```
1 ServletInputStream getInputStream()
2 该方法可以获取字节
```

- 获取字符输入流, 如果前端发送的是纯文本数据, 则使用该方法

```
1 BufferedReader getReader()
```

接下来, 大家需要思考, 要想获取到请求体的内容该如何实现?

具体实现的步骤如下:

1. 准备一个页面, 在页面中添加form表单, 用来发送post请求
2. 在Servlet的doPost方法中获取请求体数据
3. 在doPost方法中使用request的getReader()或者getInputStream()来获取
4. 访问测试

1. 在项目的webapp目录下添加一个html页面, 名称为: `req.html`

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
```

```
4     <meta charset="UTF-8">
5     <title>Title</title>
6 </head>
7 <body>
8 <!--
9     action: form表单提交的请求地址
10    method: 请求方式, 指定为post
11 -->
12 <form action="/request-demo/req1" method="post">
13     <input type="text" name="username">
14     <input type="password" name="password">
15     <input type="submit">
16 </form>
17 </body>
18 </html>
```

2. 在Servlet的doPost方法中获取数据

```
1 /**
2  * request 获取请求数据
3 */
4 @WebServlet("/req1")
5 public class RequestDemo1 extends HttpServlet {
6     @Override
7     protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
8     }
9     @Override
10    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
11        //在此处获取请求体中的数据
12    }
13 }
```

3. 调用getReader()或者getInputStream()方法, 因为目前前端传递的是纯文本数据, 所以我们采用getReader()方法来获取

```
1 /**
2  * request 获取请求数据
3 */
4 @WebServlet("/req1")
5 public class RequestDemo1 extends HttpServlet {
6     @Override
7     protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
8     }
9     @Override
```

```

10 protected void doPost(HttpServletRequest req, HttpServletResponse resp)
11     throws ServletException, IOException {
12     //获取post 请求体: 请求参数
13     //1. 获取字符输入流
14     BufferedReader br = req.getReader();
15     //2. 读取数据
16     String line = br.readLine();
17     System.out.println(line);
18 }

```

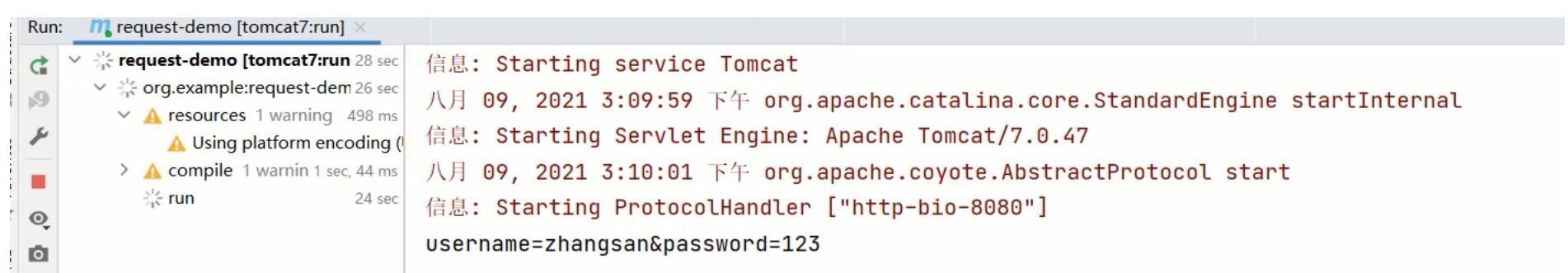
注意

BufferedReader流是通过request对象来获取的，当请求完成后request对象就会被销毁，request对象被销毁后，BufferedReader流就会自动关闭，所以此处就不需要手动关闭流了。

4. 启动服务器，通过浏览器访问 `http://localhost:8080/request-demo/req.html`



点击提交按钮后，就可以在控制台看到前端所发送的请求数据



小结

HTTP请求数据中包含了请求行、请求头和请求体，针对这三部分内容，Request对象都提供了对应的API方法来获取对应的值：

- 请求行
 - `getMethod()` 获取请求方式
 - `getContextPath()` 获取项目访问路径
 - `getRequestURL()` 获取请求URL
 - `getRequestURI()` 获取请求URI
 - `getQueryString()` 获取GET请求方式的请求参数
- 请求头
 - `getHeader(String name)` 根据请求头名称获取其对应的值
- 请求体
 - 注意： 浏览器发送的POST请求才有请求体
 - 如果是纯文本数据：`getReader()`

- 如果是字节数据如文件数据: `getInputStream()`

2.2.4 获取请求参数的通用方式

在学习下面内容之前，我们先提出两个问题：

- 什么是请求参数？
- 请求参数和请求数据的关系是什么？

1. 什么是请求参数？

为了能更好的回答上述两个问题，我们拿用户登录的例子来说明

- 想要登录网址，需要进入登录页面
- 在登录页面输入用户名和密码
- 将用户名和密码提交到后台
- 后台校验用户名和密码是否正确
- 如果正确，则正常登录，如果不正确，则提示用户名或密码错误

上述例子中，用户名和密码其实就是我们所说的请求参数。

2. 什么是请求数据？

请求数据则是包含请求行、请求头和请求体的所有数据

- 请求参数和请求数据的关系是什么？
 - 请求参数是请求数据中的部分内容
 - 如果是GET请求，请求参数在请求行中
 - 如果是POST请求，请求参数一般在请求体中

对于请求参数的获取，常用的有以下两种：

- GET方式：

```
1 string getQueryString()
```

- POST方式：

```
1 BufferedReader getReader();
```

有了上述的知识储备，我们来实现一个案例需求：

- (1) 发送一个GET请求并携带用户名，后台接收后打印到控制台
- (2) 发送一个POST请求并携带用户名，后台接收后打印到控制台

此处大家需要注意的是GET请求和POST请求接收参数的方式不一样，具体实现的代码如下：

```

1 @WebServlet("/req1")
2 public class RequestDemo1 extends HttpServlet {
3     @Override
4         protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
5
6             String result = req.getQueryString();
7             System.out.println(result);
8
9         }
10    @Override
11        protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
12            BufferedReader br = req.getReader();
13            String result = br.readLine();
14            System.out.println(result);
15        }
16 }

```

- 对于上述的代码，会存在什么问题呢？

```

@WebServlet("/req1")
public class RequestDemo1 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        String result = req.getQueryString();
        System.out.println(result);
    }
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        BufferedReader br = req.getReader();
        String result = br.readLine();
        System.out.println(result);
    }
}

```

由于获取请求参数的方式不一样，导致doGet和doPost中出现了重复代码。这行打印大家可以理解为很多相同的业务代码。

- 如何解决上述重复代码的问题呢？

```

@WebServlet("/req1")
public class RequestDemo1 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        // 获取请求参数
        // 将请求参数进行打印控制台
    }
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        this.doGet(req, resp);
    }
}

```

当然，也可以在doGet中调用doPost，在doPost中完成参数的获取和打印，另外需要注意的是，doGet和doPost方法都必须存在，不能删除任意一个。

GET请求和POST请求获取请求参数的方式不一样，在获取请求参数这块该如何实现呢？

要想实现，我们就需要**思考**：

GET请求方式和POST请求方式区别主要在于获取请求参数的方式不一样，是否可以提供一种统一获取请求参数的方式，从而统一doGet和doPost方法内的代码？

解决方案一：

```
1 @WebServlet("/req1")
2 public class RequestDemo1 extends HttpServlet {
3     @Override
4     protected void doGet(HttpServletRequest req, HttpServletResponse resp)
5         throws ServletException, IOException {
6         //获取请求方式
7         String method = req.getMethod();
8         //获取请求参数
9         String params = "";
10        if("GET".equals(method)){
11            params = req.getQueryString();
12        }else if("POST".equals(method)){
13            BufferedReader reader = req.getReader();
14            params = reader.readLine();
15        }
16        //将请求参数进行打印控制台
17        System.out.println(params);
18    }
19    @Override
20    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
21        throws ServletException, IOException {
22        this.doGet(req,resp);
23    }
}
```

使用request的getMethod()来获取请求方式，根据请求方式的不同分别获取请求参数值，这样就可以解决上述问题，但是以后每个Servlet都需要这样写代码，实现起来比较麻烦，这种方案我们不采用

解决方案二：

request对象已经将上述获取请求参数的方法进行了封装，并且request提供的方法实现的功能更大，以后只需要调用request提供的方法即可，在request的方法中都实现了哪些操作？

(1) 根据不同的请求方式获取请求参数，获取的内容如下：

username=zhangsan&hobby=1&hobby=2

(2) 把获取到的内容进行分割，内容如下：

username=zhangsan&hobby=1&hobby=2

username=zhangsan

hobby=1

hobby=2

username

zhangsan

hobby

1

hobby

2

(3) 把分割后端数据，存入到一个Map集合中：



Map<String, String[]>

注意：因为参数的值可能是一个，也可能有多个，所以Map的值的类型为String数组。

基于上述理论，request对象为我们提供了如下方法：

- 获取所有参数Map集合

```
1 Map<String, String[]> getParameterMap()
```

- 根据名称获取参数值（数组）

```
1 String[] getParameterValues(String name)
```

- 根据名称获取参数值（单个值）

```
1 String getParameter(String name)
```

接下来，我们通过案例来把上述的三个方法进行实例演示：

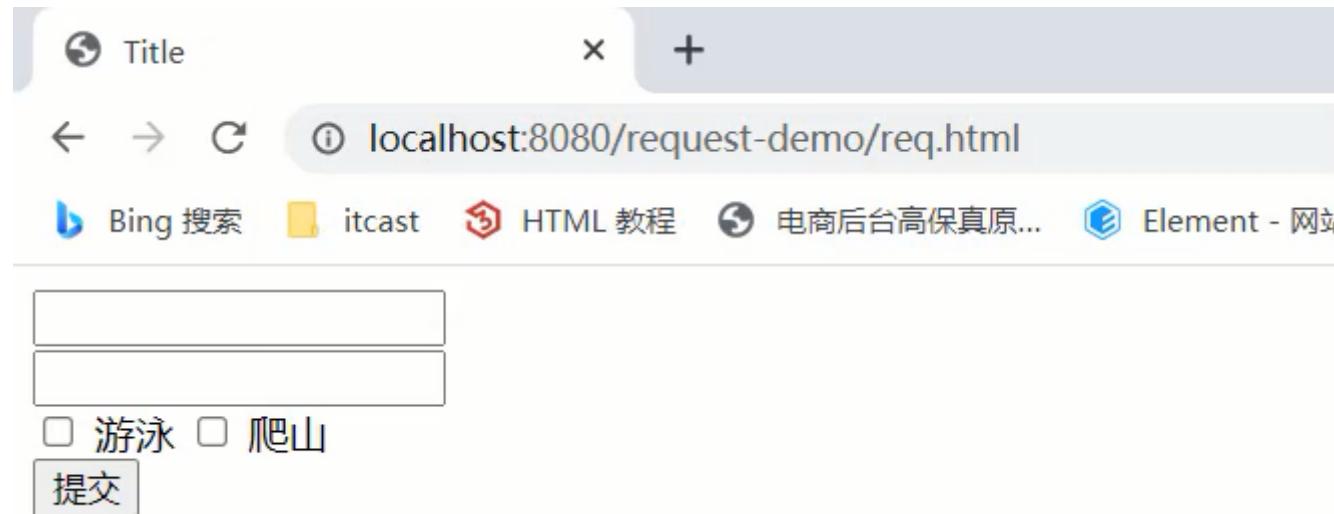
1. 修改req.html页面，添加爱好选项，爱好可以同时选多个

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>Title</title>
6 </head>
7 <body>
```

```

8 <form action="/request-demo/req2" method="get">
9   <input type="text" name="username"><br>
10  <input type="password" name="password"><br>
11  <input type="checkbox" name="hobby" value="1"> 游泳
12  <input type="checkbox" name="hobby" value="2"> 爬山 <br>
13  <input type="submit">
14
15 </form>
16 </body>
17 </html>

```



2. 在Servlet代码中获取页面传递GET请求的参数值

2.1 获取GET方式的所有请求参数

```

1 /**
2  * request 通用方式获取请求参数
3 */
4 @WebServlet("/req2")
5 public class RequestDemo2 extends HttpServlet {
6     @Override
7     protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
8         //GET请求逻辑
9         System.out.println("get....");
10        //1. 获取所有参数的Map集合
11        Map<String, String[]> map = req.getParameterMap();
12        for (String key : map.keySet()) {
13            // username:zhangsan lisi
14            System.out.print(key+":");
15
16            //获取值
17            String[] values = map.get(key);
18            for (String value : values) {
19                System.out.print(value + " ");
20            }
21
22            System.out.println();

```

```
23     }
24 }
25
26     @Override
27     protected void doPost(HttpServletRequest req, HttpServletResponse resp)
28         throws ServletException, IOException {
29 }
```

获取的结果为：

```
get....
username:zhangsan
password:123
hobby:1 2
```

2.2 获取GET请求参数中的爱好，结果是数组值

```
1 /**
2  * request 通用方式获取请求参数
3 */
4 @WebServlet("/req2")
5 public class RequestDemo2 extends HttpServlet {
6     @Override
7     protected void doGet(HttpServletRequest req, HttpServletResponse resp)
8         throws ServletException, IOException {
9         //GET请求逻辑
10        //...
11        System.out.println("-----");
12        String[] hobbies = req.getParameterValues("hobby");
13        for (String hobby : hobbies) {
14            System.out.println(hobby);
15        }
16    }
17    @Override
18    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
19        throws ServletException, IOException {
20 }
```

获取的结果为：

```
-----
1
2
```

2.3 获取GET请求参数中的用户名和密码，结果是单个值

```
1 /**
2  * request 通用方式获取请求参数
3 */
4 @WebServlet("/req2")
5 public class RequestDemo2 extends HttpServlet {
6     @Override
7     protected void doGet(HttpServletRequest req, HttpServletResponse resp)
8     throws ServletException, IOException {
9         //GET请求逻辑
10        //...
11        String username = req.getParameter("username");
12        String password = req.getParameter("password");
13        System.out.println(username);
14        System.out.println(password);
15    }
16    @Override
17    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
18    throws ServletException, IOException {
19 }
```

