# 1. Web开发概念

之前学习Java SE时，编写的程序是本地应用程序，符合C/S结构。Web开发就是开发网页应用程序，网页应用不仅包含用于展现网页内容和效果的HTML页面、CSS样式和JS脚本，还包含应用后端的具体功能的实现。而采用Java EE就可编写后端程序，属于B/S结构。

学习Web开发，必须先掌握Java SE知识和基本的前端知识。

## 1.1 服务器和Web服务软件

服务器就是在网络上提供服务的设备，如一台配置较好的计算机就能在网络上提供服务。

Web服务软件运行在服务器上，它的作用就是把本地的资源共享给外部访问。比如有了Web服务软件，用户就能访问服务器上的网页和文件等资源。

进行Java Web开发，也需要用合适的Web服务软件，来将我们开发好的网页应用提供给客户使用。目前常用的Java Web服务器软件有：

（1）Tomcat：Apache组织的产品，开源免费，支持大部分Java EE规范，可满足中小型应用的开发。（Java EE就是用Java开发企业级应用的一系列规范。）

（2）Jetty：可作为一个嵌入的服务器。

（3）JBoss：Redhat公司的产品。

另外，还有Resin、WebLogic和WebSphere等适合Java EE的产品。在目前学习阶段，使用的是Tomcat。

实际上学习了网络编程后，完全可以用Java SE自行实现一个简单的Web服务器软件，比如我们使用TCP协议监听本机8080端口，当有请求过来时，就用流输出一些数据。那么我们使用浏览器访问该端口时，就会得到这些数据。（必须使用可靠的TCP协议，这是显而易见的）

# 2. Tomcat服务器

开发Web应用，首先要在本机搭建一个服务器，以便在本机部署、运行和查看开发的Web应用。我们选用Tomcat。

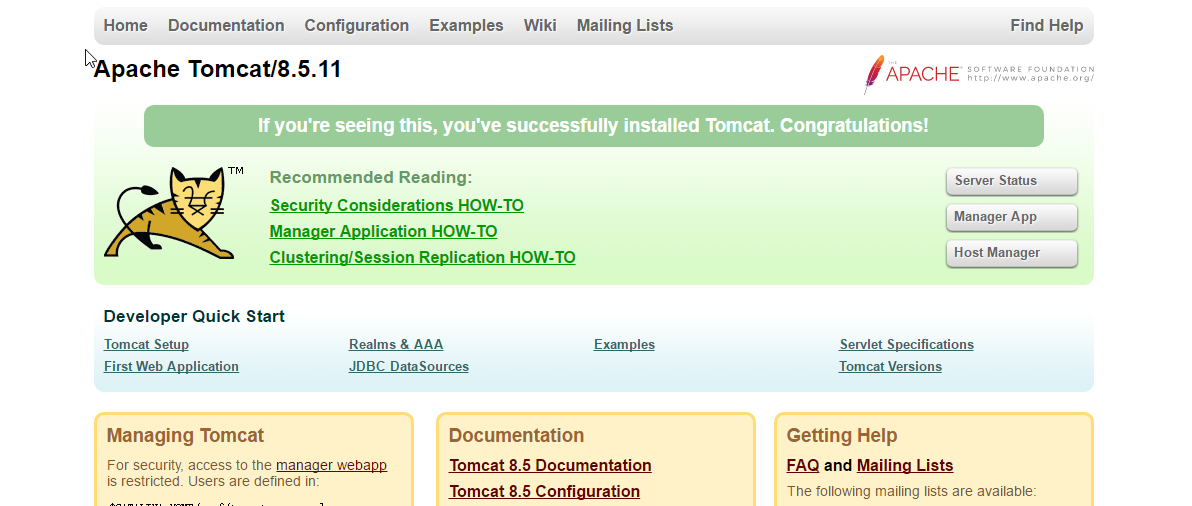
## 2.1 安装和运行

首先到apache的tomcat官网http://tomcat.apache.org/ 下载最新稳定版本的安装包。我们选择Download中的Tomcat 8，目前是8.5.11版本。在Core里面有很多版本，前两个压缩包是各平台通用的，后三个使用于Windows平台。我们选择windows-x64.zip压缩包即可。若项目在Windows平台上投入运行时，可选择最后一个安装版本，方便使用。

下载完后，直接将文件解压到某个目录中，这个自行确定。这时就完成了“安装”。我们进入Tomcat安装目录下的bin目录，里面提供了一些程序。

双击运行startup.bat即可运行Tomcat服务器软件，注意运行时，不要关闭此窗口。若想关闭Tomcat服务器，则双击运行shutdown.bat即可。

开启Tomcat后，可在浏览器中访问http://localhost:8080地址以验证Tomcat服务开启成功。若开启成功，访问此地址会顺利进入Tomcat页面。即：



