Servlet

基础

在JavaEE平台上，处理TCP连接，解析HTTP协议这些底层工作统统扔给现成的Web服务器去做，我们只需要把自己的应用程序跑在Web服务器上。为了实现这一目的，JavaEE提供了Servlet API，我们使用Servlet API编写自己的Servlet来处理HTTP请求，Web服务器实现Servlet API接口，实现底层功能：

```
                ┌─────────────┐
                │My Servlet   │
                ├─────────────┤
                │Servlet API  │
┌─────────┐ HTTP├─────────────┤
│Browser  │<────>│Web Server   │
└─────────┘     └─────────────┘
```

基础servlet (不结合jsp)

```
// WebServlet注解表示这是一个Servlet，并映射到地址/:
@WebServlet(urlPatterns = "/")
public class HelloServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        // 设置响应类型:
        resp.setContentType("text/html");
        // 获取输出流:
        PrintWriter pw = resp.getWriter();
        // 写入响应:
        pw.write("<h1>Hello, world!</h1>");
        // 最后不要忘记flush强制输出:
        pw.flush();
    }
}
```

在Servlet容器中运行的Servlet具有如下特点：

- 无法在代码中直接通过new创建Servlet实例，必须由Servlet容器自动创建Servlet实例；
- Servlet容器只会给每个Servlet类创建唯一实例；
- Servlet容器会使用多线程执行`doGet()`或`doPost()`方法。

复习一下Java多线程的内容，我们可以得出结论：

- 在Servlet中定义的实例变量会被多个线程同时访问，要注意线程安全；
- `HttpServletRequest`和`HttpServletResponse`实例是由Servlet容器传入的局部变量，它们只能被当前线程访问，不存在多个线程访问的问题；
- 在`doGet()`或`doPost()`方法中，如果使用了`ThreadLocal`，但没有清理，那么它的状态很可能会影响到下次的某个请求，因为Servlet容器很可能用线程池实现线程复用。

进阶

一个Web App就是由一个或多个Servlet组成的，每个Servlet通过注解说明自己能处理的路径。例如：

```
@WebServlet(urlPatterns = "/hello")
public class HelloServlet extends HttpServlet {
    ...
}
```

上述`HelloServlet`能处理`/hello`这个路径的请求。

早期的Servlet需要在web.xml中配置映射路径，但最新Servlet版本只需要通过注解就可以完成映射。

因为浏览器发送请求的时候，还会有请求方法（HTTP Method）：即GET、POST、PUT等不同类型的请求。因此，要处理GET请求，我们要覆写`doGet()`方法：

```
@WebServlet(urlPatterns = "/hello")
public class HelloServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        ...
    }
}
```

类似的，要处理POST请求，就需要覆写`doPost()`方法。

如果没有覆写`doPost()`方法，那么`HelloServlet`能不能处理`POST /hello`请求呢？

我们查看一下`HttpServlet`的`doPost()`方法就一目了然了：它会直接返回405或400错误。因此，一个Servlet如果映射到`/hello`，那么所有请求方法都会由这个Servlet处理，至于能不能返回200成功响应，要看有没有覆写对应的请求方法。

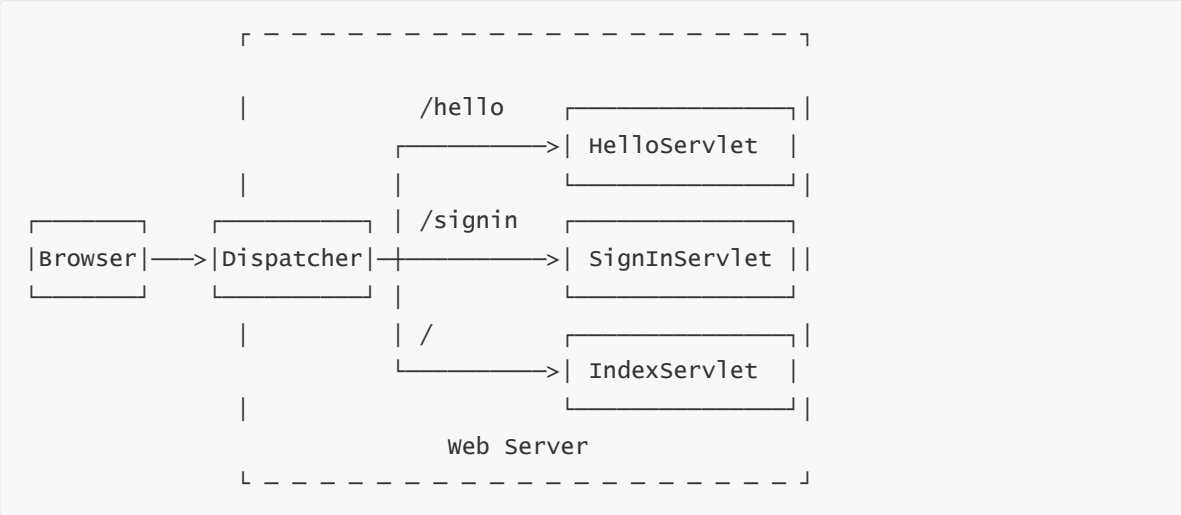
一个Webapp完全可以有多个Servlet，分别映射不同的路径。例如：

```
@WebServlet(urlPatterns = "/hello")
public class HelloServlet extends HttpServlet {
    ...
}

@WebServlet(urlPatterns = "/signin")
public class SignInServlet extends HttpServlet {
    ...
}

@WebServlet(urlPatterns = "/")
public class IndexServlet extends HttpServlet {
    ...
}
```

浏览器发出的HTTP请求总是由Web Server先接收，然后，根据Servlet配置的映射，不同的路径转发到不同的Servlet：



这种根据路径转发的功能我们一般称为Dispatch。映射到`/`的`IndexServlet`比较特殊，它实际上会接收所有未匹配的路径，相当于`/*`，因为Dispatcher的逻辑可以用伪代码实现如下：

```
String path = ...
if (path.equals("/hello")) {
    dispatchTo(helloServlet);
} else if (path.equals("/signin")) {
    dispatchTo(signinServlet);
} else {
    // 所有未匹配的路径均转发到"/"
    dispatchTo(indexServlet);
}
```

所以我们在浏览器输入一个`http://localhost:8080/abc`也会看到`IndexServlet`生成的页面。

HttpServletRequest

`HttpServletRequest`封装了一个HTTP请求，它实际上是从`ServletRequest`继承而来。最早设计Servlet时，设计者希望Servlet不仅能处理HTTP，也能处理类似SMTP等其他协议，因此，单独抽出了`ServletRequest`接口，但实际上除了HTTP外，并没有其他协议会用Servlet处理，所以这是一个过度设计。