获取的结果为：

```
zhangsan
123
```

3. 在Servlet代码中获取页面传递POST请求的参数值

3.1 将req.html页面form表单的提交方式改成post

3.2 将doGet方法中的内容复制到doPost方法中即可

小结

- req.getParameter()方法使用的频率会比较高
- 以后我们再写代码的时候，就只需要按照如下格式来编写：

```
1 public class RequestDemo1 extends HttpServlet {  
2     @Override  
3     protected void doGet(HttpServletRequest req, HttpServletResponse resp)  
throws ServletException, IOException {  
4         //采用request提供的获取请求参数的通用方式来获取请求参数  
5         //编写其他的业务代码...  
6     }  
7     @Override  
8     protected void doPost(HttpServletRequest req, HttpServletResponse resp)  
throws ServletException, IOException {  
9         this.doGet(req,resp);  
10    }  
11 }
```

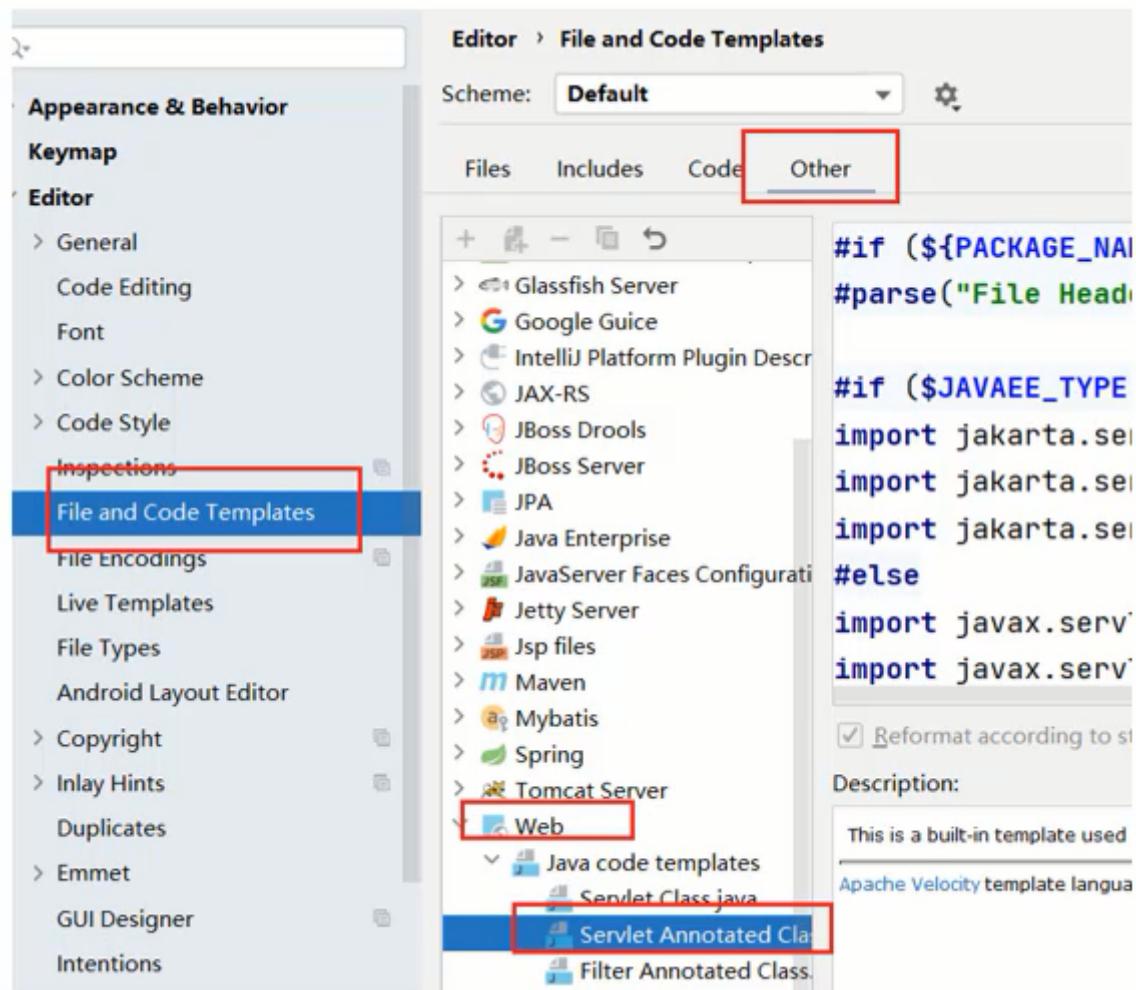
2.3 IDEA快速创建Servlet

使用通用方式获取请求参数后，屏蔽了GET和POST的请求方式代码的不同，则代码可以定义如下格式：

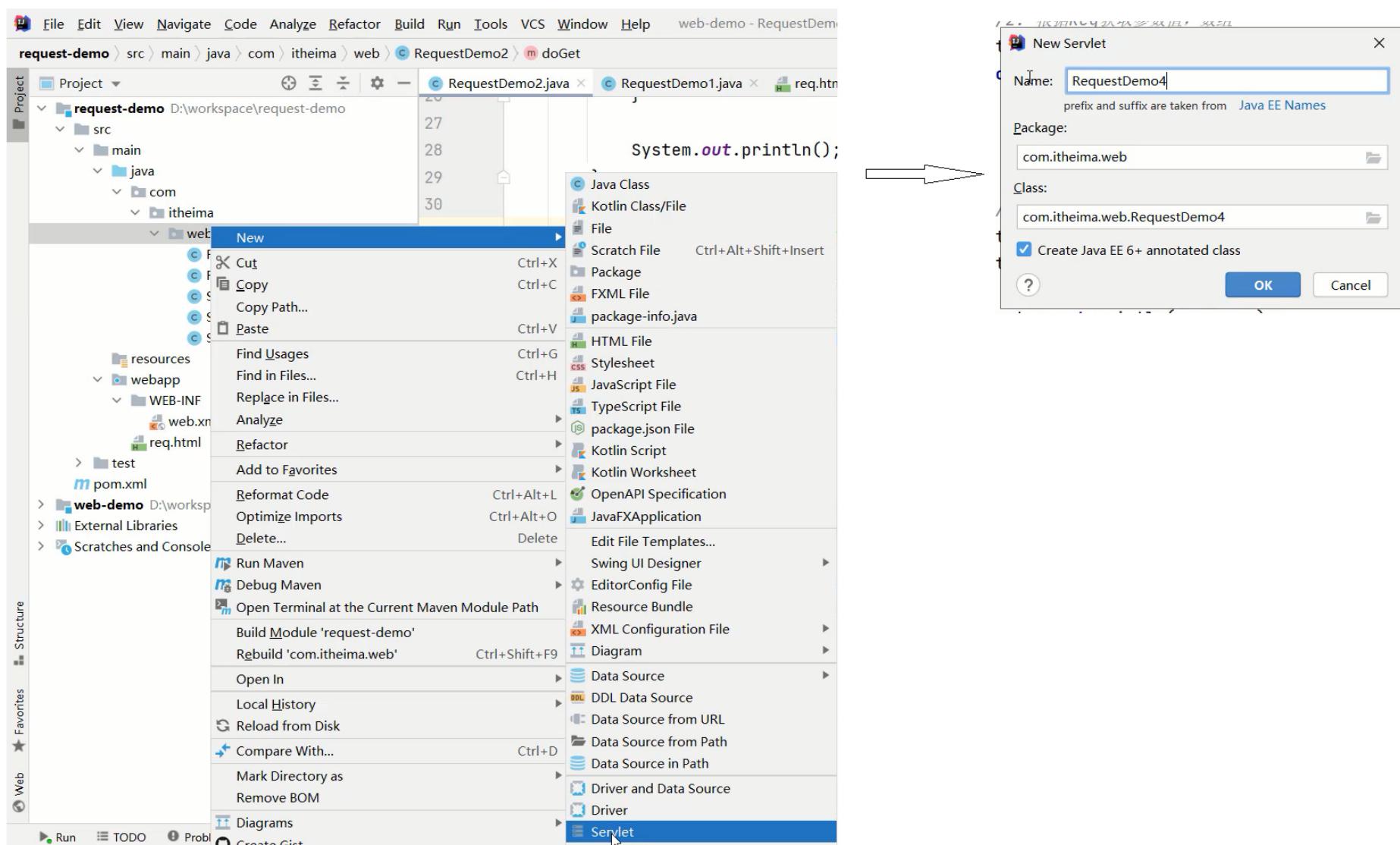
```
@WebServlet("/reqDemo3")  
public class RequestDemo3 extends HttpServlet {  
    @Override  
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) {  
        //  
    }  
    @Override  
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) {  
        this.doGet(req,resp);  
    }  
}
```

由于格式固定，所以我们可以使用IDEA提供的模板来制作一个Servlet的模板，这样我们后期在创建Servlet的时候就会更高效，具体如何实现：

- (1) 按照自己的需求，修改Servlet创建的模板内容



(2) 使用servlet模板创建Servlet类



2.4 请求参数中文乱码问题

问题展示：

(1) 将req.html页面的请求方式修改为get

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
```

```

4   <meta charset="UTF-8">
5   <title>Title</title>
6 </head>
7 <body>
8 <form action="/request-demo/req2" method="get">
9   <input type="text" name="username"><br>
10  <input type="password" name="password"><br>
11  <input type="checkbox" name="hobby" value="1"> 游泳
12  <input type="checkbox" name="hobby" value="2"> 爬山 <br>
13  <input type="submit">
14
15 </form>
16 </body>
17 </html>

```

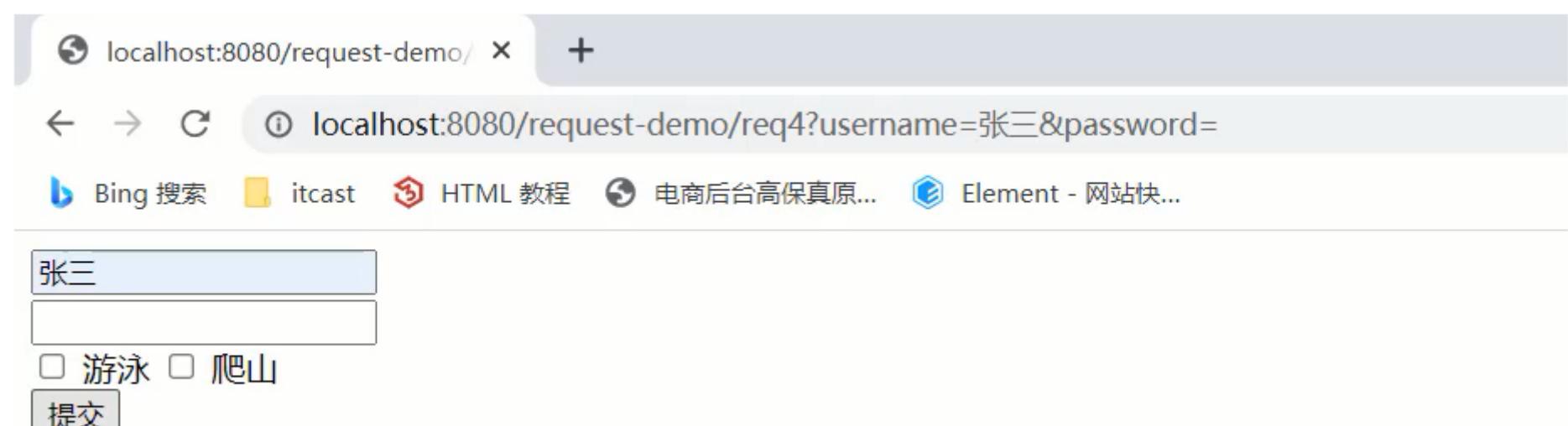
(2) 在Servlet方法中获取参数，并打印

```

1 /**
2  * 中文乱码问题解决方案
3 */
4 @WebServlet("/req4")
5 public class RequestDemo4 extends HttpServlet {
6     @Override
7     protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
8         //1. 获取username
9         String username = request.getParameter("username");
10        System.out.println(username);
11    }
12
13    @Override
14    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
15        this.doGet(request, response);
16    }
17 }

```

(3) 启动服务器，页面上输入中文参数



(4) 查看控制台打印内容

å% ä,

(5) 把req.html页面的请求方式改成post,再次发送请求和中文参数

The screenshot shows a browser window with the URL `localhost:8080/request-demo/req4`. The page has input fields for '张三' (Zhang San) and '地址' (Address). There are two checkboxes for '游泳' (Swimming) and '爬山' (Hiking). A '提交' (Submit) button is at the bottom. The browser tabs show other pages like Bing 搜索 and HTML 教程.

(6) 查看控制台打印内容，依然为乱码

å% ä,

通过上面的案例，会发现，不管是GET还是POST请求，在发送的请求参数中如果有中文，在后台接收的时候，都会出现中文乱码的问题。具体该如何解决呢？

2.4.1 POST请求解决方案

- 分析出现中文乱码的原因：

- POST的请求参数是通过`request.getReader()`来获取流中的数据
- TOMCAT在获取流的时候采用的编码是ISO-8859-1
- ISO-8859-1编码是不支持中文的，所以会出现乱码

- 解决方案：

- 页面设置的编码格式为UTF-8
- 把TOMCAT在获取流数据之前的编码设置为UTF-8
- 通过`request.setCharacterEncoding("UTF-8")`设置编码，UTF-8也可以写成小写

修改后的代码为：