因为Tomcat默认监听的端口就是本机的8080端口。

常见的Tomcat不能启动的问题：

（1）Tomcat基于Java开发，因此要确保本机Java环境安装正确，否则就会闪退。

（2）端口占用错误：8080端口可能已经被其他应用占用了。方法一就是找到占用端口的进程，杀掉此进程。方法二是修改Tomcat默认的端口：打开Tomcat安装目录下conf目录下的server.xml文件，修改Connector标签中的port属性，比如改为8090。这样就能修改Tomcat监听端口为8090，启动后，访问地址也为localhost:8090。

（3）查看系统中是否有CATALINA\_HOME这个环境变量，如果有，删除它。因为该变量的作用是设置Tomcat的根目录。如果之前安装过tomcat的Windows安装版程序，该程序会自动设置好该环境变量。这样此时启动Tomcat时，仍然会到CATALINA\_HOME设置的目录中启动Tomcat，因此导致启动不成功。不设置该变量的话，默认就启动当前目录下的tomcat。

## 2.2 Tomcat的目录结构

bin：存放可执行命令。

conf：存放配置文件；其中server.xml是核心配置文件。

lib：存放Tomcat服务器的核心类库。如果需要扩展Tomcat的功能，也可以将第三方类库复制到该路径下。

logs：存放Tomcat产生的日志文件。

temp：用于存放临时文件。

webapps：该路径用于部署Web应用。将Web应用复制在该路径下，Tomcat就会自动部署该应用。应用资源不能直接放在webapps目录下，必须在webapps中新建一个文件夹，将资源放入文件夹中，访问时要带上该文件夹名。比如将1.html放在webapps/mytest下，则访问路径是：localhost:8080/mytest/1.html。

work：tomcat的运行目录。JSP运行时产生的临时文件就存放在这里。

## 2.3 体验Tomcat的作用

我们先写一个静态HTML页面，比如叫index.html。现在目的是通过Tomcat发布此页面，让本机浏览器和本机所在局域网中的其他人能访问该页面。步骤如下：

（1）应用要放在webapps目录下，一定要新建一个文件夹再将这些资源放进去。该文件夹就相当于该应用的“应用名”。比如新建的文件夹是exp。

（2）开启Tomcat服务器，访问地址：[http://localhost:8080/exp/index.html 即可。如果你改过端口号，还要把8080改成你设置的端口。如果局域网其他人想访问此页面，只要将localhost换成你的机器的IP地址即可。](http://localhost:8080/exp/index.html即可。如果你改过端口号，还要把8080改成你设置的端口。如果局域网其他人想访问此页面，只要将localhost换成你的机器的IP地址即可。)

实际上浏览器访问的流程如下：

浏览器先找到系统的hosts文件，看里面有没有域名对应的ip地址。比如hosts里面定义了localhost的ip地址是127.0.0.1，localhost是本地域名。如果有，则直接按照hosts里对应的IP直接访问，如果没有，再向外面请求域名服务器DNS。

# 3. 动态资源开发技术：Servlet

动态资源就是用程序的来控制输出不同内容的HTML页面。Servlet是用Java语言开发动态网站的技术。比如一个页面将用户的登录信息发送给后端Servlet，Servlet中有Java代码连接数据库验证用户信息的合法性，最后根据结果返回不同内容的HTML。这些工作就是后端来做的。

## 3.1 第一个Servlet程序

Oracle只提供了Servlet的规范，具体的由各个服务器厂商实现。Tomcat的lib目录的servlet-api.jar就是实现Servlet的jar包。现在我们在项目中引入此包，完全可以在Java SE项目下开发第一个Servlet程序。

引入jar包后，新建一个Servlet类，名叫MyServlet，要求Servlet继承自HttpServlet。Servlet中可以重写doGet()和doPost()方法，分别用于处理网页的GET和POST请求。示例如下：

|  |
| --- |
| **public class** MyServlet **extends** HttpServlet {  @Override  **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) {    }   @Override  **protected void** doPost(HttpServletRequest req, HttpServletResponse resp) {    } } |

由于Servlet是在Tomcat服务器中运行的，因此并不要写main主方法。方法中有两个参数，分别是HttpServletRequest和HttpServletResponse对象，其中就用于封装网络请求和响应。

现在我们正式写好第一个Servlet类：

|  |
| --- |
| **package** com.zhang.test;  **import** javax.servlet.http.HttpServlet; **import** javax.servlet.http.HttpServletRequest; **import** javax.servlet.http.HttpServletResponse; **import** java.io.IOException; **import** java.util.Date;  **public class** MyServlet **extends** HttpServlet {  @Override  **protected void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** IOException {  *// 现在只处理GET请求  // 用response对象设置返回的响应数据   // 设置响应编码为utf-8，防止中文乱码* response.setContentType(**"text/html;charset=UTF-8"**);  *// 想浏览器输出内容* response.getWriter().write(**"你好，当前时间为："** + **new** Date());  } } |