我们通过`HttpServletRequest`提供的接口方法可以拿到HTTP请求的几乎全部信息，常用的方法有：

- getMethod()：返回请求方法，例如，`"GET"`，`"POST"`；
- **getRequestURI()**：返回请求路径，但不包括请求参数，例如，`"/hello"`；
- getQueryString()：返回请求参数，例如，`"name=Bob&a=1&b=2"`；
- **getParameter(name)**：返回请求参数，GET请求从URL读取参数，POST请求从Body中读取参数；
- getContentType()：获取请求Body的类型，例如，`"application/x-www-form-urlencoded"`；
- **getContextPath()**：获取当前Webapp挂载的路径，对于ROOT来说，总是返回空字符串`""`；
- getCookies()：返回请求携带的所有Cookie；
- getHeader(name)：获取指定的Header，对Header名称不区分大小写；
- getHeaderNames()：返回所有Header名称；
- getInputStream()：如果该请求带有HTTP Body，该方法将打开一个输入流用于读取Body；
- getReader()：和getInputStream()类似，但打开的是Reader；
- getRemoteAddr()：返回客户端的IP地址；
- getScheme()：返回协议类型，例如，`"http"`，`"https"`；

此外，`HttpServletRequest`还有两个方法：`setAttribute()`和`getAttribute()`，可以给当前`HttpServletRequest`对象附加多个Key-Value，相当于把`HttpServletRequest`当作一个`Map<String, Object>`使用。

调用`HttpServletRequest`的方法时，注意务必阅读接口方法的文档说明，因为有的方法会返回`null`，例如`getQueryString()`的文档就写了：

```
... This method returns null if the URL does not have a query string...
```

HttpServletResponse

`HttpServletResponse`封装了一个HTTP响应。由于HTTP响应必须先发送Header，再发送Body，所以，操作`HttpServletResponse`对象时，必须先调用设置Header的方法，最后调用发送Body的方法。

常用的设置Header的方法有：

- setStatus(sc)：设置响应代码，默认是`200`；
- setContentType(type)：设置Body的类型，例如，`"text/html"`；
- setCharacterEncoding(charset)：设置字符编码，例如，`"UTF-8"`；
- setHeader(name, value)：设置一个Header的值；
- **addCookie(cookie)**：给响应添加一个Cookie；
- **addHeader(name, value)**：给响应添加一个Header，因为HTTP协议允许有多个相同的Header；

写入响应时，需要通过`getOutputStream()`获取写入流，或者通过`getWriter()`获取字符流，二者只能获取其中一个。

写入响应前，无需设置`setContentLength()`，因为底层服务器会根据写入的字节数自动设置，如果写入的数据量很小，实际上会先写入缓冲区，如果写入的数据量很大，服务器会自动采用Chunked编码让浏览器能识别数据结束符而不需要设置Content-Length头。

但是，写入完毕后调用`flush()`却是必须的，因为大部分Web服务器都基于HTTP/1.1协议，会复用TCP连接。如果没有调用`flush()`，将导致缓冲区的内容无法及时发送到客户端。此外，写入完毕后千万不要调用`close()`，原因同样是因为会复用TCP连接，如果关闭写入流，将关闭TCP连接，使得Web服务器无法复用此TCP连接。

写入完毕后对输出流调用flush()而不是close()方法！

有了`HttpServletRequest`和`HttpServletResponse`这两个高级接口，我们就不需要直接处理HTTP协议。注意到具体的实现类是由各服务器提供的，而我们编写的Web应用程序只关心接口方法，并不需要关心具体实现的子类。

Servlet多线程模型

一个Servlet类在服务器中只有一个实例，但对于每个HTTP请求，Web服务器会使用多线程执行请求。因此，一个Servlet的`doGet()`、`doPost()`等处理请求的方法是多线程并发执行的。如果Servlet中定义了字段，要注意多线程并发访问的问题：

```
public class HelloServlet extends HttpServlet {
    private Map<String, String> map = new ConcurrentHashMap<>();

    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 注意读写map字段是多线程并发的:
        this.map.put(key, value);
    }
}
```

对于每个请求，Web服务器会创建唯一的`HttpServletRequest`和`HttpServletResponse`实例，因此，`HttpServletRequest`和`HttpServletResponse`实例只有在当前处理线程中有效，它们总是局部变量，不存在多线程共享的问题。

小结

一个Webapp中的多个Servlet依靠路径映射来处理不同的请求；

映射为`/`的Servlet可处理所有“未匹配”的请求；

如何处理请求取决于Servlet覆写的对应方法；

Web服务器通过多线程处理HTTP请求，一个Servlet的处理方法可以由多线程并发执行。

重定向和转发

Redirect

重定向是指当浏览器请求一个URL时，服务器返回一个重定向指令，告诉浏览器地址已经变了，麻烦**使用新的URL再重新发送新请求。**

例如，我们已经编写了一个能处理`/hello`的`HelloServlet`，如果收到的路径为`/hi`，希望能重定向到`/hello`，可以再编写一个`RedirectServlet`：

```
@WebServlet(urlPatterns = "/hi")
public class RedirectServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 构造重定向的路径:
        String name = req.getParameter("name");
        String redirectToUrl = "/hello" + (name == null ? "" : "?name=" + name);
        // 发送重定向响应:
        resp.sendRedirect(redirectToUrl);
    }
}
```

如果浏览器发送`GET /hi`请求，`RedirectServlet`将处理此请求。由于`RedirectServlet`在内部又发送了重定向响应，因此，浏览器会收到如下响应：