```
1 /**
2  * 中文乱码问题解决方案
3 */
4 @WebServlet("/req4")
5 public class RequestDemo4 extends HttpServlet {
6     @Override
7     protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
8         //1. 解决乱码：POST getReader()
9         //设置字符输入流的编码，设置的字符集要和页面保持一致
10        request.setCharacterEncoding("UTF-8");
11        //2. 获取username
```

```

12     String username = request.getParameter("username");
13     System.out.println(username);
14 }
15
16 @Override
17 protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
18     this.doGet(request, response);
19 }
20 }
```

重新发送POST请求，就会在控制台看到正常展示的中文结果。

至此POST请求中文乱码的问题就已经解决，但是这种方案不适用于GET请求，这个原因是什么呢，咱们下面再分析。

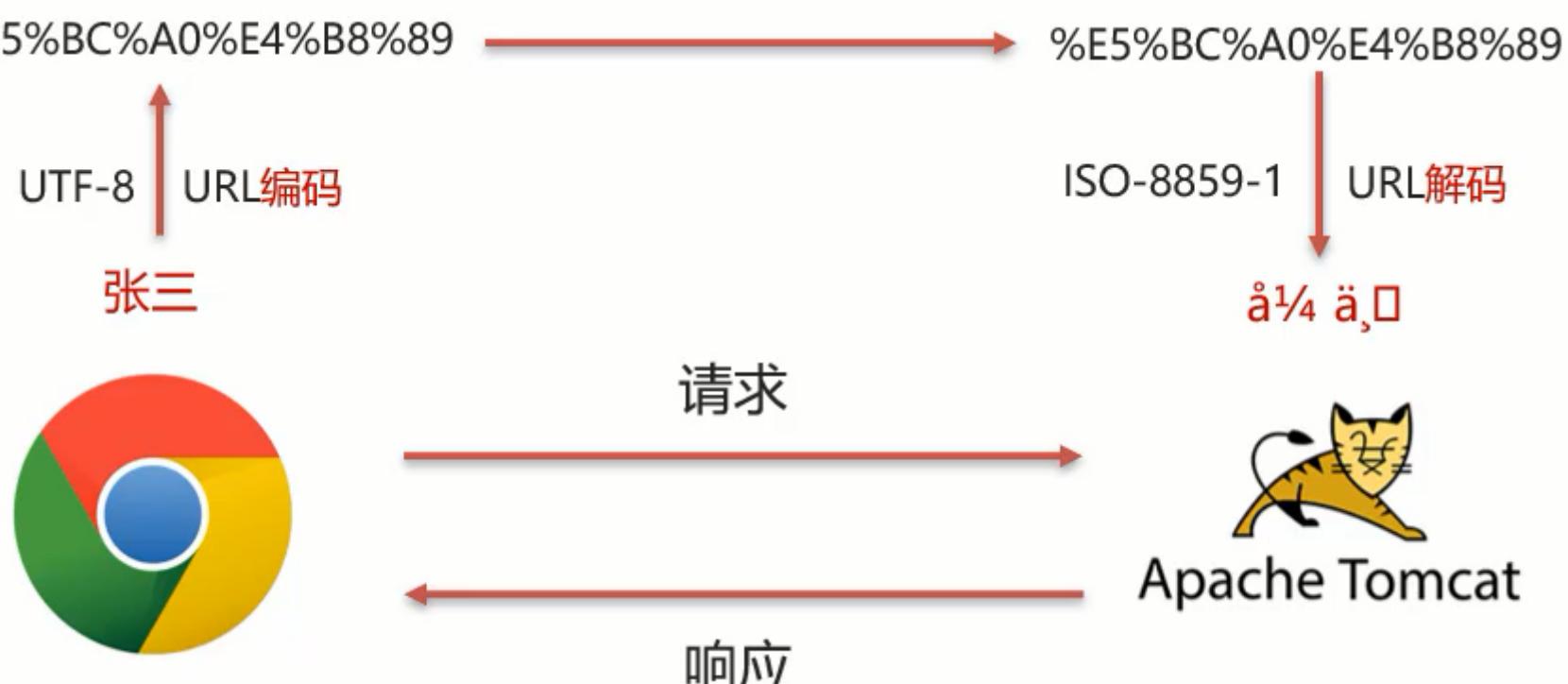
2.4.2 GET请求解决方案

刚才提到一个问题就是POST请求的中文乱码解决方案为什么不适合GET请求？

- GET请求获取请求参数的方式是`request.getQueryString()`
- POST请求获取请求参数的方式是`request.getReader()`
- `request.setCharacterEncoding("utf-8")`是设置`request`处理流的编码
- `getQueryString`方法并没有通过流的方式获取数据

所以GET请求不能用设置编码的方式来解决中文乱码问题，那问题又来了，如何解决GET请求的中文乱码呢？

1. 首先我们需要先分析下GET请求出现乱码的原因：



(1) 浏览器通过HTTP协议发送请求和数据给后台服务器（Tomcat）

(2) 浏览器在发送HTTP的过程中会对中文数据进行URL**编码**

(3) 在进行URL编码的时候会采用页面`<meta>`标签指定的UTF-8的方式进行编码，张三编码后的结果为`%E5%BC%A0%E4%B8%89`

(4) 后台服务器(Tomcat)接收到%E5%BC%A0%E4%B8%89后会默认按照ISO-8859-1进行URL解码

(5) 由于前后编码与解码采用的格式不一样，就会导致后台获取到的数据为乱码。

思考：如果把req.html页面的<meta>标签的charset属性改成ISO-8859-1，后台不做操作，能解决中文乱码问题么？

答案是否定的，因为ISO-8859-1本身是不支持中文展示的，所以改了标签的charset属性后，会导致页面上的中文内容都无法正常展示。

分析完上面的问题后，我们会发现，其中有两个我们不熟悉的内容就是URL编码和URL解码，什么是URL编码，什么又是URL解码呢？

URL编码

这块知识我们只需要了解下即可，具体编码过程分两步，分别是：

(1) 将字符串按照编码方式转为二进制

(2) 每个字节转为2个16进制数并在前边加上%

张三按照UTF-8的方式转换成二进制的结果为：

```
1 1110 0101 1011 1100 1010 0000 1110 0100 1011 1000 1000 1001
```

这个结果是如何计算的？

使用http://www.mytju.com/classcode/tools/encode_utf8.asp，输入张三

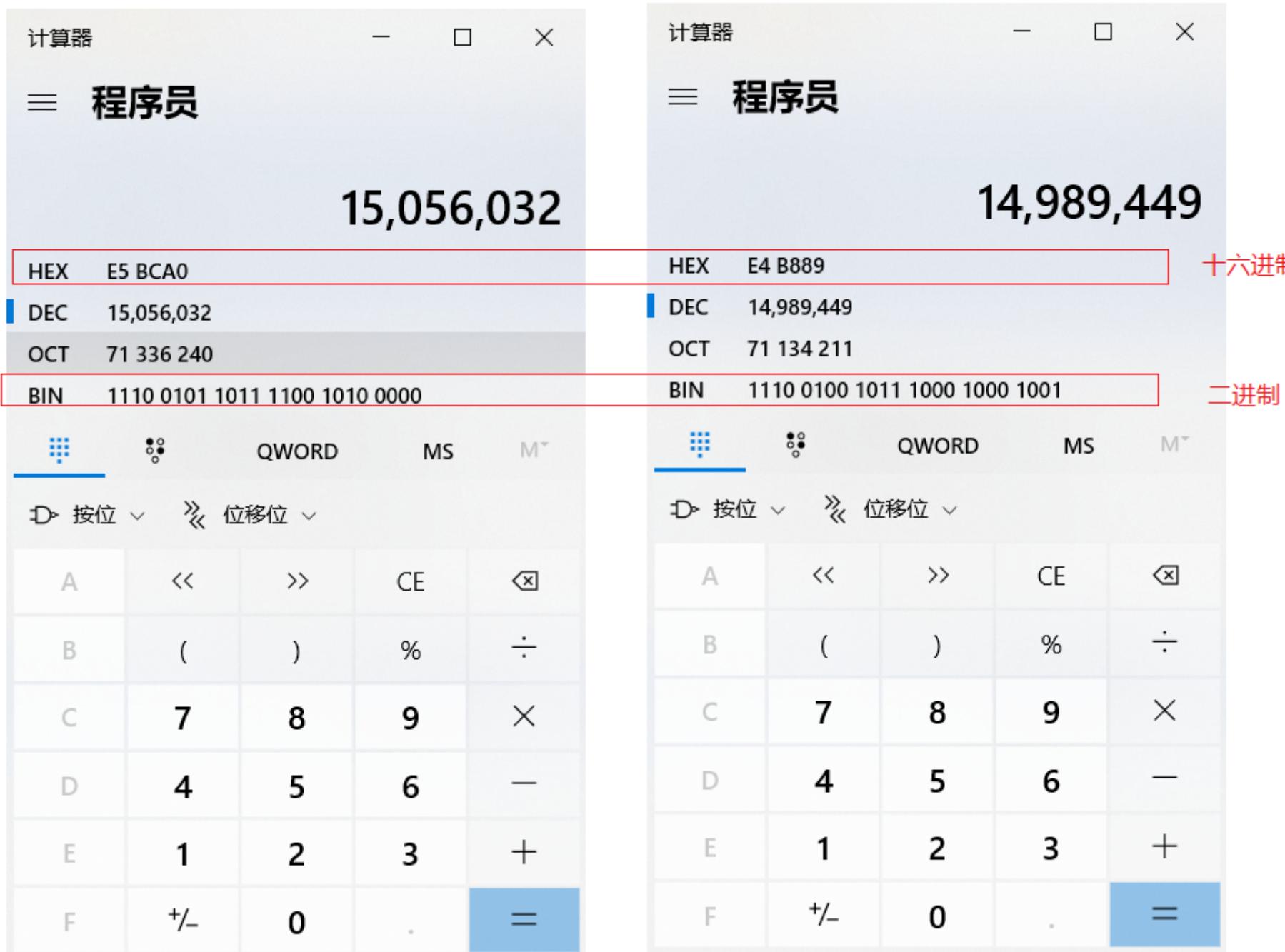
查看字符编码 (UTF-8)

请输入文字 (可以输入多个) :
如：“春眠不觉晓，处处闻啼鸟。”

请输入编码 (只可输入一个) :
如：“&HB4BA”，“65”

字符	编码10进制	编码16进制	Unicode编码10进制	Unicode编码16进制
张	15056032	E5BCA0	24352	5F20
三	14989449	E4B889	19977	4E09

就可以获取张和三分别对应的10进制，然后在使用计算器，选择程序员模式，计算出对应的二进制数据结果：



在计算的十六进制结果中，每两位前面加一个%，就可以获取到%E5%BC%A0%E4%B8%89。

当然你从上面所提供的网站中就已经能看到编码16进制的结果了：

字符		编码10进制	编码16进制	Unicode编码10进制	Unicode编码16进制
张		15056032	E5BCA0	24352	5F20
三		14989449	E4B889	19977	4E09

但是对于上面的计算过程，如果没有工具，纯手工计算的话，相对来说还是比较复杂的，我们也不需要进行手动计算，在Java中已经为我们提供了编码和解码的API工具类可以让我们更快速的进行编码和解码：

编码：

```
1 java.net.URLEncoder.encode("需要被编码的内容", "字符集(UTF-8)")
```

解码：

```
1 java.net.URLDecoder.decode("需要被解码的内容", "字符集(UTF-8)")
```

接下来咱们对张三来进行编码和解码

```
1 public class URLEmo {
```

```

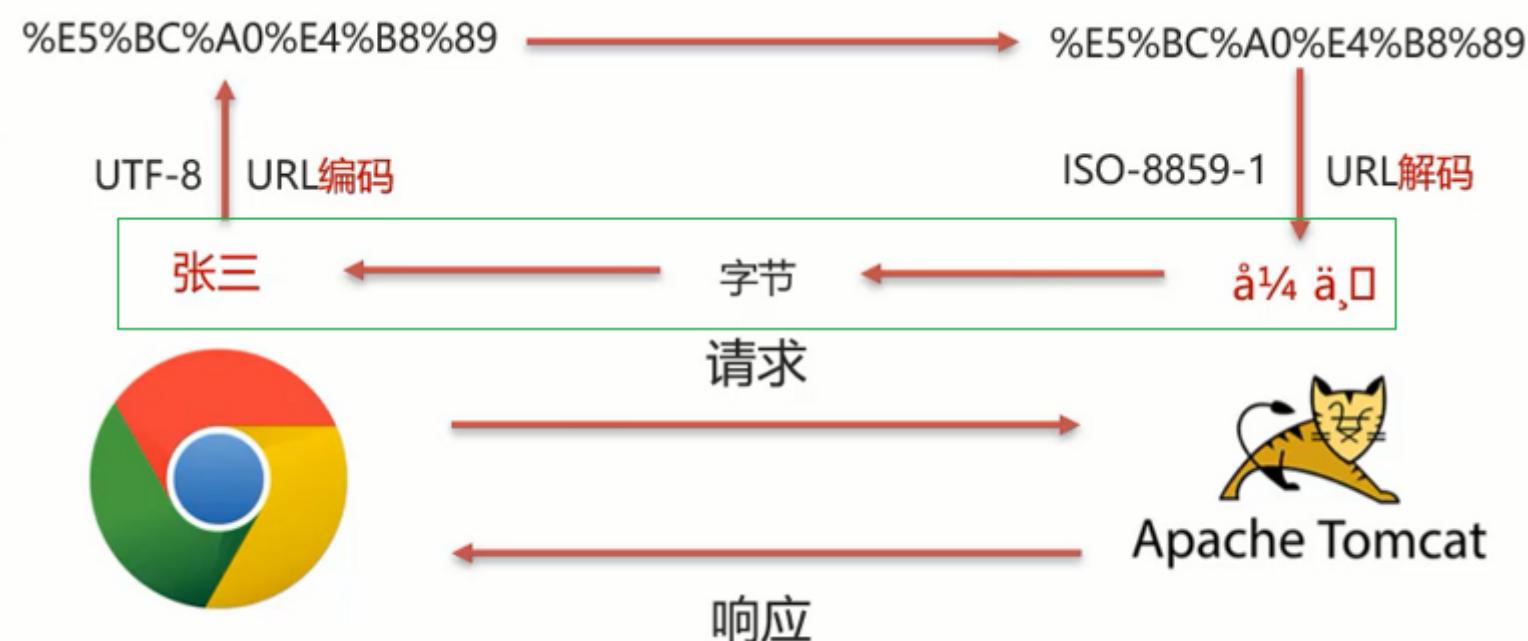
2
3     public static void main(String[] args) throws UnsupportedEncodingException
{
4         String username = "张三";
5         //1. URL编码
6         String encode = URLEncoder.encode(username, "utf-8");
7         System.out.println(encode); //打印:%E5%BC%A0%E4%B8%89
8
9         //2. URL解码
10        //String decode = URLDecoder.decode(encode, "utf-8"); //打印:张三
11        String decode = URLDecoder.decode(encode, "ISO-8859-1"); //打印:`å¼ ä, `
12        System.out.println(decode);
13    }
14 }
15

```

到这，我们就可以分析出GET请求中文参数出现乱码的原因了，

- 浏览器把中文参数按照UTF-8进行URL编码
- Tomcat对获取到的内容进行了ISO-8859-1的URL解码
- 在控制台就会出现类似`å¼ ä, `的乱码，最后一位是个空格

2. 清楚了出现乱码的原因，接下来我们就需要想办法进行解决



从上图可以看住，

- 在进行编码和解码的时候，不管使用的是哪个字符集，他们对应的%E5%BC%A0%E4%B8%89是一致的
- 那他们对应的二进制值也是一样的，为：
 - 1 1110 0101 1011 1100 1010 0000 1110 0100 1011 1000 1000 1001
- 所以我们可以考虑把 å¼ ä, 转换成字节，在把字节转换成张三，在转换的过程中是它们的编码一致，就可以解决中文乱码问题。

具体的实现步骤为：

1. 按照ISO-8859-1编码获取乱码 å¼ ä, 对应的字节数组

2. 按照UTF-8编码获取字节数组对应的字符串

实现代码如下：