该Servlet处理类返回了当前服务器时间。现在，需要把该类放到Tomcat中运行，并能接收浏览器请求返回相应数据。步骤如下：

（1）首先在webapps目录下新建一个文件夹，比如叫DynamicWeb，即我们的“网站”就叫DynamicWeb，然后我们需要将资源全部放在此目录下。

（2）Java Web应用的资源是有规范的，首先应该在DynamicWeb下创建一个WEB-INF文件夹，该文件夹下创建一个web.xml文件和一个classes目录。该web.xml用于配置该网络应用，而classes目录下专门用于存放Java代码生成的字节码文件。

（3）首先拿到我们MyServlet的字节码文件MyServlet.class，因为IDE会自动编译保存的Java文件，所以在IDE中很容易拿到。然后将MyServlet.class放入classes目录下，注意的是，由于MyServlet类是在包com.zhang.test中的，因此必须要在classes下新建com/zhang/test目录，然后把字节码放入test目录中，否则无效。

（4）配置web.xml文件，主要就是配置该Servlet的访问路径（映射）。例子：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?> <!-- Schema约束 -->* <**web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://xmlns.jcp.org/xml/ns/javaee"  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"  id="WebApp\_ID" version="3.1"**>  *<!-- servlet标签用于配置程序中的一个servlet -->* <**servlet**>  *<!-- 定义该Servlet的名称 -->* <**servlet-name**>MyFirstServlet</**servlet-name**>  *<!-- 指定Servlet所在的类 -->* <**servlet-class**>com.zhang.test.MyServlet</**servlet-class**>  </**servlet**>  *<!-- servlet-mapping用于配置一个Servlet的访问路径 -->* <**servlet-mapping**>  *<!-- servlet的名称，和上面的一致 -->* <**servlet-name**>MyFirstServlet</**servlet-name**>  *<!-- 指定该servlet的访问路径，就是浏览器访问的路径。一定要写上斜杠 -->* <**url-pattern**>/hello</**url-pattern**>  </**servlet-mapping**> </**web-app**> |

web.xml中指明了可以用/hello路径来访问此Servlet。

（5）最后，我们启动Tomcat，在浏览器中输入：localhost:8080/DynamicWeb/hello 即可得到响应结果。至此我们完成了第一个Servlet的开发。

## 3.2 在Eclipse中开发Servlet

### 3.2.1 在Eclipse配置Tomcat

上述的过程只是为了简单了解Servlet是如何运作的，当然实际开发中不可能这么做。IDE中对Java Web开发提供了很好的支持，能够提升开发效率。下面讲如何在Eclipse中进行开发。当然，你选择的版本一定要是Eclipse IDE For Java EE Developers版本，而不是“For SE”版本。

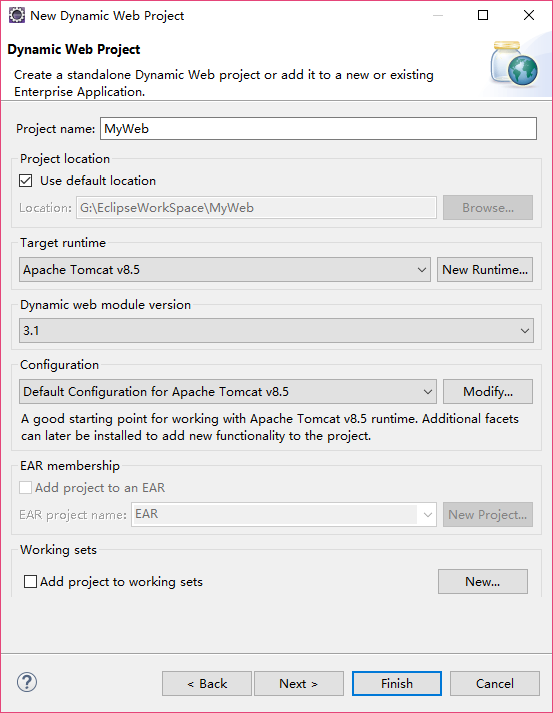
首先在Eclipse中配置好Tomcat。点击Window -> Proferences ，选择左边树状菜单中Server中的“Runtime Environments”一项。点击右侧界面的“Add...”按钮，选择Apache下的对应的Tomcat版本（这里选择v8.5），随后点击“Next”，浏览选择我们Tomcat的安装目录。最后Finish完成。

随后在Eclipse中新建一个Tomcat服务器实例。点击Window -> Show View -> Servers，显示服务器选项卡。点击Servers中的蓝色链接新建一个服务器，就选择新建一个Tomcat v8.5，其他保持默认，点击Finish完成。