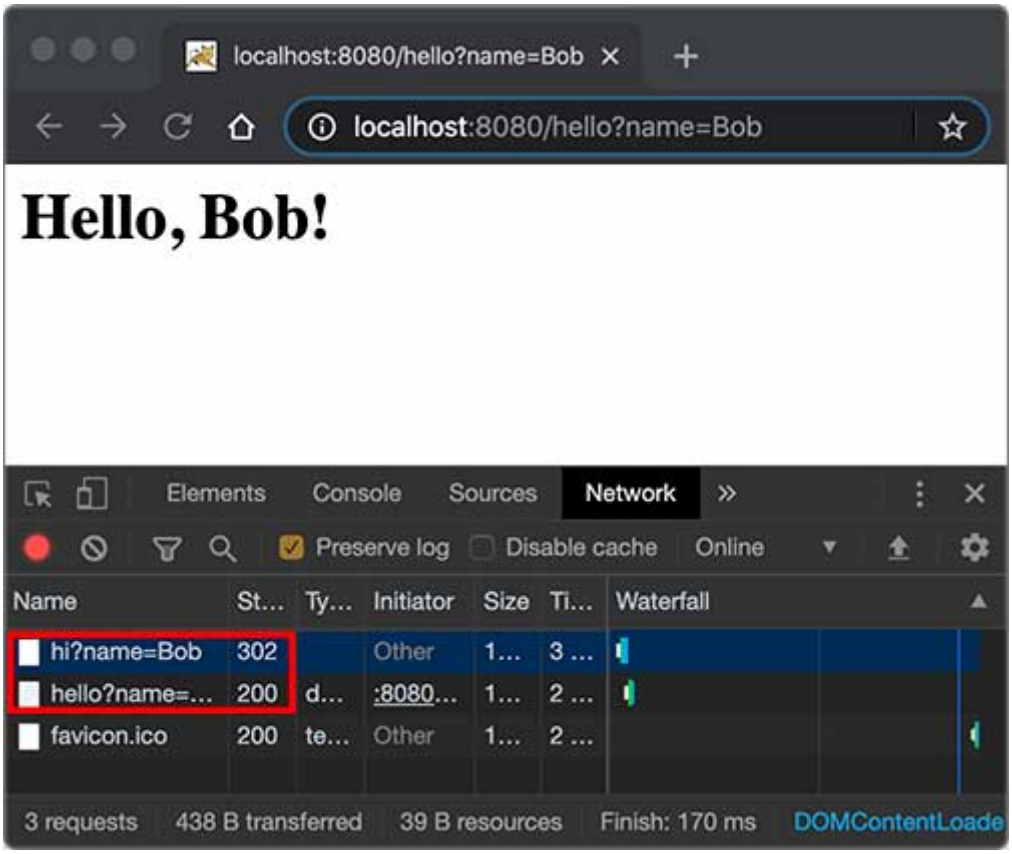
```
HTTP/1.1 302 Found
Location: /hello
```

当浏览器收到302响应后，它会立刻根据`Location`的指示发送一个新的`GET /hello`请求，这个过程就是重定向：

```
┌────────┐    GET /hi     ┌─────────────────┐
|Browser| ──────────────> |RedirectServlet|
|        | <──────────────  |                 |
└────────┘    302          └─────────────────┘


┌────────┐   GET /hello    ┌─────────────────┐
|Browser| ──────────────> |  HelloServlet   |
|        | <──────────────  |                 |
└────────┘    200 <html>   └─────────────────┘
```

观察Chrome浏览器的网络请求，可以看到两次HTTP请求：



并且浏览器的地址栏路径自动更新为`/hello`。

重定向有两种：一种是302响应，称为临时重定向，一种是301响应，称为永久重定向。两者的区别是，如果服务器发送301永久重定向响应，浏览器会缓存`/hi`到`/hello`这个重定向的关联，下次请求`/hi`的时候，浏览器就直接发送`/hello`请求了。

重定向有什么作用？**重定向的目的是当Web应用升级后，如果请求路径发生了变化，可以将原来的路径重定向到新路径，从而避免浏览器请求原路径找不到资源。**

`HttpServletResponse`提供了快捷的`redirect()`方法实现302重定向。如果要实现301永久重定向，可以这么写：

```
resp.setStatus(HttpServletResponse.SC_MOVED_PERMANENTLY); // 301
resp.setHeader("Location", "/hello");
```

Forward

Forward是指内部转发。当一个Servlet处理请求的时候，它可以决定**自己不继续处理，而是转发给另一个Servlet处理。**

此时的请求和响应都以引用的形式传递, 即多个servlet共享.

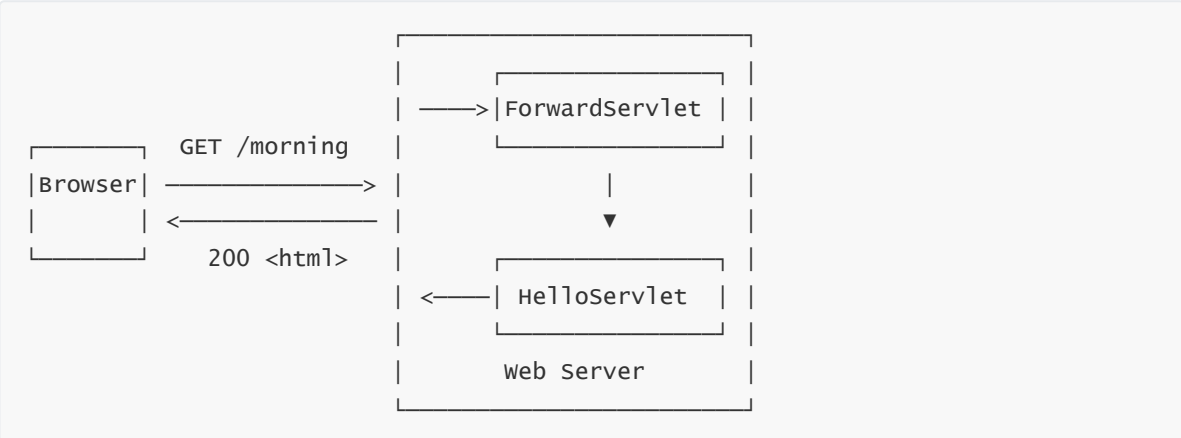
例如，我们已经编写了一个能处理`/hello`的`HelloServlet`，继续编写一个能处理`/morning`的`ForwardServlet`：

```
@WebServlet(urlPatterns = "/morning")
public class ForwardServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        req.getRequestDispatcher("/hello").forward(req, resp);
    }
}
```