```
1 public class URLDemo {  
2  
3     public static void main(String[] args) throws UnsupportedEncodingException {  
4         String username = "张三";  
5         //1. URL编码  
6         String encode = URLEncoder.encode(username, "utf-8");  
7         System.out.println(encode);  
8         //2. URL解码  
9         String decode = URLDecoder.decode(encode, "ISO-8859-1");  
10  
11         System.out.println(decode); //此处打印的是对应的乱码数据  
12  
13         //3. 转换为字节数据, 编码  
14         byte[] bytes = decode.getBytes("ISO-8859-1");  
15         for (byte b : bytes) {  
16             System.out.print(b + " ");  
17         }  
18         //此处打印的是:-27 -68 -96 -28 -72 -119  
19         //4. 将字节数组转为字符串, 解码  
20         String s = new String(bytes, "utf-8");  
21         System.out.println(s); //此处打印的是张三  
22     }  
23 }
```

说明:在第18行中打印的数据是-27 -68 -96 -28 -72 -119和张三转换成的二进制数据1110 0101 1011 1100 1010 0000 1110 0100 1011 1000 1000 1001为什么不一样呢?

其实打印出来的是十进制数据，我们只需要使用计算机换算下就能得到他们的对应关系，如下图：



至此对于GET请求中文乱码的解决方案，我们就已经分析完了，最后在代码中去实现下：

```
1 /**  
2  * 中文乱码问题解决方案  
3  */
```

```

4  @WebServlet("/req4")
5  public class RequestDemo4 extends HttpServlet {
6      @Override
7      protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
8          //1. 解决乱码: POST, getReader()
9          //request.setCharacterEncoding("UTF-8");//设置字符输入流的编码
10
11         //2. 获取username
12         String username = request.getParameter("username");
13         System.out.println("解决乱码前: "+username);
14
15         //3. GET,获取参数的方式: getQueryString
16         // 乱码原因: tomcat进行URL解码, 默认的字符集ISO-8859-1
17         /* //3.1 先对乱码数据进行编码: 转为字节数组
18         byte[] bytes = username.getBytes(StandardCharsets.ISO_8859_1);
19         //3.2 字节数组解码
20         username = new String(bytes, StandardCharsets.UTF_8);*/
21
22         username = new
23         String(username.getBytes(StandardCharsets.ISO_8859_1), StandardCharsets.UTF_8)
24 ;
25
26     }
27
28     @Override
29     protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
30         this.doGet(request, response);
31     }
32 }

```

注意

- 把`request.setCharacterEncoding("UTF-8")`代码注释掉后, 会发现GET请求参数乱码解决方案同时也可也把POST请求参数乱码的问题也解决了
- 只不过对于POST请求参数一般都会比较多, 采用这种方式解决乱码起来比较麻烦, 所以对于POST请求还是建议使用设置编码的方式进行。

另外需要说明一点的是**Tomcat8.0之后, 已将GET请求乱码问题解决, 设置默认的解码方式为UTF-8**

小结

1. 中文乱码解决方案

- POST请求和GET请求的参数中如果有中文, 后台接收数据就会出现中文乱码问题

GET请求在Tomcat8.0以后的版本就不会出现了

- POST请求解决方案是：设置输入流的编码

```
1 request.setCharacterEncoding("UTF-8");  
2 注意：设置的字符集要和页面保持一致
```

- 通用方式（GET/POST）：需要先解码，再编码

```
1 new String(username.getBytes("ISO-8859-1"), "UTF-8");
```

2. URL编码实现方式：

- 编码：

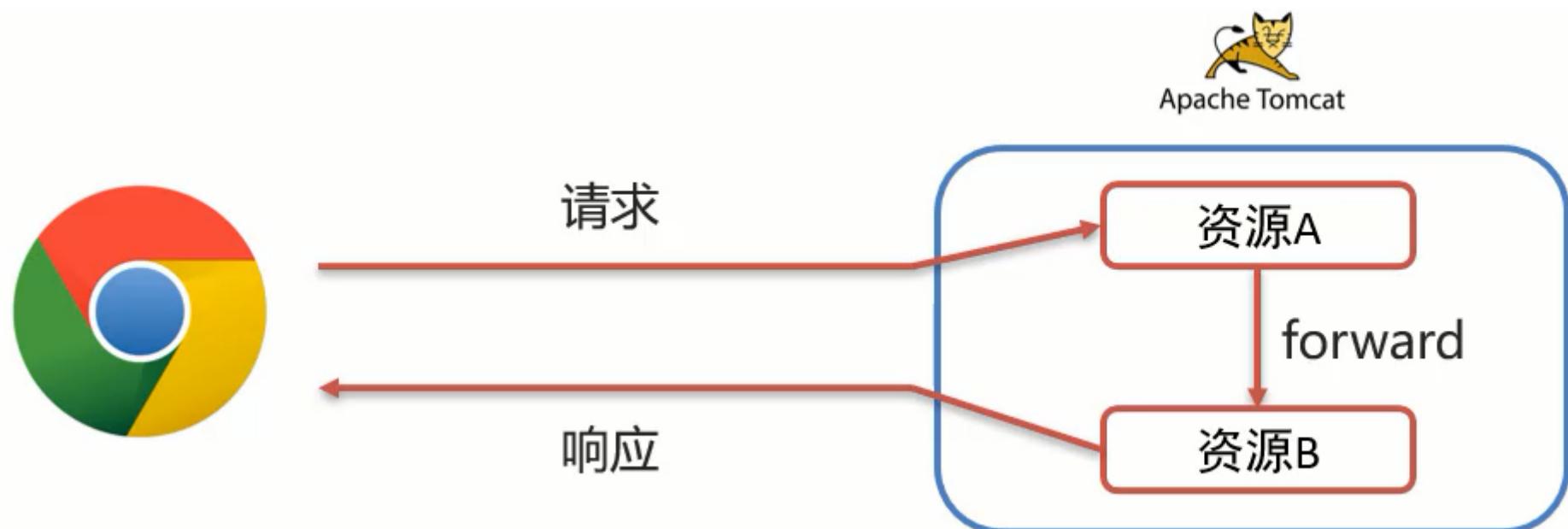
```
1 URLEncoder.encode(str, "UTF-8");
```

- 解码：

```
1 URLDecoder.decode(s, "ISO-8859-1");
```

2.5 Request请求转发

1. 请求转发（forward）：一种在服务器内部的资源跳转方式。



(1) 浏览器发送请求给服务器，服务器中对应的资源A接收到请求

(2) 资源A处理完请求后将请求发给资源B

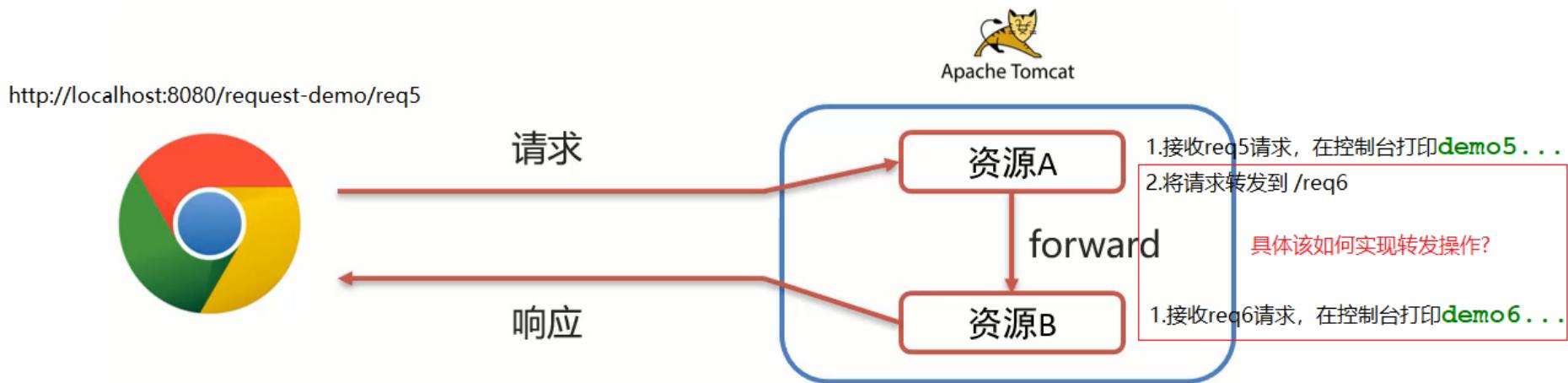
(3) 资源B处理完后将结果响应给浏览器

(4) 请求从资源A到资源B的过程就叫**请求转发**

2. 请求转发的实现方式：

```
1 req.getRequestDispatcher("资源B路径").forward(req, resp);
```

具体如何来使用，我们先来看下需求：



针对上述需求，具体的实现步骤为：

1. 创建一个 RequestDemo5 类，接收 /req5 的请求，在 doGet 方法中打印 demo5
2. 创建一个 RequestDemo6 类，接收 /req6 的请求，在 doGet 方法中打印 demo6
3. 在 RequestDemo5 的方法中使用
req.getRequestDispatcher("/req6").forward(req, resp) 进行请求转发
4. 启动测试

(1) 创建 RequestDemo5 类

```

1 /**
2  * 请求转发
3 */
4 @WebServlet("/req5")
5 public class RequestDemo5 extends HttpServlet {
6     @Override
7     protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
8         System.out.println("demo5...");
9     }
10
11    @Override
12    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
13        this.doGet(request, response);
14    }
15 }
```

(2) 创建 RequestDemo6 类

```

1 /**
2  * 请求转发
3 */
4 @WebServlet("/req6")
5 public class RequestDemo6 extends HttpServlet {
6     @Override
```

```

7     protected void doGet(HttpServletRequest request, HttpServletResponse
8         response) throws ServletException, IOException {
9             System.out.println("demo6..."); 
10        }
11    @Override
12    protected void doPost(HttpServletRequest request, HttpServletResponse
13        response) throws ServletException, IOException {
14        this.doGet(request, response);
15    }

```

(3) 在 RequestDemo5 的 doGet 方法中进行请求转发

```

1 /**
2  * 请求转发
3 */
4 @WebServlet("/req5")
5 public class RequestDemo5 extends HttpServlet {
6     @Override
7     protected void doGet(HttpServletRequest request, HttpServletResponse
8         response) throws ServletException, IOException {
9         System.out.println("demo5..."); 
10        //请求转发
11        request.getRequestDispatcher("/req6").forward(request, response);
12    }
13
14    @Override
15    protected void doPost(HttpServletRequest request, HttpServletResponse
16        response) throws ServletException, IOException {
17        this.doGet(request, response);
18    }
19 }

```

(4) 启动测试

访问 `http://localhost:8080/request-demo/req5`, 就可以在控制台看到如下内容:

```

demo5...
demo6...

```

说明请求已经转发到了 /req6

3. 请求转发资源间共享数据: 使用 Request 对象

此处主要解决的问题是把请求从 /req5 转发到 /req6 的时候, 如何传递数据给 /req6。

需要使用 request 对象提供的三个方法:

- 存储数据到 request 域 [范围, 数据是存储在 request 对象] 中