最后，双击新建完成的服务器，进行如下配置：在Server Locations中选择“Use Tomcat installation”一项，并且修改Deploy Path为webapps。这样才算完成Tomcat配置。如果不进行最后这项配置，也能运行项目，但是默认不会把项目部署到Tomcat的安装路径。为了直观就这样配置，方便查看。如果以后采用了Maven管理项目，就无所谓了。

### 3.2.2 新建Java Web项目

首先新建一个Project，选择Web下的Dynamic Web Project，点击Next，输入项目名，下面各项选择和Tomcat有关的最新版本，例如：



点击Next，不变，再点击Next，把“Generate web.xml deployment description”勾选上，这样会自动帮我们创建web.xml文件。此页面的Content directory可以不改，不过以前人们喜欢改成“WebRoot”。最后点击Finish完成。

提示：如果发现点击Finish后，Eclipse卡在“installing dynamic web module facet”，则先断开网络连接，重新新建项目即可。

建好项目以后，发现项目的WebContent文件夹下和我们需要复制到Tomcat目录下的文件结构基本一致。实际上项目运行时，Eclipse会自动把工作空间中WebContent目录下的内容复制到Tomcat对应的我们的应用文件夹中。这里的WebContent就相当于一个“虚拟目录”。这样的话，进行开发部署调试就很方便。因为IDE自动把项目部署到本地服务器上。

WebContent目录中可新建资源文件和文件夹，比如HTML等，因为运行项目时，此目录下资源会被复制到Tomcat/webapps/我们的应用目录下。其中，WEB-INF/lib文件夹中专门用于存放项目中需要使用的第三方jar包。Web项目中，直接把第三方jar包复制到此目录中即可完成jar包导入，是自动的。

新建好项目，即可编写Servlet了。在项目上右键New可直接新建Servlet等，但是不推荐这样用（设置较繁杂且生成垃圾代码多），还是自己新建Class类自行编写配置。

可以按照第一个Servlet示例编写Servlet和web.xml文件。编写完毕后，选中Servers选项卡中我们新建好的服务器，右键“Add and Remove...”，把项目添加到右边中，这样就完成部署了，最后右键服务器点击“Start”即启动项目。启动完毕我们即可在浏览器中浏览了。

## 3.3 使用Servlet注解

在上述开发Servlet时，我们使用web.xml文件配置了Servlet的访问路径。每个Servlet都要在web.xml中写相应的配置，这样比较麻烦。我们之前学习过注解，可以用来简化配置。在Servlet中也可以使用注解，现在Servlet的最新版本是3.1（我们新建项目时选择的Dynamic Web module version就是3.1版本），对注解有很好的支持。

在Servlet类上使用@WebServlet注解来配置此Servlet。@WebServlet中用name属性表示此Servlet名称，用urlPatterns属性表示此Servlet的映射地址。其中urlPatterns必须要写，否则无法访问。urlPatterns类型是数组，说明可以让多个地址访问同一个Servlet。

例子：

|  |
| --- |
| **package** com.zhang.test;  **import** javax.servlet.annotation.WebServlet; **import** javax.servlet.http.HttpServlet; **import** javax.servlet.http.HttpServletRequest; **import** javax.servlet.http.HttpServletResponse; **import** java.io.IOException; **import** java.util.Date;  @WebServlet(urlPatterns = {**"/hello"**, **"/world"**}) **public class** MyServlet **extends** HttpServlet {  @Override  **protected void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** IOException {  response.setContentType(**"text/html;charset=UTF-8"**);  response.getWriter().write(**"你好，当前时间为："** + **new** Date());  } } |

之后就能通过localhost:8080/你的应用名/hello或者/world进行访问此Servlet了。注意，不能同时用注解和web.xml进行Servlet的配置。可以感觉到使用注解确实简化了配置。

# 4. Servlet处理请求和响应

## 4.1 HTTP协议

HTTP协议就是客户端（浏览器）和服务器之间传输数据的协议规范。HTTP通信基于TCP协议，它包含了HTTP请求和响应。我们用Servlet处理的也主要是请求和响应的数据。

先了解下HTTP请求。客户端会发送HTTP请求服务器网页资源，比如HTML，请求的格式和内容基本如下：

|  |
| --- |
| POST /MyApps/test.html HTTP1.1 -- 第一行是请求行。以下都是请求头（即多个键值）。  Host: localhost:8080 -- 请求的主机地址（主机:端口）  Accept: text/html,image/\* -- 浏览器接受的数据类型，包括html文本和各种格式的图像(因为用了通配符\*)  Referer: <http://www.it.org/index.html> -- 表示当前请求来自该网络地址  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/56.0.2924.76 Safari/537.36 -- 表示客户端浏览器类型等信息  Cookie: name=eric -- 浏览器保存的cookie信息  Connection: keep-alive -- 表示保持连接  --请求头结束，要有一个空行  Name=123&age=23 -- 实体内容，即传递到服务器的参数数据，是可选的 |