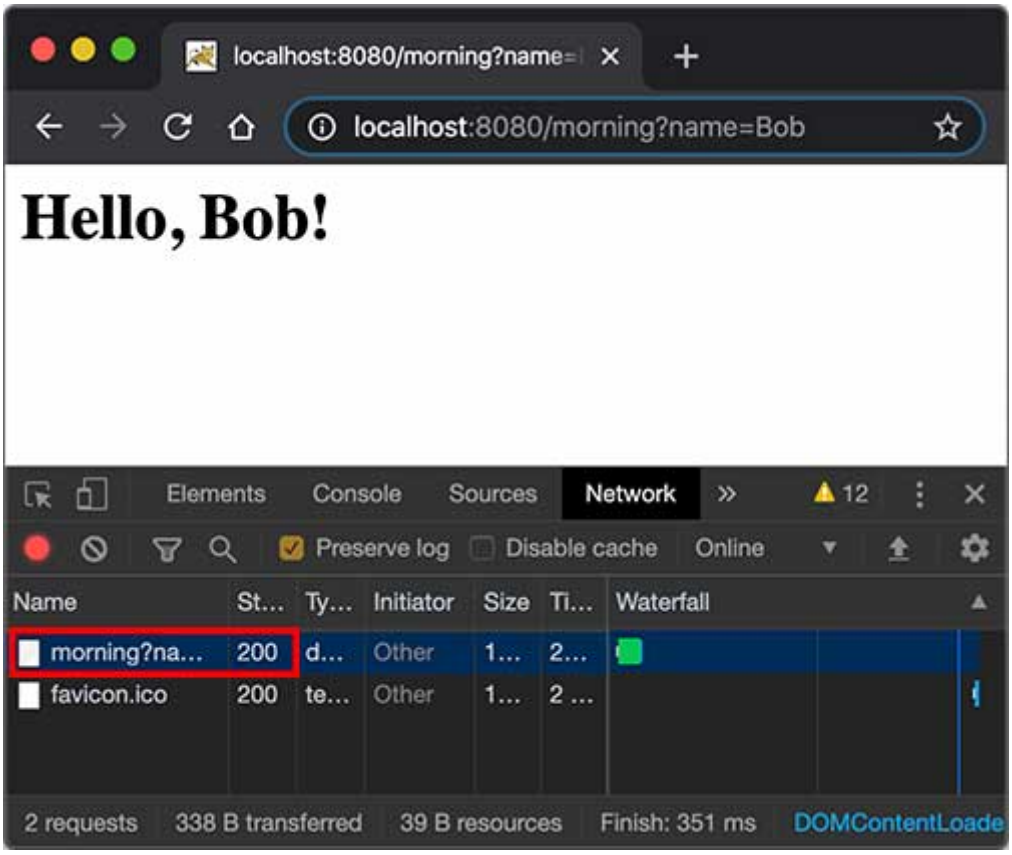
`ForwardServlet`在收到请求后，它并不自己发送响应，而是把请求和响应都转发给路径为`/hello`的Servlet，即下面的代码：

```
req.getRequestDispatcher("/hello").forward(req, resp);
```

后续请求的处理实际上是由`HelloServlet`完成的。这种处理方式称为转发（Forward），我们用流程图画出来如下：



转发和重定向的区别在于，转发是在Web服务器内部完成的，对浏览器来说，它只发出了一个HTTP请求：



注意到使用转发的时候，浏览器的地址栏路径仍然是`/morning`，浏览器并不知道该请求在Web服务器内部实际上做了一次转发。

Session和Cookie

在Web应用程序中，我们经常要跟踪用户身份。当一个用户登录成功后，如果他继续访问其他页面，Web程序如何才能识别出该用户身份？

因为HTTP协议是一个无状态协议，即Web应用程序无法区分收到的两个HTTP请求是否是同一个浏览器发出的。为了跟踪用户状态，服务器可以向浏览器分配一个唯一ID，并以Cookie的形式发送到浏览器，浏览器在后续访问时总是附带此Cookie，这样，服务器就可以识别用户身份。

Session

我们把这种基于唯一ID识别用户身份的机制称为Session。每个用户第一次访问服务器后，会自动获得一个Session ID。如果用户在一段时间内没有访问服务器，那么Session会自动失效，下次即使带着上次分配的Session ID访问，服务器也认为这是一个新用户，会分配新的Session ID。

JavaEE的Servlet机制内建了对Session的支持。我们以登录为例，当一个用户登录成功后，我们就可以把**这个用户的名字放入一个`HttpSession`对象**，以便后续访问其他页面的时候，能直接从`HttpSession`取出用户名：

```
@WebServlet(urlPatterns = "/signin")
public class SignInServlet extends HttpServlet {
    // 模拟一个数据库:
    private Map<String, String> users = Map.of("bob", "bob123", "alice",
"alice123", "tom", "tomcat");

    // GET请求时显示登录页:
```

```
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        resp.setContentType("text/html");
        PrintWriter pw = resp.getWriter();
        pw.write("<h1>Sign In</h1>");
        pw.write("<form action=\"/signin\" method=\"post\">");
        pw.write("<p>Username: <input name=\"username\"></p>");
        pw.write("<p>Password: <input name=\"password\" type=\"password\">
</p>");
        pw.write("<p><button type=\"submit\">Sign In</button> <a
href=\"/\">Cancel</a></p>");
        pw.write("</form>");
        pw.flush();
    }

    // POST请求时处理用户登录:
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        String name = req.getParameter("username");
        String password = req.getParameter("password");
        String expectedPassword = users.get(name.toLowerCase());
        if (expectedPassword != null && expectedPassword.equals(password)) {
            // 登录成功:
            req.getSession().setAttribute("user", name);
            resp.sendRedirect("/");
        } else {
            resp.sendError(HttpServletResponse.SC_FORBIDDEN);
        }
    }
}
```

上述`SignInServlet`在判断用户登录成功后，立刻将用户名放入当前`HttpSession`中：

```
HttpSession session = req.getSession();
session.setAttribute("user", name);
```

在`IndexServlet`中，可以从`HttpSession`取出用户名：

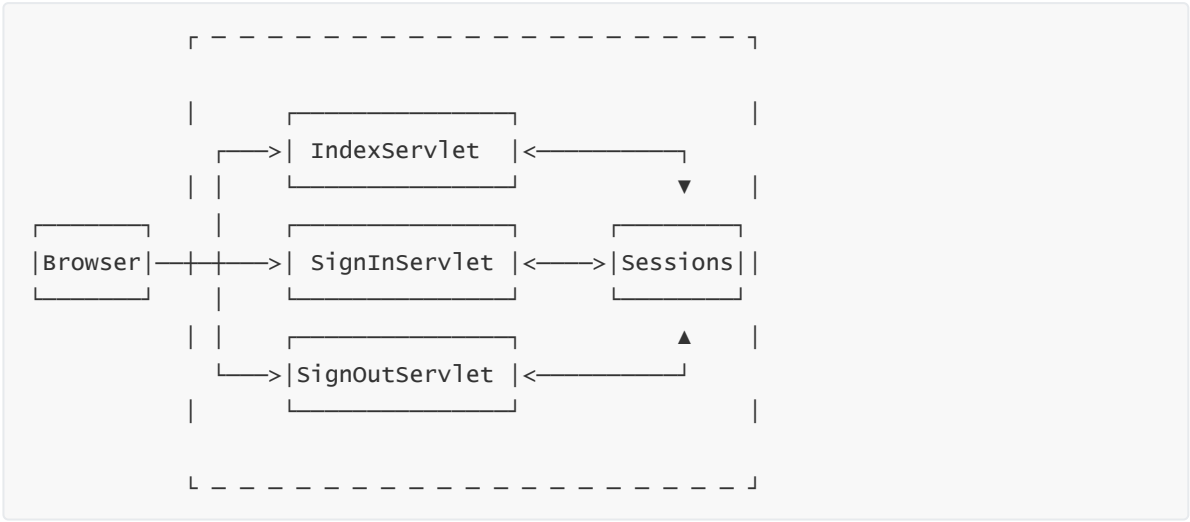
```
@WebServlet(urlPatterns = "/")
public class IndexServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 从HttpSession获取当前用户名:
        String user = (String) req.getSession().getAttribute("user");
        resp.setContentType("text/html");
        resp.setCharacterEncoding("UTF-8");
        resp.setHeader("X-Powered-By", "JavaEE Servlet");
        PrintWriter pw = resp.getWriter();
        pw.write("<h1>Welcome, " + (user != null ? user : "Guest") + "</h1>");
        if (user == null) {
            // 未登录，显示登录链接:
            pw.write("<p><a href=\"/signin\">Sign In</a></p>");
        } else {
            // 已登录，显示登出链接:
```

```
            pw.write("<p><a href=\"/signout\">Sign Out</a></p>");
        }
        pw.flush();
    }
}
```