```
1 void setAttribute(String name, Object o);
```

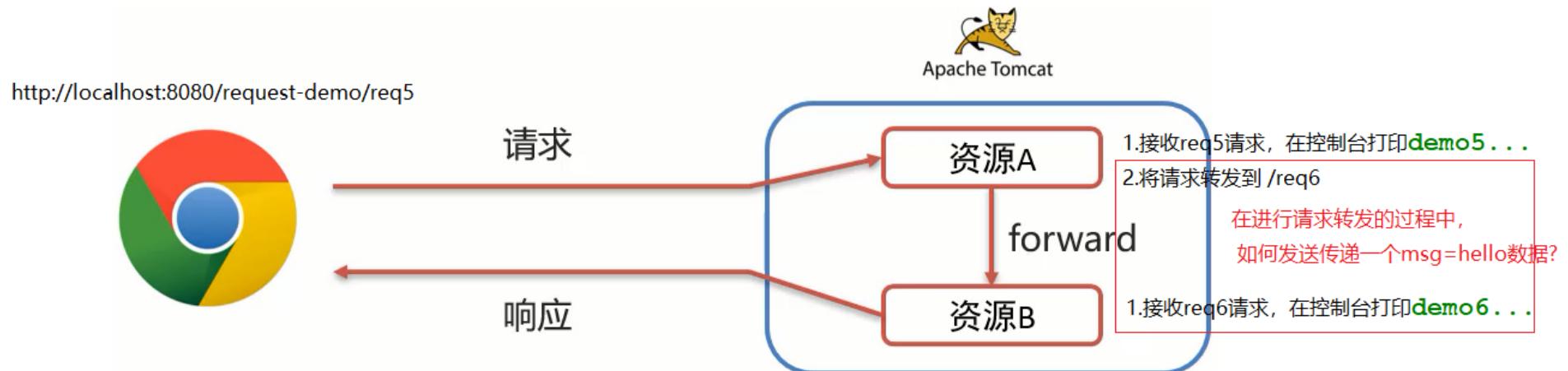
- 根据key获取值

```
1 Object getAttribute(String name);
```

- 根据key删除该键值对

```
1 void removeAttribute(String name);
```

接着上个需求来：



1. 在RequestDemo5的doGet方法中转发请求之前，将数据存入request域对象中

2. 在RequestDemo6的doGet方法从request域对象中获取数据，并将数据打印到控制台

3. 启动访问测试

(1) 修改RequestDemo5中的方法

```
1 @WebServlet("/req5")
2 public class RequestDemo5 extends HttpServlet {
3     @Override
4     protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
5         System.out.println("demo5...");
6         //存储数据
7         request.setAttribute("msg", "hello");
8         //请求转发
9         request.getRequestDispatcher("/req6").forward(request, response);
10    }
11
12
13    @Override
14    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
15        this.doGet(request, response);
16    }
17 }
```

(2) 修改RequestDemo6中的方法

```

1 /**
2  * 请求转发
3 */
4 @WebServlet("/req6")
5 public class RequestDemo6 extends HttpServlet {
6     @Override
7     protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
8         System.out.println("demo6...");
9         //获取数据
10        Object msg = request.getAttribute("msg");
11        System.out.println(msg);
12    }
13
14    @Override
15    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
16        this.doGet(request, response);
17    }
18 }
19

```

(3) 启动测试

访问 `http://localhost:8080/request-demo/req5`, 就可以在控制台看到如下内容:

```

demo5...
demo6...
hello

```

此时就可以实现在转发多个资源之间共享数据。

4. 请求转发的特点

- 浏览器地址栏路径不发生变化

虽然后台从 `/req5` 转发到 `/req6`, 但是浏览器的地址一直是 `/req5`, 未发生变化



- 只能转发到当前服务器的内部资源

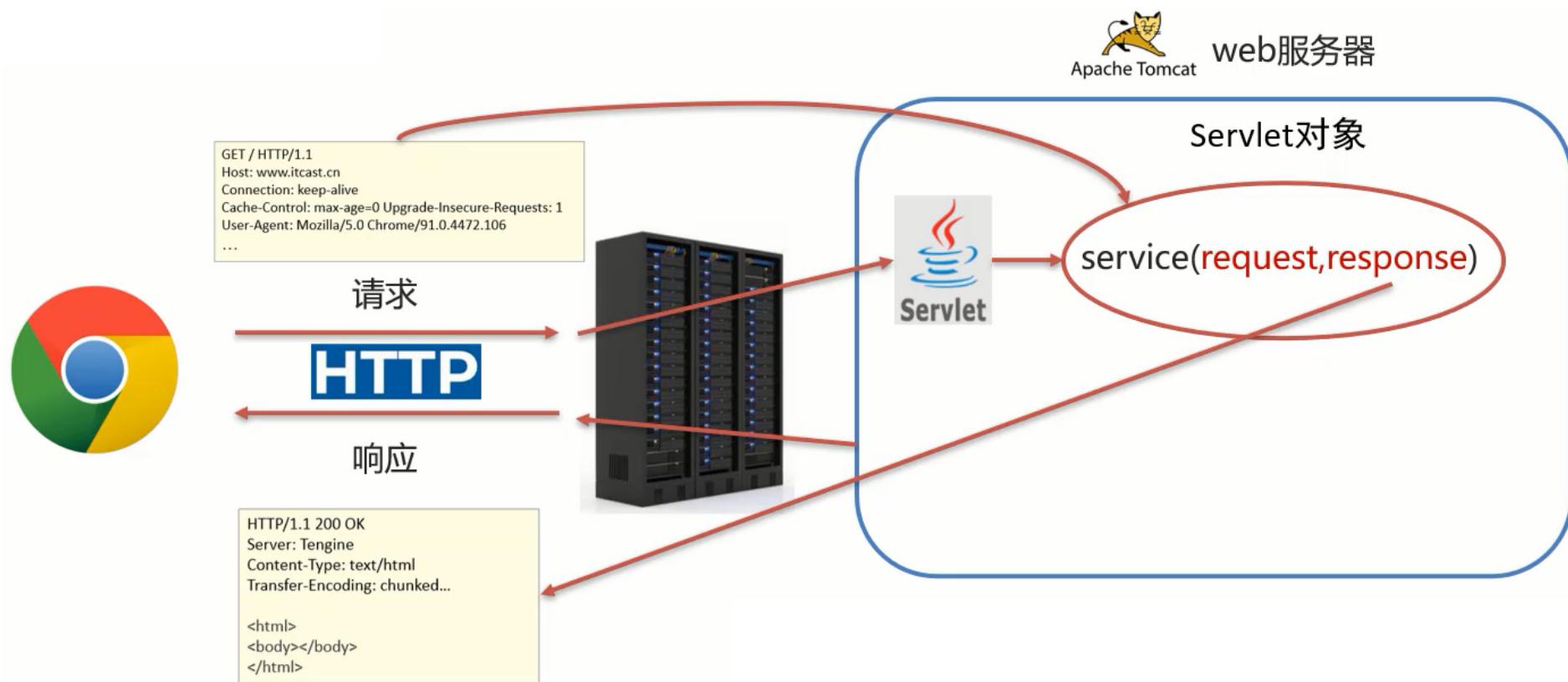
不能从一个服务器通过转发访问另一台服务器

- 一次请求, 可以在转发资源间使用 `request` 共享数据

虽然后台从 `/req5` 转发到 `/req6`, 但是这个 **只有一次请求**

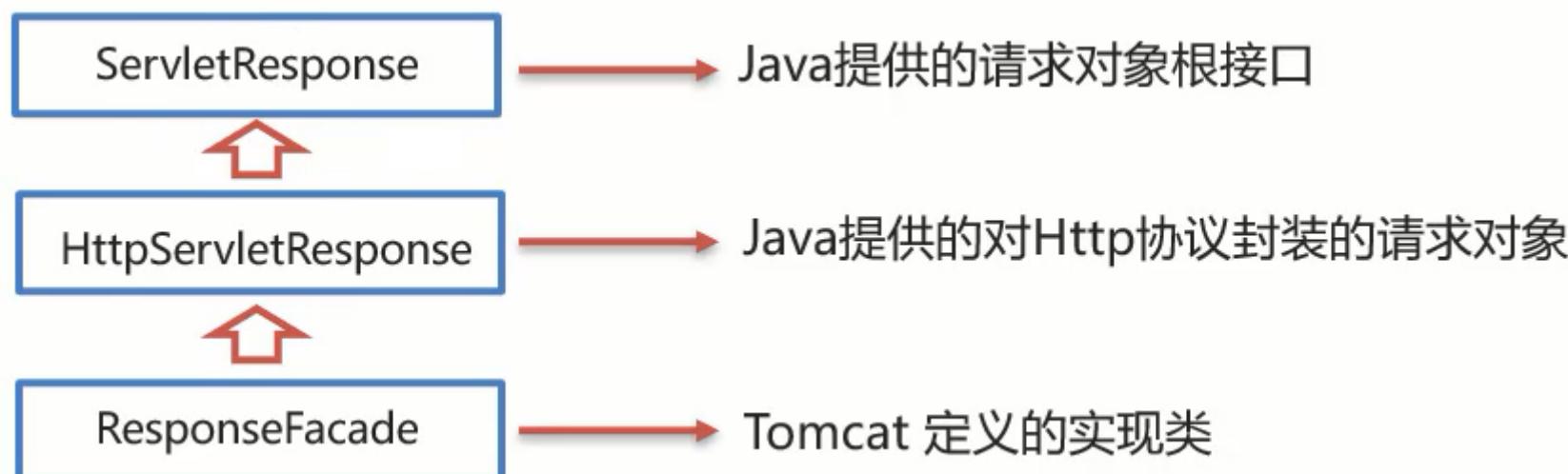
3, Response对象

前面讲解完Request对象，接下来我们回到刚开始的那张图：



- Request: 使用request对象来**获取**请求数据
- Response: 使用response对象来**设置**响应数据

Reponse的继承体系和Request的继承体系也非常相似：



介绍完Response的相关体系结构后，接下来对于Response我们需要学习如下内容：

- Response设置响应数据的功能介绍
- Response完成重定向
- Response响应字符数据
- Response响应字节数据

3.1 Response设置响应数据功能介绍

HTTP响应数据总共分为三部分内容，分别是**响应行、响应头、响应体**，对于这三部分内容的数据，responce对象都提供了哪些方法来进行设置？

1. 响应行



对于响应头，比较常用的就是设置响应状态码：

```
1 void setStatus(int sc);
```

2. 响应头

Content-Type: text/html

键 值

设置响应头键值对：

```
1 void setHeader(String name, String value);
```

3. 响应体

<html><head>head><body></body></html>

对于响应体，是通过字符、字节输出流的方式往浏览器写，

获取字符输出流：

```
1 PrintWriter getWriter();
```

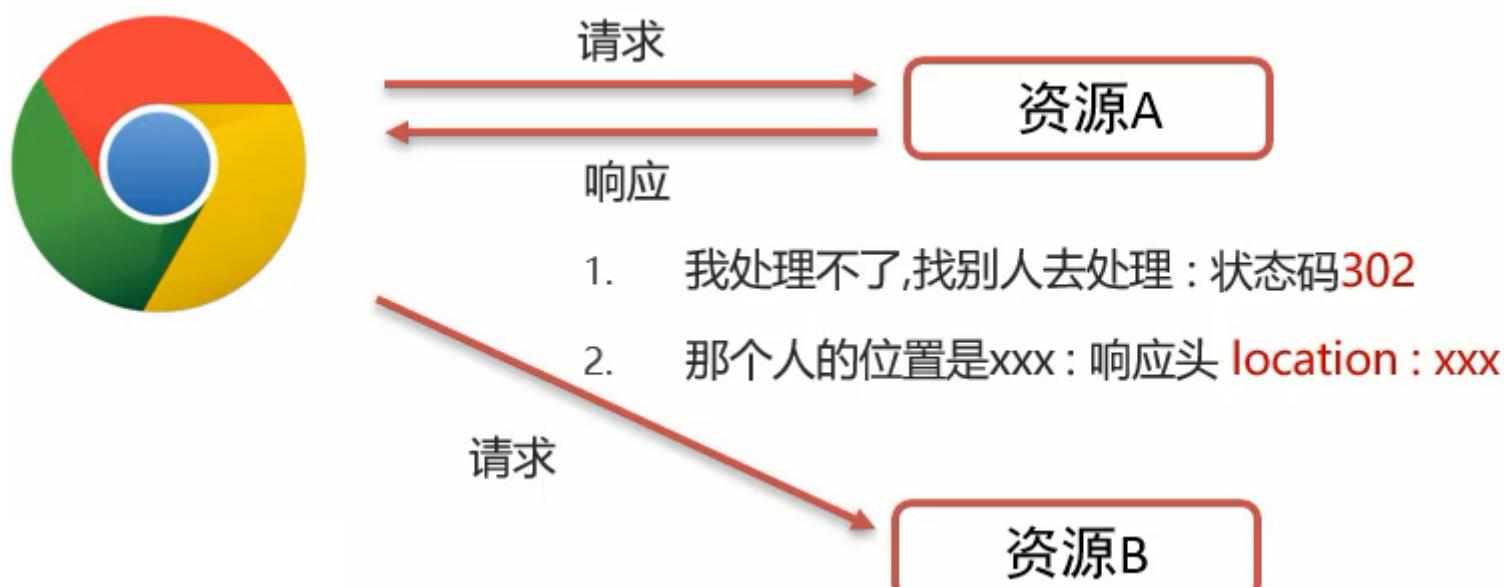
获取字节输出流