说明：

（1）第一行是请求行，其中说明了请求方式（GET或者POST）、请求的资源和HTTP协议版本。现在基本都会使用1.1版本，HTTP2 也会日渐流行。

（2）请求行的下面就是请求头了。请求头可有多个，每个请求头都是一个“键值对”，注意规范的格式是冒号后面要有一个空格再写值。表中列出了常见的请求头。

（3）请求头结束以后，如果有请求参数，那么需要空一行再写请求参数。各请求参数用&符号分隔。

（4）目前我们可用表单提交POST和GET请求，这点在学习HTML时已经很清楚。POST和GET就是用不同的方式向服务器传递参数数据，马上就学习如何在Servlet中得到数据并处理。两种请求区别在于：表单提交GET方式时，地址栏会将显示请求的数据，例如“/test?name=ABC”之类的，不利于保密；且GET请求内容大小不能超过1KB。而使用POST提交时，数据不会显示在地址栏中，而是在请求实体内容中传递。且POST请求内容没有大小限制。

（5）POST后面使用的是URI路径。需要注意URI和URL的区别：

URI是统一资源标记符。可用于标记任何位置的资源，包括互联网、本地文件系统和局域网的资源，比如：//192.168.14.10/book/css.pdf。

URL是统一资源定位器。比如http://localhost:8080/day/tes.html。URL用于说明如何找到该资源，比如说明了是HTTP协议，能容易的找到资源。URL是URI的子集。

同样，HTTP响应也包含了响应行、响应头和实体内容，比如：

|  |
| --- |
| HTTP/1.1 200 OK -- 响应行（下面开始是响应头）  Server: Apache-Coyote/1.1 -- 服务器名称  Content-Length: 24 -- 表示服务器发送给浏览器的数据长度  Date: Fri, 30 Jan 2015 01:54:57 GMT -- 服务器时间  Content-Type: text/html; charset=UTF-8 -- 表示服务器响应数据的类型和编码  Set-Cookie: name=eric; path=/search -- 表示服务器设置给浏览器的Cookie信息  -- 一个空行，下面是实体内容，比如HTML数据  <html>This is HTML.</html> -- 实体内容 |

需要说明的是响应行中返回的状态。200就是响应的状态码，状态码后面是对此状态码的描述。常用的状态码有：

|  |  |
| --- | --- |
| 200 | 表示响应成功 |
| 302 | 表示资源重定向 |
| 404 | 找不到资源 |
| 500 | 服务器内部错误 |

## 4.2 Servlet获得请求头和请求参数

在doGet和doPost方法中，有两个参数，其中HttpServletRequest对象req就专门用于封装请求信息，包括请求头和请求的参数等信息。

如果客户端发送的是GET请求，那么执行的就是doGet()方法，若发送的是POST请求，则会执行doPost()方法。

（1）用req获得请求行信息：

req.getMethod()：获得请求方式（POST 或GET）

req.getRequestURI()：获得请求的URI。

req.getRequestURL()，获得请求的URL。

req.getProtocol()：获得请求的协议名称。比如HTTP/1.1。

（2）用req获得请求头信息：

req.getHeader(String arg)：获取指定的请求头值。比如req.getHeader(“User-Agent”)。

req.getHeaderNames()：获得请求中所有的请求头名字。返回的是Enumeration<String>集合。通过此方法能得到所有的请求头名字，再通过getHeader()方法就能得到所有请求头键值。