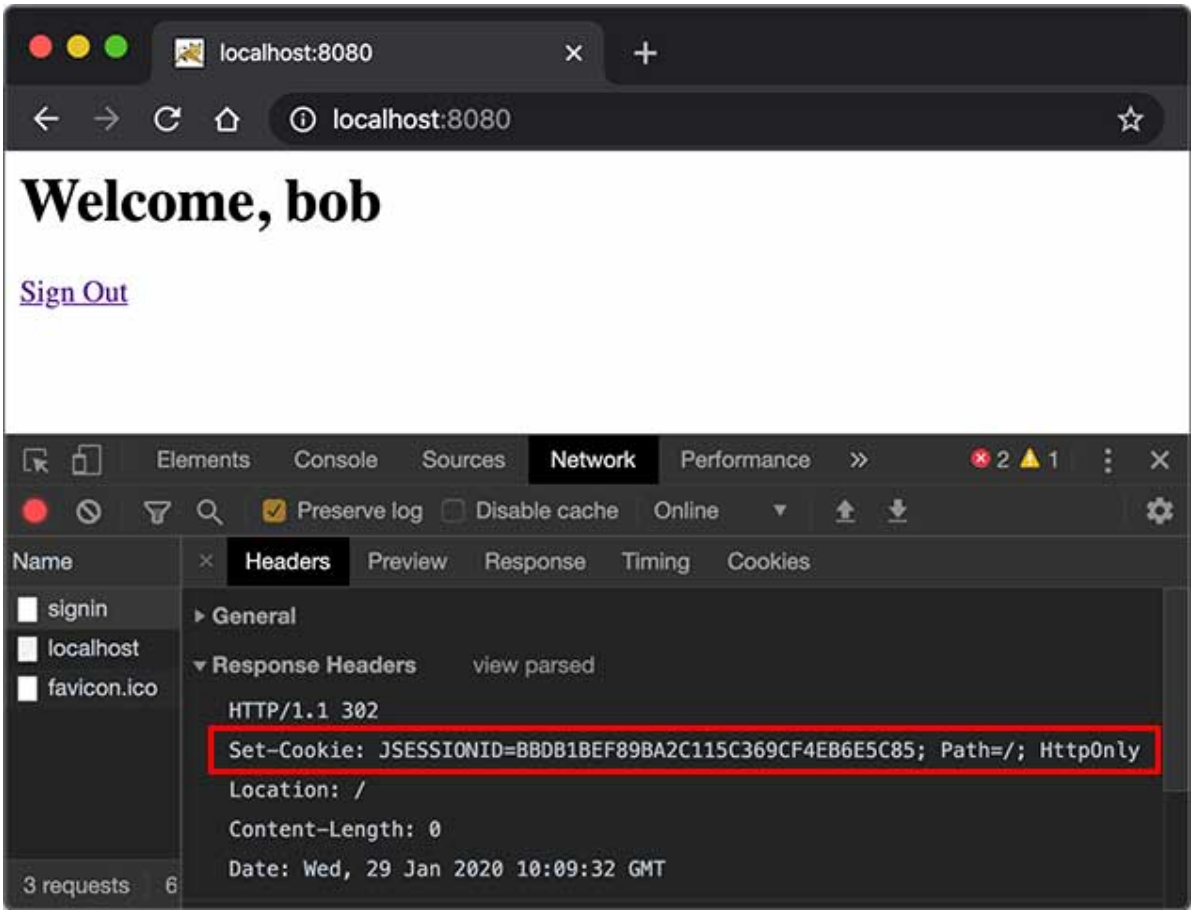
如果用户已登录，可以通过访问`/signout`登出。登出逻辑就是从`HttpSession`中移除用户相关信息：

```
@WebServlet(urlPatterns = "/signout")
public class SignOutServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 从HttpSession移除用户名:
        req.getSession().removeAttribute("user");
        resp.sendRedirect("/");
    }
}
```

对于Web应用程序来说，我们总是通过`HttpSession`这个高级接口访问当前Session。如果要深入理解Session原理，可以认为Web服务器在内存中自动维护了一个ID到`HttpSession`的映射表，我们可以用下图表示：



而服务器识别Session的关键就是依靠一个名为`JSESSIONID`的Cookie。在Servlet中第一次调用`req.getSession()`时，Servlet容器自动创建一个Session ID，然后通过一个名为`JSESSIONID`的Cookie发送给浏览器：



这里要注意的几点是：

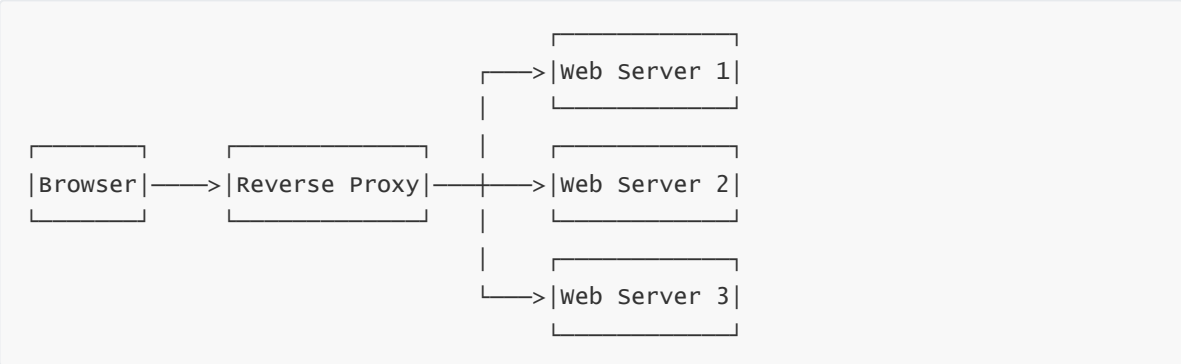
- `JSESSIONID`是由Servlet容器自动创建的，目的是维护一个浏览器会话，它和我们的登录逻辑没有关系；
- 登录和登出的业务逻辑是我们自己根据`HttpSession`是否存在一个`"user"`的Key判断的，登出后，Session ID并不会改变；
- 即使没有登录功能，仍然可以使用`HttpSession`追踪用户，例如，放入一些用户配置信息等。