```
1 ServletOutputStream getOutputStream();
```

介绍完这些方法后，后面我们会通过案例把这些方法都用一用，首先先来完成下重定向的功能开发。

3.2 Response请求重定向

1. Response重定向 (redirect) : 一种资源跳转方式。



(1) 浏览器发送请求给服务器，服务器中对应的资源A接收到请求

(2) 资源A现在无法处理该请求，就会给浏览器响应一个302的状态码+location的一个访问资源B的路径

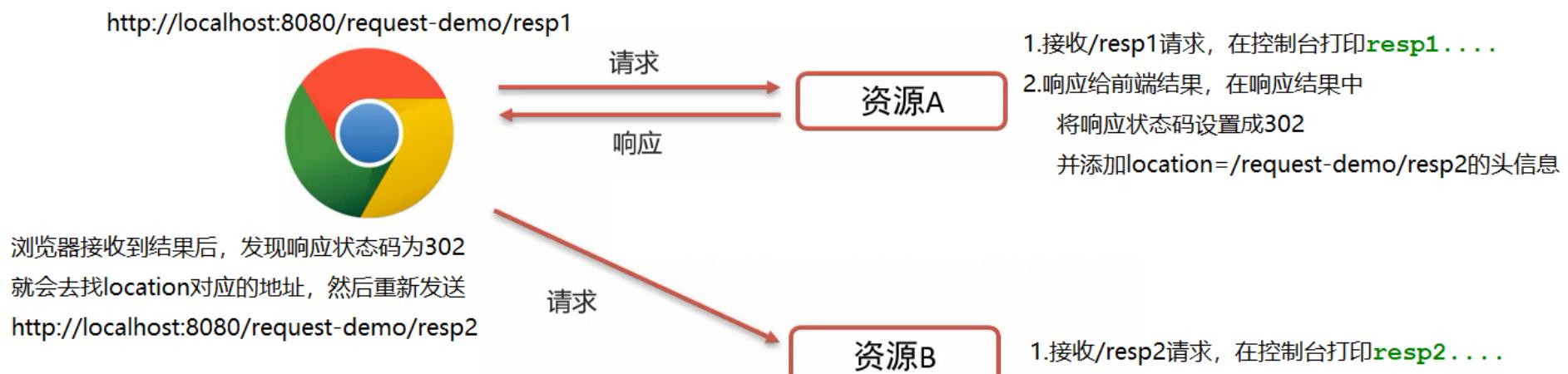
(3) 浏览器接收到响应状态码为302就会重新发送请求到location对应的访问地址去访问资源B

(4) 资源B接收到请求后进行处理并最终给浏览器响应结果，这整个过程就叫**重定向**

2. 重定向的实现方式：

```
1 resp.setStatus(302);  
2 resp.setHeader("Location", "资源B的访问路径");
```

具体如何来使用，我们先来看下需求：



针对上述需求，具体的实现步骤为：

1. 创建一个ResponseDemo1类，接收/resp1的请求，在doGet方法中打印resp1....

2. 创建一个ResponseDemo2类，接收/resp2的请求，在doGet方法中打印resp2....

3. 在ResponseDemo1的方法中使用

```
response.setStatus(302);
```

```
response.setHeader("Location", "/request-demo/resp2") 来给前端响应结果数据
```

4. 启动测试

(1) 创建ResponseDemo1类

```
1 @WebServlet("/resp1")
2 public class ResponseDemo1 extends HttpServlet {
3     @Override
4         protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
5             System.out.println("resp1....");
6         }
7
8     @Override
9         protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
10            this.doGet(request, response);
11        }
12 }
```

(2) 创建 ResponseDemo2 类

```
1 @WebServlet("/resp2")
2 public class ResponseDemo2 extends HttpServlet {
3     @Override
4         protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
5             System.out.println("resp2....");
6         }
7
8     @Override
9         protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
10            this.doGet(request, response);
11        }
12 }
```

(3) 在 ResponseDemo1 的 doGet 方法中给前端响应数据

```
1 @WebServlet("/resp1")
2 public class ResponseDemo1 extends HttpServlet {
3     @Override
4         protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
5             System.out.println("resp1....");
6             // 重定向
7             // 1. 设置响应状态码 302
8             response.setStatus(302);
9             // 2. 设置响应头 Location
10            response.setHeader("Location", "/request-demo/resp2");
11        }
12
13     @Override
```

```

14     protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
15         this.doGet(request, response);
16     }
17 }
```

(4) 启动测试

访问 `http://localhost:8080/request-demo/resp1`, 就可以在控制台看到如下内容:

```

| resp1....
| resp2....
```

说明 `/resp1` 和 `/resp2` 都被访问到了。到这重定向就已经完成了。

虽然功能已经实现，但是从设置重定向的两行代码来看，会发现除了重定向的地址不一样，其他的内容都是一模一样，所以 `request` 对象给我们提供了简化的编写方式为：

```
1 response.sendRedirect("/request-demo/resp2")
```

所以第3步中的代码就可以简化为：

```

1 @WebServlet("/resp1")
2 public class ResponseDemo1 extends HttpServlet {
3     @Override
4     protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
5         System.out.println("resp1....");
6         //重定向
7         response.sendRedirect("/request-demo/resp2");
8     }
9
10    @Override
11    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
12        this.doGet(request, response);
13    }
14 }
```

3. 重定向的特点

- 浏览器地址栏路径发生变化

当进行重定向访问的时候，由于是由浏览器发送的两次请求，所以地址会发生变化



- 可以重定向到任何位置的资源(服务内容、外部均可)

因为第一次响应结果中包含了浏览器下次要跳转的路径，所以这个路径是可以任意位置资源。

- 两次请求，不能在多个资源使用request共享数据

因为浏览器发送了两次请求，是两个不同的request对象，就无法通过request对象进行共享数据

介绍完**请求重定向**和**请求转发**以后，接下来需要把这两个放在一块对比下：

● 重定向特点：	● 请求转发特点：
➤ 浏览器地址栏路径发生变化	➤ 浏览器地址栏路径不发生变化
➤ 可以重定向到任意位置的资源（服务器内部、外部均可）	➤ 只能转发到当前服务器的内部资源
➤ 两次请求，不能在多个资源使用request共享数据	➤ 一次请求，可以在转发的资源间使用request共享数据

以后到底用哪个，还是需要根据具体的业务来决定。

3.3 路径问题

- 问题1：转发的时候路径上没有加/request-demo而重定向加了，那么到底什么时候需要加，什么时候不需要加呢？

转发: `request.getRequestDispatcher(path: "/req6").forward(request, response);`

重定向: `response.setHeader(name: "Location", value: "/request-demo/resp2");`

其实判断的依据很简单，只需要记住下面的规则即可：

- 浏览器使用：需要加虚拟目录（项目访问路径）
- 服务端使用：不需要加虚拟目录

对于转发来说，因为是在服务端进行的，所以不需要加虚拟目录

对于重定向来说，路径最终是由浏览器来发送请求，就需要添加虚拟目录。

掌握了这个规则，接下来就通过一些练习来强化下知识的学习：

- ``
- `<form action='路径'>`
- `req.getRequestDispatcher("路径")`
- `resp.sendRedirect("路径")`

答案：

1. 超链接，从浏览器发送，需要加
2. 表单，从浏览器发送，需要加
3. 转发，是从服务器内部跳转，不需要加
4. 重定向，是由浏览器进行跳转，需要加。

- 问题2：在重定向的代码中，/request-demo是固定编码的，如果后期通过Tomcat插件配置了项目的访问路径，那么所有需要重定向的地方都需要重新修改，该如何优化？

```

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.tomcat.maven</groupId>
            <artifactId>tomcat7-maven-plugin</artifactId>
            <version>2.2</version>
            <!-- <configuration> 配置项目的访问地址
                <path>/xxx</path>
            </configuration>-->
        </plugin>
    </plugins>
</build>

```

答案也比较简单，我们可以在代码中动态去获取项目访问的虚拟目录，具体如何获取，我们可以借助前面咱们所学习的request对象中的getContextPath()方法，修改后的代码如下：

```

1 @WebServlet("/resp1")
2 public class ResponseDemo1 extends HttpServlet {
3     @Override
4     protected void doGet(HttpServletRequest request, HttpServletResponse
5 response) throws ServletException, IOException {
6         System.out.println("resp1....");
7
8         //简化方式完成重定向
9         //动态获取虚拟目录
10        String contextPath = request.getContextPath();
11        response.sendRedirect(contextPath+"/resp2");
12    }
13
14    @Override
15    protected void doPost(HttpServletRequest request, HttpServletResponse
16 response) throws ServletException, IOException {
17        this.doGet(request, response);
18    }
19 }

```

重新启动访问测试，功能依然能够实现，此时就可以动态获取项目访问的虚拟路径，从而降低代码的耦合度。

3.4 Response响应字符数据

要想将字符数据写回到浏览器，我们需要两个步骤：

- 通过Response对象获取字符输出流： PrintWriter writer = resp.getWriter();
- 通过字符输出流写数据： writer.write("aaa");

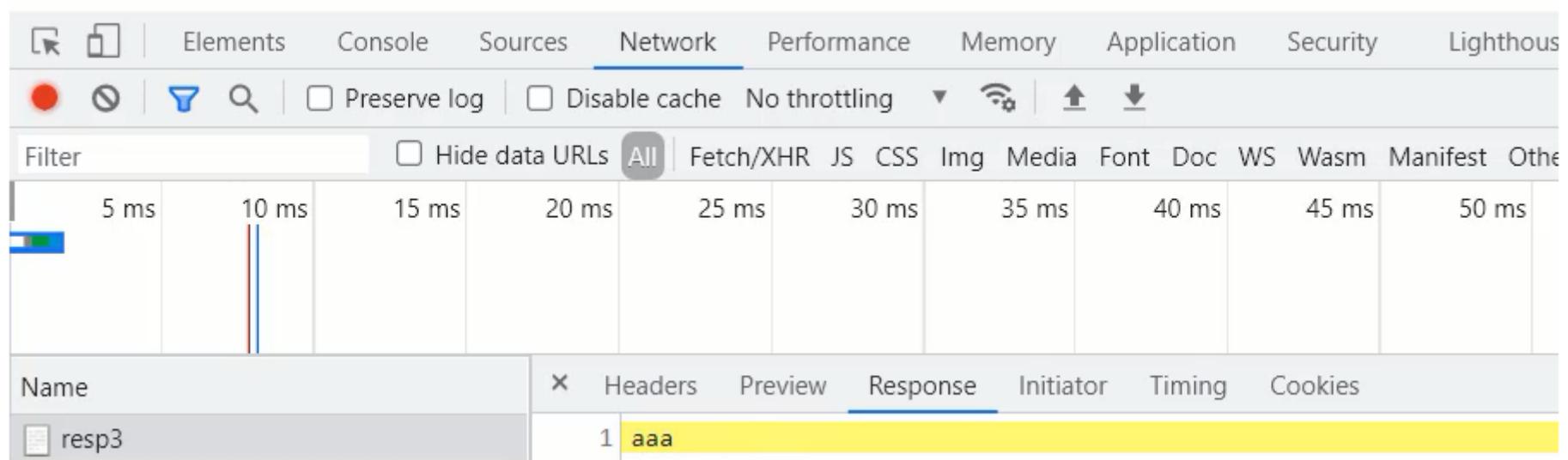
接下来，我们实现通过些案例把响应字符数据给实际应用下：

1. 返回一个简单的字符串aaa

```
1 /**
2  * 响应字符数据：设置字符数据的响应体
3 */
4 @WebServlet("/resp3")
5 public class ResponseDemo3 extends HttpServlet {
6     @Override
7     protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
8         response.setContentType("text/html;charset=utf-8");
9         //1. 获取字符输出流
10        PrintWriter writer = response.getWriter();
11        writer.write("aaa");
12    }
13    @Override
14    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
15        this.doGet(request, response);
16    }
17 }
```

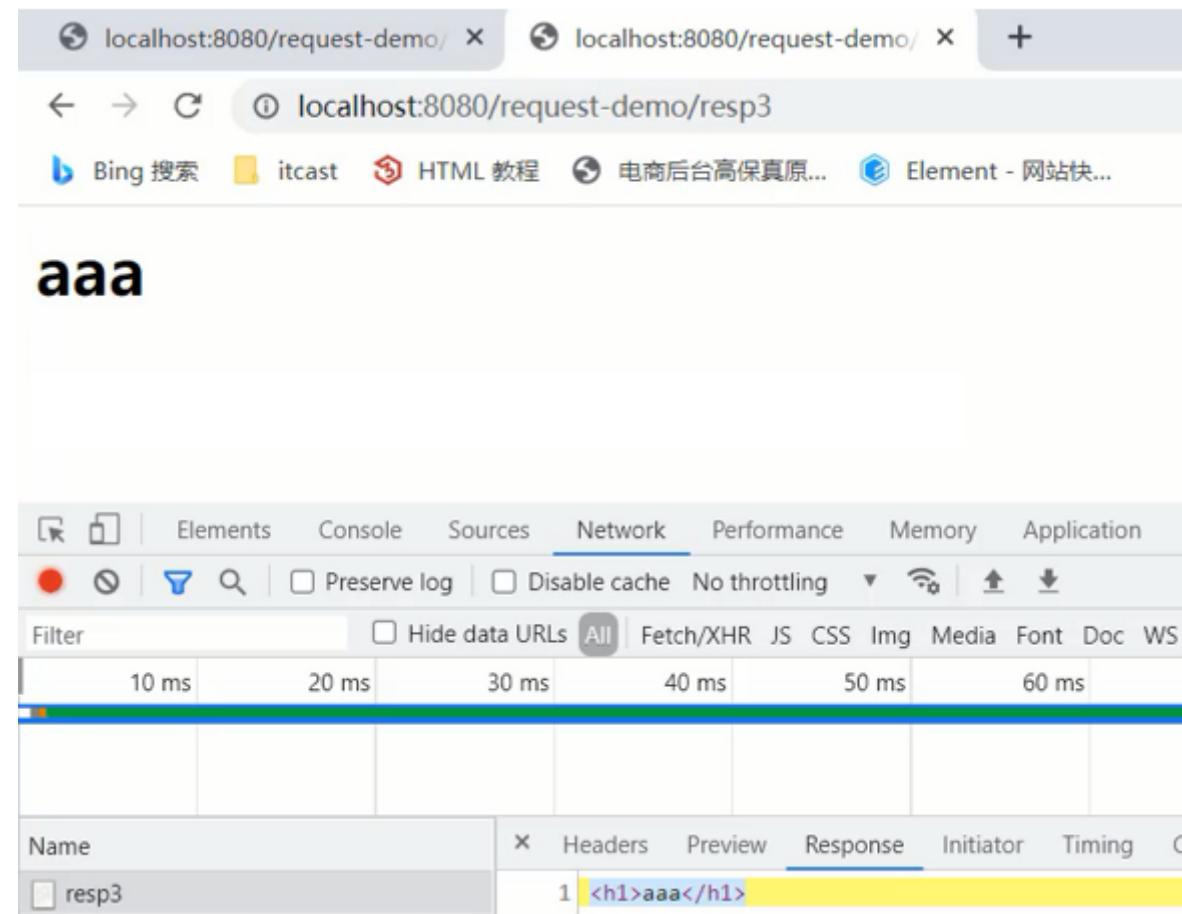


aaa



2. 返回一串html字符串，并且能被浏览器解析