（3）获取请求参数：

req.getParameter("参数名")：获取指定参数名的参数值。注意，此方法只能获取一个参数的值。有时复选框会上传通过参数的多个值，就要用下面的方法：

req.getParameterValues("参数名")：即根据参数名获取参数值数组，这样就能获得checkbox的多个值。

req.getParameterNames()：获取所有参数名称列表，返回的还是Enumeration<String>。

对于GET请求，可直接用req.getQueryString()方法获得请求中传输的字符串；

对于POST请求，需要用req.getInputStream()方法获得请求中传输的实体数据。然后用流的方式读取数据。

## 4.3 请求对象HttpServletRequest的应用

上述讲了request对象的常用方法，现在进行实际应用。

例1：获得请求头信息。根据user-agent请求头判断用户的浏览器。

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.IOException; **import** java.util.Enumeration;  **import** javax.servlet.ServletException; **import** javax.servlet.annotation.WebServlet; **import** javax.servlet.http.HttpServlet; **import** javax.servlet.http.HttpServletRequest; **import** javax.servlet.http.HttpServletResponse;  @WebServlet(urlPatterns = {**"/checkBrowser"**}) **public class** MyServlet **extends** HttpServlet {  @Override  **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  *// 首先设置响应类型和编码* resp.setContentType(**"text/html;charset=utf-8"**);  *// 获得所有请求头名称集合* Enumeration<String> headerNames = req.getHeaderNames();  *// 遍历，输出* **while** (headerNames.hasMoreElements()) {  String headerName = headerNames.nextElement();  resp.getWriter().write(**"请求头名称："** + headerName + **" 请求头值："** + req.getHeader(headerName) + **"<br>"**);  }    *// 获得user-agent请求头* String userAgent = req.getHeader(**"user-agent"**);  **if** (userAgent.contains(**"Chrome"**)) {  resp.getWriter().write(**"您使用的是Chrome浏览器"**);  } **else if** (userAgent.contains(**"Firefox"**)) {  resp.getWriter().write(**"您使用的是Firefox浏览器"**);  } **else if** (userAgent.contains(**"Triden"**)) {  resp.getWriter().write(**"您使用的是IE浏览器"**);  } **else** {  resp.getWriter().write(**"未知"**);  }  } } |

例2：判断请求的来源页面。很多页面要求用户浏览了广告后才能看本页面内容。

面对这样的需求，我们需要拿到请求头Referer的值。该请求头记录了请求的来源页面。如果没有Referer请求头或者Referer请求头的值不是广告页面，我们即可判断用户没有看过广告。

先写好广告页面（可直接在WebContent目录下写HTML，名叫ad.html）：

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <title>广告</title>  </head>  <body>  这是广告页面。<a href=*"http://localhost:8080/MyTest/view"*>点击</a>查看精彩内容。  </body>  </html> |

下面写访问地址为view的Servlet：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.IOException;  **import** javax.servlet.ServletException; // ...  @WebServlet(urlPatterns = {**"/view"**}) **public class** MyServlet **extends** HttpServlet {  @Override  **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  resp.setContentType(**"text/html;charset=utf-8"**);  *// 获得referer请求头* String originUrl = req.getHeader(**"referer"**);  **if** (**"http://localhost:8080/MyTest/ad.html"**.equals(originUrl)) {  resp.getWriter().write(**"这是精彩内容！"**);  } **else** {  resp.getWriter().write(**"未浏览广告，不给看内容"**);  }  } } |

这样我们只有在广告页面点击链接才能查看内容，否则直接访问view的话是不会显示内容的。

例3：获取请求参数。

首先写以下请求数据的页面：

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset=*"UTF-8"* />  <title>请求数据</title>  </head>  <body>  <form method=*"GET"* action=*"http://localhost:8080/MyTest/getData"*>  <p>姓名：<input name=*"name"* type=*"text"* /> </p>  <p>  爱好：  <input name=*"hobby"* type=*"checkbox"* />看书  <input name=*"hobby"* type=*"checkbox"* />跑步  <input name=*"hobby"* type=*"checkbox"* />跳舞  </p>  <p>  <input type=*"submit"* />  </p>  </form>  </body>  </html> |

其中也可改成POST请求。这里在Servlet中写好两种方式的请求处理：

|  |
| --- |
| **package** com.zhang.test;  // **import ...**;   @WebServlet(urlPatterns = {**"/getData"**}) **public class** MyServlet **extends** HttpServlet {  @Override  **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  *// 获得消息字符串：* System.***out***.println(req.getQueryString());  *// 获得name* System.***out***.println(req.getParameter(**"name"**));  *// 获得复选框多个值* System.***out***.println(Arrays.*toString*(req.getParameterValues(**"hobby"**)));  }   @Override  **protected void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** IOException {  *// 获得name* System.***out***.println(req.getParameter(**"name"**));  *// 获得复选框多个值* System.***out***.println(Arrays.*toString*(req.getParameterValues(**"hobby"**)));  } } |

页面设置GET提交时，接收参数（包括中文）是正常的。对于getQueryString()，其中的参数数据是用URL编码过的形式。网页本身是UTF-8编码，我们接收到的数据不会乱码。

但是将页面设置为POST提交时，发现中文乱码了！这是因为Tomcat默认采用ISO-8859-1对数据进行编码，而我们传递数据时是使用UTF-8的。解决办法之一是在doPost()方法一开始设置请求编码为UTF-8：req.setCharacterEncoding(“utf-8”)。另一个解决办法是将已经得到的参数数据用ISO-8859-1解码后再用UTF-8编码，但此法只能在POST中使用：

|  |
| --- |
| String name = **new** String(req.getParameter(**"name"**).getBytes(**"ISO-8859-1"**), **"utf-8"**); |

另外，在doPost()中，可以通过流来读取实体内容：

|  |
| --- |
| BufferedReader br = **new** BufferedReader(**new** InputStreamReader(request.getInputStream())); String line = **null**; **while** ( (line = br.readLine()) != **null** ) {  System.***out***.println(line); } |