除了使用Cookie机制可以实现Session外，还可以通过隐藏表单、URL末尾附加ID来追踪Session。这些机制很少使用，最常用的Session机制仍然是Cookie。

使用Session时，由于**服务器把所有用户的Session都存储在内存中**，如果遇到内存不足的情况，就需要把部分不活动的Session序列化到磁盘上，这会大大降低服务器的运行效率，因此，放入Session的对象要小，通常我们放入一个简单的`User`对象就足够了：

```
public class User {
    public long id; // 唯一标识
    public String email;
    public String name;
}
```

在使用多台服务器构成集群时，使用Session会遇到一些额外的问题。通常，多台服务器集群使用反向代理作为网站入口：



如果多台Web Server采用无状态集群，那么反向代理总是以轮询方式将请求依次转发给每台Web Server，这会造成一个用户在Web Server 1存储的Session信息，在Web Server 2和3上并不存在，即从Web Server 1登录后，如果后续请求被转发到Web Server 2或3，那么用户看到的仍然是未登录状态。

要解决这个问题，方案一是在所有Web Server之间进行Session复制，但这样会严重消耗网络带宽，并且，每个Web Server的内存均存储所有用户的Session，内存使用率很低。

另一个方案是采用粘滞会话（Sticky Session）机制，即反向代理在转发请求的时候，总是根据JSESSIONID的值判断，相同的JSESSIONID总是转发到固定的Web Server，但这需要反向代理的支持。

无论采用何种方案，使用Session机制，会使得Web Server的集群很难扩展，因此，Session适用于中小型Web应用程序。**对于大型Web应用程序来说，通常需要避免使用Session机制。**

Cookie

实际上，Servlet提供的`HttpSession`本质上就是通过一个名为`JSESSIONID`的Cookie来跟踪用户会话的。除了这个名称外，其他名称的Cookie我们可以任意使用。

如果我们想要设置一个Cookie，例如，记录用户选择的语言，可以编写一个`LanguageServlet`：

```
@WebServlet(urlPatterns = "/pref")
public class LanguageServlet extends HttpServlet {

    private static final Set<String> LANGUAGES = Set.of("en", "zh");

    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        String lang = req.getParameter("lang");
        if (LANGUAGES.contains(lang)) {
            // 创建一个新的Cookie:
            Cookie cookie = new Cookie("lang", lang);
            // 该Cookie生效的路径范围:
            cookie.setPath("/");
            // 该Cookie有效期:
            cookie.setMaxAge(8640000); // 8640000秒=100天
            // 将该Cookie添加到响应:
            resp.addCookie(cookie);
        }
        resp.sendRedirect("/");
    }
}
```

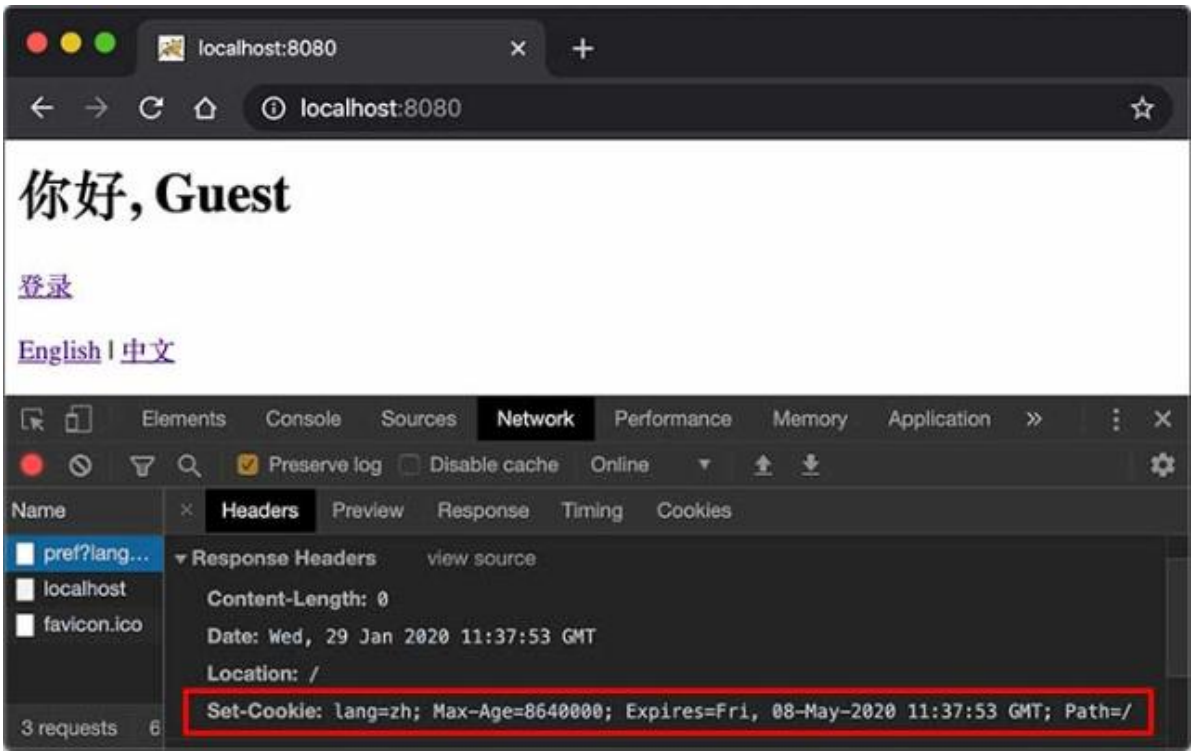
创建一个新Cookie时，除了指定名称和值以外，通常需要设置`setPath("/")`，浏览器根据此前缀决定是否发送Cookie。如果一个Cookie调用了`setPath("/user/")`，那么浏览器只有在请求以`/user/`开头的路径时才会附加此Cookie。通过`setMaxAge()`设置Cookie的有效期，单位为秒，最后通过`resp.addCookie()`把它添加到响应。