```
1 PrintWriter writer = response.getWriter();
2 //content-type, 告诉浏览器返回的数据类型是HTML类型数据, 这样浏览器才会解析HTML标签
3 response.setHeader("content-type", "text/html");
4 writer.write("<h1>aaa</h1>");
```



注意:一次请求响应结束后, response对象就会被销毁掉, 所以不要手动关闭流。

3. 返回一个中文的字符串你好, 需要注意设置响应数据的编码为 utf-8

```
1 //设置响应的数据格式及数据的编码
2 response.setContentType("text/html;charset=utf-8");
3 writer.write("你好");
```



3.3 Response响应字节数据

要想将字节数据写回到浏览器, 我们需要两个步骤:

- 通过Response对象获取字节输出流: ServletOutputStream outputStream = resp.getOutputStream();
- 通过字节输出流写数据: outputStream.write(字节数据);

接下来, 我们实现通过些案例把响应字符数据给实际应用下:

1. 返回一个图片文件到浏览器

```
1 /**
2 * 响应字节数据: 设置字节数据的响应体
```

```
3  */
4 @webServlet("/resp4")
5 public class ResponseDemo4 extends HttpServlet {
6     @Override
7     protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
8         //1. 读取文件
9         FileInputStream fis = new FileInputStream("d://a.jpg");
10        //2. 获取response字节输出流
11        ServletOutputStream os = response.getOutputStream();
12        //3. 完成流的copy
13        byte[] buff = new byte[1024];
14        int len = 0;
15        while ((len = fis.read(buff)) != -1){
16            os.write(buff, 0, len);
17        }
18        fis.close();
19    }
20
21    @Override
22    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
23        this.doGet(request, response);
24    }
25 }
```

resp4 (420×652) × + ← → C ① localhost:8080/request-demo/resp4

Bing 搜索 itcast HTML 教程 电商后台高保真原... Element - 网站快...

		2021東京奧運會	金	銀	銅	總計
1	中国	38	32	18	88	
	中国香港	1	2	3	6	
	中国台北	2	4	6	12	
2	美国	39	41	33	113	
3	日本	27	14	17	58	
4	英國	22	21	22	65	
5	ROC	20	28	23	71	
6	澳大利亚	17	7	22	46	
7	荷兰	10	12	14	36	
8	法国	10	12	11	33	
9	德国	10	11	16	37	
10	意大利	10	10	20	40	

上述代码中，对于流的copy的代码还是比较复杂的，所以我们可以使用别人提供好的方法来简化代码的开发，具体的步骤是：

(1) pom.xml添加依赖

```

1 <dependency>
2   <groupId>commons-io</groupId>
3   <artifactId>commons-io</artifactId>
4   <version>2.6</version>
5 </dependency>
```

(2) 调用工具类方法

```

1 //fis:输入流
2 //os:输出流
3 IOUtils.copy(fis,os);
```

优化后的代码：

```

1 /**
2  * 响应字节数据：设置字节数据的响应体
3 */
```

```

4  @WebServlet("/resp4")
5  public class ResponseDemo4 extends HttpServlet {
6      @Override
7      protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
8          //1. 读取文件
9          FileInputStream fis = new FileInputStream("d://a.jpg");
10         //2. 获取response字节输出流
11         ServletOutputStream os = response.getOutputStream();
12         //3. 完成流的copy
13         IOutils.copy(fis,os);
14         fis.close();
15     }
16
17     @Override
18     protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
19         this.doGet(request, response);
20     }
21 }

```

4. 用户注册登录案例

接下来我们通过两个比较常见的案例，一个是**注册**，一个是**登录**来对今天学习的内容进行一个实战演练，首先来实现用户登录。

4.1 用户登录

4.1.1 需求分析

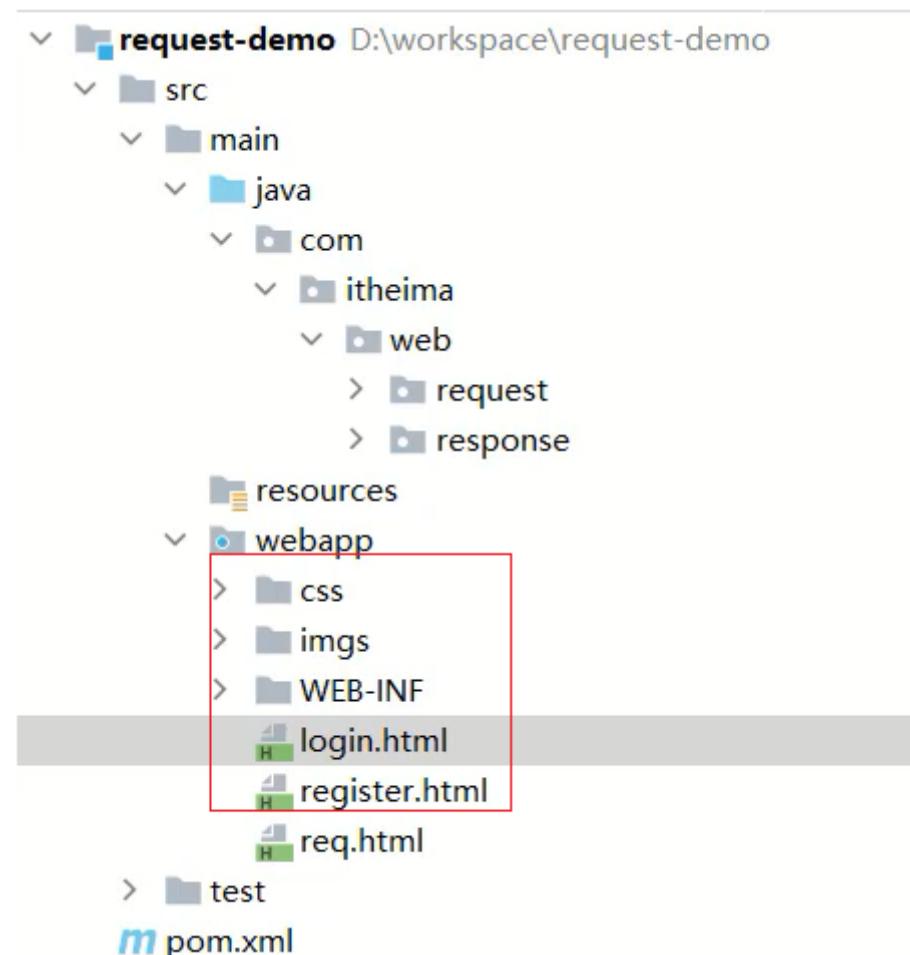


1. 用户在登录页面输入用户名和密码，提交请求给LoginServlet
2. 在LoginServlet中接收请求和数据 [用户名和密码]
3. 在LoginServlet中通过Mybatis实现调用UserMapper来根据用户名和密码查询数据库表
4. 将查询的结果封装到User对象中进行返回
5. 在LoginServlet中判断返回的User对象是否为null
6. 如果为null，说明根据用户名和密码没有查询到用户，则登录失败，返回"登录失败"数据给前端
7. 如果不为null，则说明用户存在并且密码正确，则登录成功，返回"登录成功"数据给前端

4.1.2 环境准备

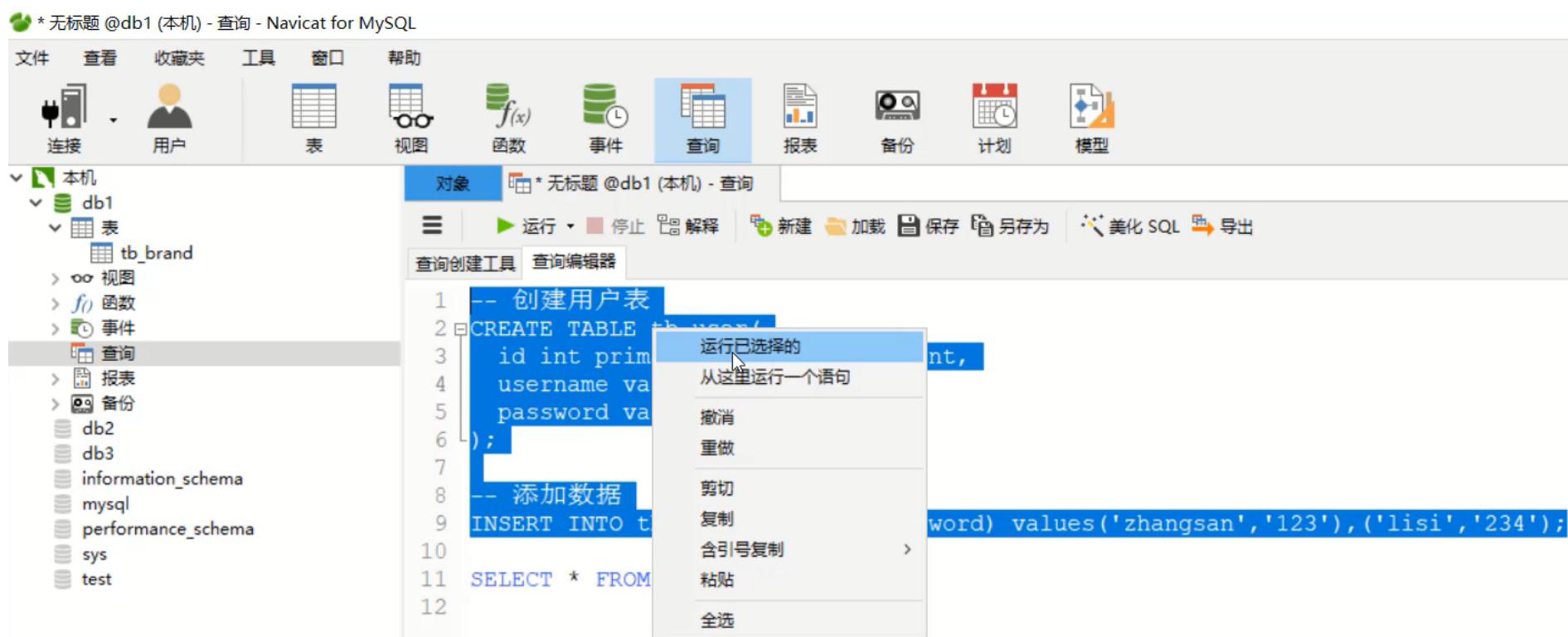
1. 复制资料中的静态页面到项目的webapp目录下

参考资料\1. 登陆注册案例\1. 静态页面，拷贝完效果如下：

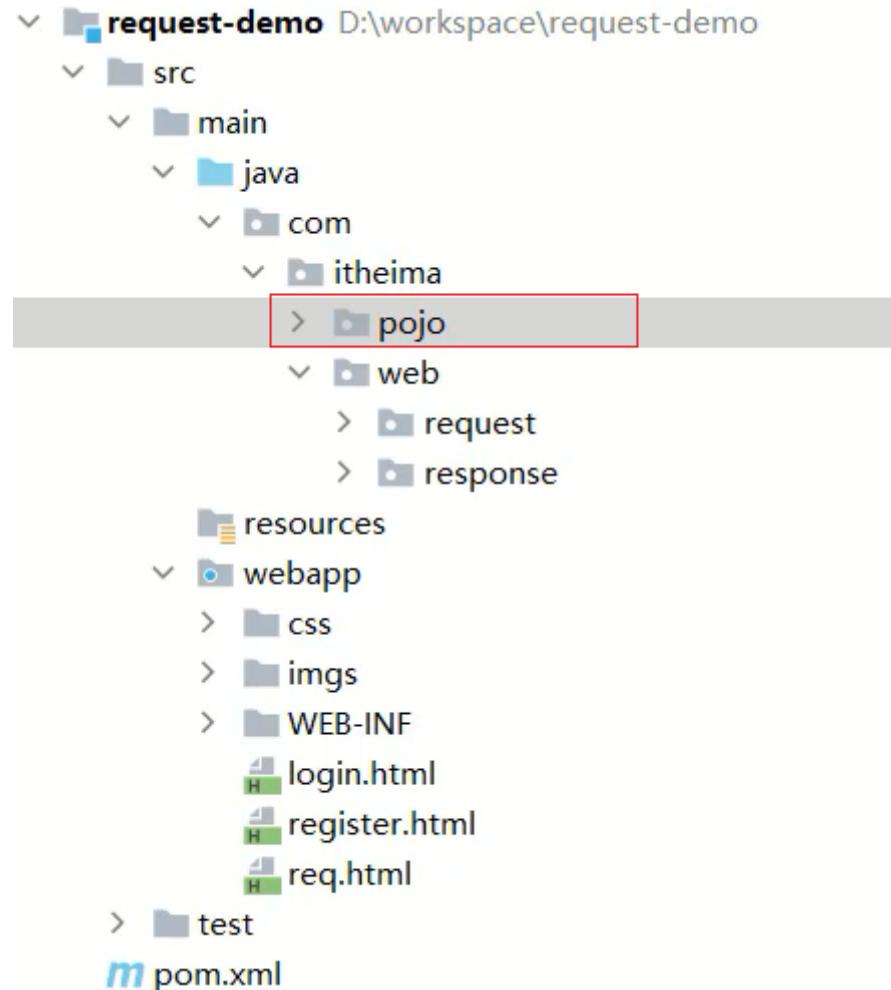


2. 创建db1数据库，创建tb_user表，创建User实体类

2.1 将资料\1. 登陆注册案例\2. MyBatis环境\tb_user.sql中的sql语句执行下：



2.2 将资料\1. 登陆注册案例\2. MyBatis环境\User.java拷贝到com.itheima.pojo



3. 在项目的pom.xml导入Mybatis和Mysql驱动坐标