但是使用流读取数据和使用getParameter()读取数据，两者只能选一，也就是说不能同时使用，若同时使用，后使用的必然得不到任何数据。

## 4.4 Servlet 获得响应头和设置响应数据

HttpServletRequest对象封装了请求相关信息，同样，HttpServletResponse就专门设置响应的相关信息，并会被传送给浏览器客户端，这样就得到了响应的数据。

我们在Servlet中设置响应的数据同样通过resp对象。主要有以下方法：

（1）响应行

resp.setStatus(int code)：设置响应的状态码。

（2）响应头

resp.setHeader(String name, String value)：设置一个响应头的键值对。

（3）实体内容

resp.getWriter.write(String data)：可响应字符实体内容，比如HTML内容等，如resp.getWriter.write(“<html><h4>标题</h4></html>”)。

resp.getOutputStream().write(data)：可响应字节实体内容，比如文件等。比如resp.getOutputStream.write(“你好”.getBytes())。

前面我们用getWriter().write()很多，但是注意，上面两个输出方法不能同时使用，否则会运行时出错。

## 4.5 响应对象HttpServletResponse的应用

例1：状态码和响应头应用。

（1）可设置404状态，用程序方式使页面“找不到”。即resp.setStatus(404)。例如：

|  |
| --- |
| **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** IOException {  resp.setContentType(**"text/html;charset=utf-8"**);  resp.setStatus(404);  resp.getWriter().write(**"404-找不到资源"**); } |

一般也会使用resp.sendError(404)，这时会使用Tomcat的404界面。

（2）页面的重定向。可设置状态码为302，设置location请求头的值为跳转路径，即可网浏览器跳转到其他页面。例如：

|  |
| --- |
| **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** IOException {  resp.setStatus(302);  resp.setHeader(**"location"**, **"http://www.baidu.com"**); } |

实际上网页重定向可用一句话实现：resp.sendRedirect(网址)。

（3）设置网页3秒后跳转到另一个页面。使用refresh响应头实现。该响应头值中可设置秒数和网址。若不设置网址，则网站会定时刷新。

|  |
| --- |
| **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** IOException {  resp.setHeader(**"refresh"**, **"3; http://www.baidu.com"**); } |

例2：响应数据的编码问题。

若发送文本内容，一般可设置响应编码为UTF-8，可在响应头设置响应类型，用setCharacterEncoding设置编码：

|  |
| --- |
| resp.setHeader(**"content-type"**, **"text/html"**); resp.setCharacterEncoding(**"utf-8"**); |

但是如果仅仅是发送文本内容，一般用我们熟悉的resp.setContentType("text/html;charset=utf-8");即可一句话搞定。然后输出实体内容。

当然，如果使用字节流的话，最好还是要指定UTF-8编码，因为系统可能使用系统默认的GBK编码，例如：

|  |
| --- |
| resp.getOutputStream().write(**"hello你好"**.getBytes(**"utf-8"**)); |

# 5. Servlet进阶

## 5.1 Servlet的缺省路径

Servlet的缺省路径是“/”，该路径是Tomcat服务器内置的一个路径，该路径也对应了一个名叫DefaultServlet的Servlet。实际上该Servlet也是在web.xml中配置的，Tomcat的conf目录中有个web.xml，默认的内容是（去掉了注释）：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"**>  *<!-- 默认的Servlet -->* <**servlet**>  <**servlet-name**>default</**servlet-name**>  <**servlet-class**>org.apache.catalina.servlets.DefaultServlet</**servlet-class**>  <**init-param**>  <**param-name**>debug</**param-name**>  <**param-value**>0</**param-value**>  </**init-param**>  <**init-param**>  <**param-name**>listings</**param-name**>  <**param-value**>false</**param-value**>  </**init-param**>  <**load-on-startup**>1</**load-on-startup**>  </**servlet**>  <**servlet**>  <**servlet-name**>jsp</**servlet-name**>  <**servlet-class**>org.apache.jasper.servlet.JspServlet</**servlet-class**>  <**init-param**>  <**param-name**>fork</**param-name**>  <**param-value**>false</**param-value**>  </**init-param**>  <**init-param**>  <**param-name**>xpoweredBy</**param-name**>  <**param-value**>false</**param-value**>  </**init-param**>  <**load-on-startup**>3</**load-on-startup**>  </**servlet**>  *<!-- The mapping for the default servlet -->* <**servlet-mapping**>  <**servlet-name**>default</**servlet-name**>  <**url-pattern**>/</**url-pattern**>  </**servlet-mapping**>   *<!-- The mappings for the JSP servlet -->* <**servlet-mapping**>  <**servlet-name**>jsp</**servlet-name**>  <**url-pattern**>\*.jsp</**url-pattern**>  <**url-pattern**>\*.jspx</**url-pattern**>  </**servlet-mapping**>    <**mime-mapping**>  <**extension**>123</**extension**>  <**mime-type**>application/vnd.lotus-1-2-3</**mime-type**>  </**mime-mapping**>  *<!-- 下面还是一堆的MIME类型配置 -->* </**web-app**> |