如果访问的是https网页，还需要调用`setSecure(true)`，否则浏览器不会发送该Cookie。

因此，务必注意：浏览器在请求某个URL时，是否携带指定的Cookie，取决于Cookie是否满足以下所有要求：

- URL前缀是设置Cookie时的Path；
- Cookie在有效期内；
- Cookie设置了secure时必须以https访问。

我们可以在浏览器看到服务器发送的Cookie：



如果我们要读取Cookie，例如，在`IndexServlet`中，读取名为`lang`的Cookie以获取用户设置的语言，可以写一个方法如下：

```
private String parseLanguageFromCookie(HttpServletRequest req) {
    // 获取请求附带的所有Cookie:
    Cookie[] cookies = req.getCookies();
    // 如果获取到Cookie:
    if (cookies != null) {
        // 循环每个Cookie:
        for (Cookie cookie : cookies) {
            // 如果Cookie名称为lang:
            if (cookie.getName().equals("lang")) {
                // 返回Cookie的值:
                return cookie.getValue();
            }
        }
    }
    // 返回默认值:
    return "en";
```

```
}
```

可见，读取Cookie主要依靠遍历`HttpServletRequest`附带的所有Cookie。

总结:

servlet要获得参数有三种方式:

- 直接从浏览器请求路径上获取
- 从服务器的session中获取
- 从浏览器的cookie中获取

结合JSP开发

我们从前面的章节可以看到，Servlet就是一个能处理HTTP请求，发送HTTP响应的小程序，而发送响应无非就是获取`PrintWriter`，然后输出HTML：

```
PrintWriter pw = resp.getWriter();
pw.write("<html>");
pw.write("<body>");
pw.write("<h1>Welcome, " + name + "!</h1>");
pw.write("</body>");
pw.write("</html>");
pw.flush();
```

只不过，用PrintWriter输出HTML比较痛苦，因为不但要正确编写HTML，还需要插入各种变量。如果想在Servlet中输出一个类似新浪首页的HTML，写对HTML基本上不太可能。那有没有更简单的输出HTML的办法？我们可以使用JSP。

JSP是Java Server Pages的缩写，它的文件必须放到`/src/main/webapp`下，文件名必须以`.jsp`结尾，整个文件与HTML并无太大区别，但需要插入变量，或者动态输出的地方，使用特殊指令`<% ... %>`。

我们来编写一个`hello.jsp`，内容如下：

```
<html>
<head>
    <title>Hello World - JSP</title>
</head>
<body>
    <%-- JSP Comment --%>
    <h1>Hello World!</h1>
    <p>
    <%
         out.println("Your IP address is ");
    %>
    <span style="color:red">
        <%= request.getRemoteAddr() %>
    </span>
    </p>
</body>
</html>
```

整个JSP的内容实际上是一个HTML，但是稍有不同：

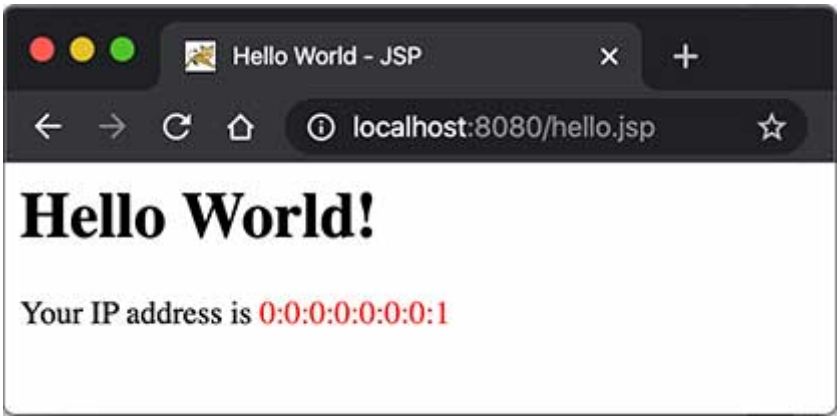
- 包含在`<%--`和`--%>`之间的是JSP的注释，它们会被完全忽略；
- 包含在`<%`和`%>`之间的是Java代码，可以编写任意Java代码；
- 如果使用`<%= xxx %>`则可以快捷输出一个变量的值。

JSP页面内置了几个变量：

- out：表示HttpServletResponse的PrintWriter；
  - out.println可以在页面上渲染文字
- session：表示当前HttpSession对象；
- request：表示HttpServletRequest对象。

这几个变量可以直接使用。

访问JSP页面时，直接指定完整路径。例如，`http://localhost:8080/hello.jsp`，浏览器显示如下：



JSP和Servlet有什么区别？**其实它们没有任何区别，因为JSP在执行前首先被编译成一个Servlet。**在Tomcat的临时目录下，可以找到一个`hello_jsp.java`的源文件，这个文件就是Tomcat把JSP自动转换成的Servlet源码：

```
package org.apache.jsp;
import ...

public final class hello_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent,
               org.apache.jasper.runtime.JspSourceImports {

    ...

    public void _jspService(final javax.servlet.http.HttpServletRequest request,
final javax.servlet.http.HttpServletResponse response)
        throws java.io.IOException, javax.servlet.ServletException {
        ...
        out.write("<html>\n");
        out.write("<head>\n");
        out.write("    <title>Hello World - JSP</title>\n");
        out.write("</head>\n");
        out.write("<body>\n");
        ...
    }
    ...
}
```

可见JSP本质上就是一个Servlet，只不过无需配置映射路径，Web Server会根据路径查找对应的`.jsp`文件，如果找到了，就自动编译成Servlet再执行。在服务器运行过程中，如果修改了JSP的内容，那么服务器会自动重新编译。

JSP高级功能

JSP的指令非常复杂，除了`<% ... %>`外，JSP页面本身可以通过`page`指令引入Java类：

```
<%@ page import="java.io.*" %>
<%@ page import="java.util.*" %>
```

这样后续的Java代码才能引用简单类名而不是完整类名。

使用`include`指令可以引入另一个JSP文件：

```
<html>
<body>
    <%@ include file="header.jsp"%>
    <h1>Index Page</h1>
    <%@ include file="footer.jsp"%>
</body>
```

JSP Tag

JSP还允许自定义输出的tag，例如：

```
<c:out value = "${sessionScope.user.name}"/>
```

JSP Tag需要正确引入taglib的jar包，并且还需要正确声明，使用起来非常复杂，对于页面开发来说，*不推荐*使用JSP Tag，因为我们后续会介绍更简单的模板引擎，这里我们不再介绍如何使用taglib。