```

1 <dependency>
2   <groupId>org.mybatis</groupId>
3   <artifactId>mybatis</artifactId>
4   <version>3.5.5</version>
5 </dependency>
6
7 <dependency>
8   <groupId>mysql</groupId>
9   <artifactId>mysql-connector-java</artifactId>
10  <version>5.1.34</version>
11 </dependency>
  
```

4. 创建mybatis-config.xml核心配置文件, UserMapper.xml映射文件, UserMapper接口

4.1 将资料\1. 登陆注册案例\2. MyBatis环境\mybatis-config.xml拷贝到resources目录下

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE configuration
3   PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4   "http://mybatis.org/dtd/mybatis-3-config.dtd">
5 <configuration>
6   <!--起别名-->
7   <typeAliases>
8     <package name="com.itheima.pojo"/>
9   </typeAliases>
10
11  <environments default="development">
12    <environment id="development">
13      <transactionManager type="JDBC"/>
  
```

```

14      <dataSource type="POOLED">
15          <property name="driver" value="com.mysql.jdbc.Driver"/>
16          <!--
17              useSSL:关闭SSL安全连接 性能更高
18              useServerPrepStmts:开启预编译功能
19              & 等同于 & ,xml配置文件中不能直接写 &符号
20          -->
21          <property name="url" value="jdbc:mysql:///db1?
useSSL=false&useServerPrepStmts=true"/>
22          <property name="username" value="root"/>
23          <property name="password" value="1234"/>
24      </dataSource>
25  </environment>
26 </environments>
27 <mappers>
28     <!--扫描mapper-->
29     <package name="com.itheima.mapper"/>
30 </mappers>
31 </configuration>

```

4.2 在com.itheima.mapper包下创建UserMapper接口

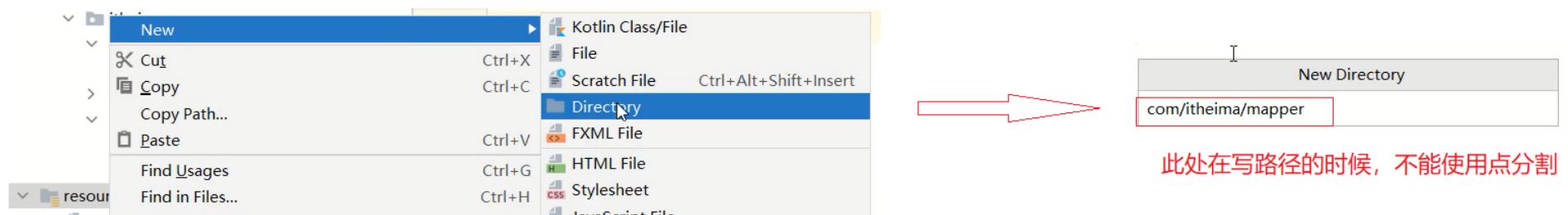
```

1 public interface UserMapper {
2
3 }

```

4.3 将资料\1. 登陆注册案例\2. MyBatis环境\UserMapper.xml拷贝到resources目录下

注意：在resources下创建UserMapper.xml的目录时，要使用/分割



至此我们所需要的环境就都已经准备好了，具体该如何实现？

4.1.3 代码实现

1. 在UserMapper接口中提供一个根据用户名和密码查询用户对象的方法

```

1 /**
2  * 根据用户名和密码查询用户对象
3  * @param username
4  * @param password
5  * @return
6 */
7 @Select("select * from tb_user where username = #{username} and password =
8 #{}password")
User select(@Param("username") String username,@Param("password") String
password);

```

说明

@Param注解的作用：用于传递参数，是方法的参数可以与SQL中的字段名相对应。

2. 修改login.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3
4 <head>
5   <meta charset="UTF-8">
6   <title>login</title>
7   <link href="css/login.css" rel="stylesheet">
8 </head>
9
10 <body>
11 <div id="loginDiv">
12   <form action="/request-demo/loginServlet" method="post" id="form">
13     <h1 id="loginMsg">LOGIN IN</h1>
14     <p>Username:<input id="username" name="username" type="text"></p>
15
16     <p>Password:<input id="password" name="password" type="password"></p>
17
18     <div id="subDiv">
19       <input type="submit" class="button" value="login up">
20       <input type="reset" class="button"
21         value="reset">&ampnbsp&ampnbsp&ampnbsp
22         <a href="register.html">没有账号？点击注册</a>
23     </div>
24   </form>
25 </div>
26 </body>
27 </html>

```

3. 编写LoginServlet

```

1  @WebServlet("/loginServlet")
2  public class LoginServlet extends HttpServlet {
3      @Override
4      protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
5          //1. 接收用户名和密码
6          String username = request.getParameter("username");
7          String password = request.getParameter("password");
8
9          //2. 调用MyBatis完成查询
10         //2.1 获取SqlSessionFactory对象
11         String resource = "mybatis-config.xml";
12         InputStream inputStream = Resources.getResourceAsStream(resource);
13         SqlSessionFactory sqlSessionFactory = new
14         SqlSessionFactoryBuilder().build(inputStream);
15         //2.2 获取sqlSession对象
16         SqlSession sqlSession = sqlSessionFactory.openSession();
17         //2.3 获取Mapper
18         UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
19         //2.4 调用方法
20         User user = userMapper.select(username, password);
21         //2.5 释放资源
22         sqlSession.close();
23
24         //获取字符输出流，并设置content type
25         response.setContentType("text/html;charset=utf-8");
26         PrintWriter writer = response.getWriter();
27         //3. 判断user释放为null
28         if(user != null){
29             // 登陆成功
30             writer.write("登陆成功");
31         }else {
32             // 登陆失败
33             writer.write("登陆失败");
34         }
35     }
36
37     @Override
38     protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
39         this.doGet(request, response);
40     }
41 }
```

4. 启动服务器测试

4.1 如果用户名和密码输入错误，则



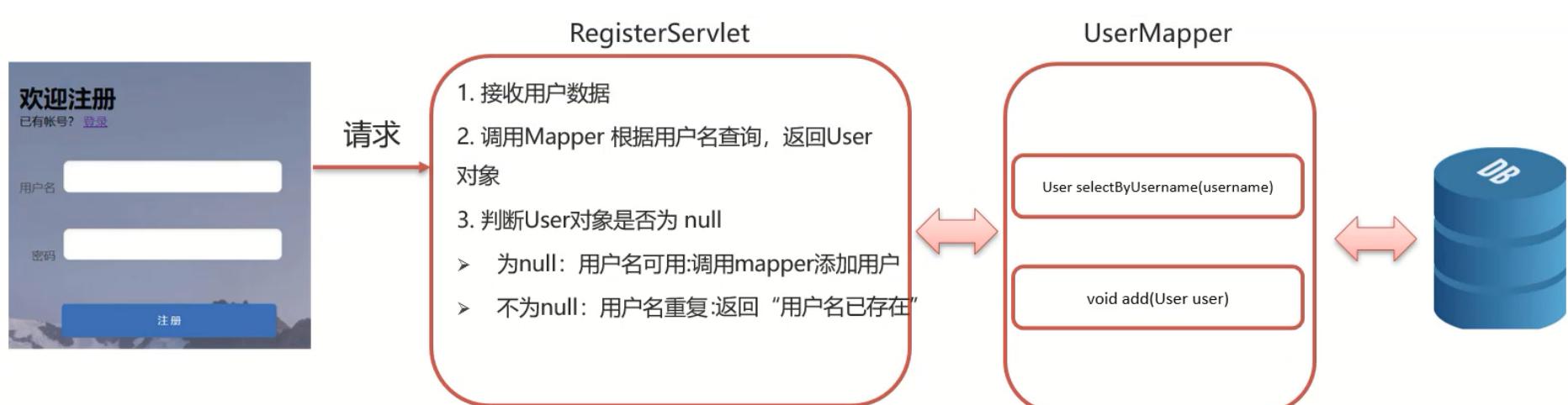
4.2 如果用户名和密码输入正确，则



至此用户的登录功能就已经完成了~

4.2 用户注册

4.2.1 需求分析



1. 用户在注册页面输入用户名和密码，提交请求给RegisterServlet
2. 在RegisterServlet中接收请求和数据 [用户名和密码]
3. 在RegisterServlet中通过Mybatis实现调用UserMapper来根据用户名查询数据库表
4. 将查询的结果封装到User对象中进行返回
5. 在RegisterServlet中判断返回的User对象是否为null
6. 如果为null，说明根据用户名可用，则调用UserMapper来实现添加用户
7. 如果不为null，则说明用户名不可以，返回"用户名已存在"数据给前端

4.2.2 代码编写

1. 编写UserMapper提供根据用户名查询用户数据方法和添加用户方法

```
1 /**
2 * 根据用户名查询用户对象
3 * @param username
4 * @return
5 */
```

```
6 @Select("select * from tb_user where username = #{username}")
7 User selectByUsername(String username);
8
9 /**
10 * 添加用户
11 * @param user
12 */
13 @Insert("insert into tb_user values(null,#{username},#{password})")
14 void add(User user);
```

2. 修改register.html

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>欢迎注册</title>
6     <link href="css/register.css" rel="stylesheet">
7 </head>
8 <body>
9
10 <div class="form-div">
11     <div class="reg-content">
12         <h1>欢迎注册</h1>
13         <span>已有帐号? </span> <a href="login.html">登录</a>
14     </div>
15     <form id="reg-form" action="/request-demo/registerServlet" method="post">
16
17         <table>
18
19             <tr>
20                 <td>用户名</td>
21                 <td class="inputs">
22                     <input name="username" type="text" id="username">
23                     <br>
24                     <span id="username_err" class="err_msg" style="display:
none">用户名不太受欢迎</span>
25                 </td>
26
27             </tr>
28
29             <tr>
30                 <td>密码</td>
31                 <td class="inputs">
32                     <input name="password" type="password" id="password">
33                     <br>
34                     <span id="password_err" class="err_msg" style="display:
none">密码格式有误</span>
```

```

35         </td>
36     </tr>
37
38 </table>
39
40 <div class="buttons">
41     <input value="注 册" type="submit" id="reg_btn">
42 </div>
43 <br class="clear">
44 </form>
45
46 </div>
47 </body>
48 </html>

```

3. 创建RegisterServlet类

```

1 @WebServlet("/registerServlet")
2 public class RegisterServlet extends HttpServlet {
3     @Override
4     protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
5         //1. 接收用户数据
6         String username = request.getParameter("username");
7         String password = request.getParameter("password");
8
9         //封装用户对象
10        User user = new User();
11        user.setUsername(username);
12        user.setPassword(password);
13
14        //2. 调用mapper 根据用户名查询用户对象
15        //2.1 获取SqlSessionFactory对象
16        String resource = "mybatis-config.xml";
17        InputStream inputStream = Resources.getResourceAsStream(resource);
18        SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
19        //2.2 获取SqlSession对象
20        SqlSession sqlSession = sqlSessionFactory.openSession();
21        //2.3 获取Mapper
22        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
23
24        //2.4 调用方法
25        User u = userMapper.selectByUsername(username);
26
27        //3. 判断用户对象释放为null
28        if( u == null){
29            // 用户名不存在, 添加用户

```

```

30         userMapper.add(user);
31
32         // 提交事务
33         sqlSession.commit();
34         // 释放资源
35         sqlSession.close();
36     }else {
37         // 用户名存在，给出提示信息
38         response.setContentType("text/html;charset=utf-8");
39         response.getWriter().write("用户名已存在");
40     }
41 }
42
43
44     @Override
45     protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
46         this doGet(request, response);
47     }
48 }
```

4. 启动服务器进行测试

4.1 如果测试成功，则在数据库中就能查看到新注册的数据

4.2 如果用户已经存在，则在页面上展示 `用户名已存在` 的提示信息

4.3 SqlSessionFactory工具类抽取

上面两个功能已经实现，但是在写Servlet的时候，因为需要使用Mybatis来完成数据库的操作，所以对于Mybatis的基础操作就出现了些重复代码，如下

```

1 String resource = "mybatis-config.xml";
2 InputStream inputStream = Resources.getResourceAsStream(resource);
3 SqlSessionFactory sqlSessionFactory = new
4     SqlSessionFactoryBuilder().build(inputStream);
```

有了这些重复代码就会造成一些问题：

- 重复代码不利于后期的维护
- SqlSessionFactory工厂类进行重复创建
 - 就相当于每次买手机都需要重新创建一个手机生产工厂来给你制造一个手机一样，资源消耗非常大但性能却非常低。所以这么做是不允许的。

那如何来优化呢？

- 代码重复可以抽取工具类
- 对指定代码只需要执行一次可以使用静态代码块

有了这两个方向后，代码具体该如何编写？

```
1 public class SqlSessionFactoryUtils {  
2  
3     private static SqlSessionFactory sqlSessionFactory;  
4  
5     static {  
6         //静态代码块会随着类的加载而自动执行，且只执行一次  
7         try {  
8             String resource = "mybatis-config.xml";  
9             InputStream inputStream =  
Resources.getResourceAsStream(resource);  
10            sqlSessionFactory = new  
SqlSessionFactoryBuilder().build(inputStream);  
11        } catch (IOException e) {  
12            e.printStackTrace();  
13        }  
14    }  
15  
16  
17    public static SqlSessionFactory getSqlSessionFactory(){  
18        return sqlSessionFactory;  
19    }  
20 }
```

工具类抽取以后，以后在对Mybatis的SqlSession进行操作的时候，就可以直接使用

```
1 SqlSessionFactory sqlSessionFactory  
=SqlSessionFactoryUtils.getSqlSessionFactory();
```

这样就可以很好的解决上面所说的代码重复和重复创建工厂导致性能低的问题了。