MVC

我们通过前面的章节可以看到：

- Servlet适合编写Java代码，实现各种复杂的业务逻辑，但不适合输出复杂的HTML；
- JSP适合编写HTML，并在其中插入动态内容，但不适合编写复杂的Java代码。

能否将两者结合起来，发挥各自的优点，避免各自的缺点？

答案是肯定的。我们来看一个具体的例子。

假设我们已经编写了几个JavaBean：

```
public class User {
    public long id;
    public String name;
    public School school;
}

public class School {
    public String name;
    public String address;
}
```

在`UserServlet`中，我们可以从数据库读取`User`、`School`等信息，然后，**把读取到的JavaBean先放到HttpServletRequest中，再通过`forward()`传给`user.jsp`处理**：

```
@WebServlet(urlPatterns = "/user")
public class UserServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 假装从数据库读取:
        School school = new School("No.1 Middle School", "101 South Street");
        User user = new User(123, "Bob", school);
        // 放入Request中:
        req.setAttribute("user", user);
        // forward给user.jsp:
        req.getRequestDispatcher("/WEB-INF/user.jsp").forward(req, resp);
    }
}
```

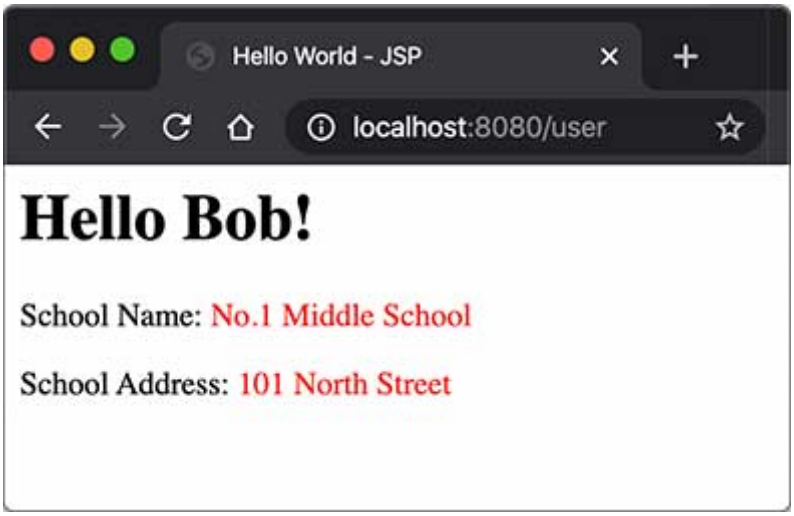
在`user.jsp`中，我们只负责展示相关JavaBean的信息，不需要编写访问数据库等复杂逻辑：

```
<%@ page import="com.itranswarp.learnjava.bean.*"%>
<%
    User user = (User) request.getAttribute("user");
%>
<html>
<head>
    <title>Hello World - JSP</title>
</head>
<body>
    <h1>Hello <%= user.name %>!</h1>
    <p>School Name:
    <span style="color:red">
        <%= user.school.name %>
    </span>
    </p>
    <p>School Address:
    <span style="color:red">
        <%= user.school.address %>
    </span>
    </p>
</body>
</html>
```

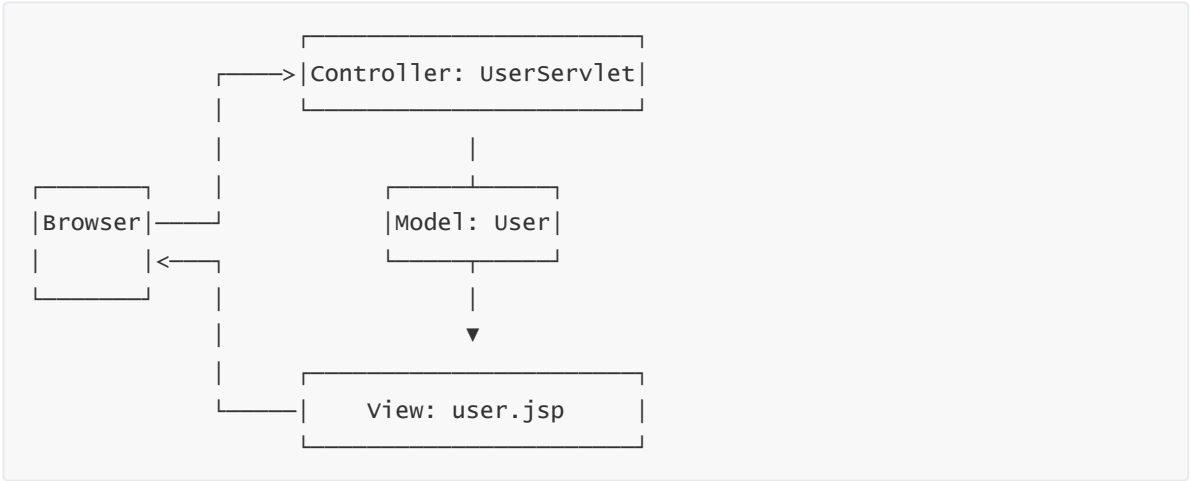
请注意几点：

- 需要展示的`User`被放入`HttpServletRequest`中以便传递给JSP，因为一个请求对应一个`HttpServletRequest`，我们也无需清理它，处理完该请求后`HttpServletRequest`实例将被丢弃；
- 把`user.jsp`放到`/WEB-INF/`目录下，是因为`WEB-INF`是一个特殊目录，Web Server会阻止浏览器对`WEB-INF`目录下任何资源的访问，这样就防止用户通过`/user.jsp`路径直接访问到JSP页面；
- JSP页面首先从`request`变量获取`User`实例，然后在页面中直接输出，此处未考虑HTML的转义问题，有潜在安全风险。

我们在浏览器访问`http://localhost:8080/user`，**请求首先由`UserServlet`处理，然后交给`user.jsp`渲染**：



我们把`UserServlet`看作业务逻辑处理，把`User`看作模型，把`user.jsp`看作渲染，这种设计模式通常被称为MVC：Model-View-Controller，即`UserServlet`作为控制器（Controller），`User`作为模型（Model），`user.jsp`作为视图（View），整个MVC架构如下：



使用MVC模式的好处是，Controller专注于业务处理，它的处理结果就是Model。Model可以是一个JavaBean，也可以是一个包含多个对象的Map，Controller只负责把Model传递给View，View只负责把Model给“渲染”出来，这样，三者职责明确，且开发更简单，因为开发Controller时无需关注页面，开发View时无需关心如何创建Model。

MVC模式广泛地应用在Web页面和传统的桌面程序中，我们在这里通过Servlet和JSP实现了一个简单的MVC模型，但它还不够简洁和灵活，后续我们会介绍更简单的Spring MVC开发。