可以看到其中就配置了缺省路径。文件中还配置了JSP相关的Servlet等配置，以后我们会有更深刻的了解。

那么这个缺省的Servlet的作用就是用于解析web应用的静态资源文件和目录列表。例如，当浏览器访问http://localhost:8080/day/index.html这个资源时，处理的过程如下：

（1）到“day”应用目录下的web.xml文件中查找是否有匹配“index.html”的url-pattern。

（2）没有找到匹配的url-pattern，则DefaultServlet会到“day”应用目录下找一个叫“index.html”的静态文件。

（3）若找到index.html文件，则读取该文件内容，并返回给浏览器。若未找到该文件，则返回404错误页面。

即Tomcat是先查找动态资源，后查找静态资源。

## 5.2 Servlet的生命周期

Servlet的生命周期由Tomcat服务器控制，即该Servlet的对象何时创建，何时调用初始化方法，何时被销毁。

Servlet不仅有doGet()何doPost()方法，还有其他方法。其中含有四个Servlet的生命周期方法：

（1）构造方法：创建该Servlet对象时调用。默认情况下，当客户第一次请求访问此Servlet时，Tomcat才会创建对象。注意，无论有多少次请求，构造方法只会被调用一次，即Servlet是单实例的。

（2）init()方法：创建完Servlet对象后会调用此init()方法。同样，该对象在Servlet生命周期中，只被调用一次。

（3）service()方法：每当客户请求此Servlet时，就会调用service()方法，即service()会被调用n次用来响应用户。当然客户第一次访问Servlet时，还是会先创建好Servlet。

（4）destroy()方法：销毁Servlet对象时调用。当Tomcat服务器停止或者重新部署时，就会调用此方法销毁Servlet对象。同样，此方法在Servlet生命周期中只调用1次。

总结：用户第一次访问Servlet时，会创建Servlet对象并调用init()方法，以后每次请求会调用一次service()方法。最后当Tomcat要关闭时，会调用destroy()方法销毁Servlet。

为了验证上述说法正确性，我们可在自己的Servlet中重写这些方法，查看这些方法何时被调用，比如：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.IOException;  **import** javax.servlet.ServletException; **import** javax.servlet.annotation.WebServlet; **import** javax.servlet.http.HttpServlet; **import** javax.servlet.http.HttpServletRequest; **import** javax.servlet.http.HttpServletResponse;  @WebServlet(urlPatterns = {**"/test"**}) **public class** MyServlet **extends** HttpServlet {  **public** MyServlet() {  **super**();  System.***out***.println(**"创建Servlet"**);  }  @Override  **public void** init() **throws** ServletException {  **super**.init();  System.***out***.println(**"调用了init()方法"**);  }  @Override  **protected void** service(HttpServletRequest arg0, HttpServletResponse arg1) **throws** ServletException, IOException {  **super**.service(arg0, arg1);  System.***out***.println(**"调用了service()方法"**);  }  @Override  **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  System.***out***.println(**"GET请求"**);  }  @Override  **public void** destroy() {  **super**.destroy();  System.***out***.println(**"调用了destroy()销毁"**);  } } |

这样多次请求/test时，发现构造函数和init()中的输出一句只有一条，而每次请求就会输出service()的语句，当重新部署或者停止Tomcat时，就会调用destroy()方法。

## 5.3 Servlet线程安全问题

Servlet表现为单实例，这是由Tomcat容器来实现的。为了实现单例Servlet满足可能存在的高并发请求的情况，Servlet采用多线程方式调用service()提供服务。可通过以下的简单Java代码了解下单实例多线程是如何实现的（这并非实际实现代码）：

（1）HttpServlet类

|  |
| --- |
| **package** com.zhang.test;  **public class** HttpServlet {  *// 实现单例* **private static** HttpServlet *servlet* = **new** HttpServlet();  **private** HttpServlet() {}  **public static** HttpServlet getInstance() {  **return** *servlet*;  }  *// service* **public void** service() {  System.***out***.println(**"do something"**);  }  } |

（2）模拟容器的多线程Demo类

|  |
| --- |
| **package** com.zhang.test;  **public class** Demo {  **public static void** main(String[] args) {  *// 得到单例Servlet* HttpServlet servlet = HttpServlet.*getInstance*();  *// 模拟10个请求线程* **for** (**int** i = 0; i < 10; i++) {  *// 线程也可单独建一个类，然后类的构造接收Servlet对象即可。* **new** Thread() {  @Override  **public void** run() {  servlet.service();  }  }.start();  }  } } |

由于Servlet是多线程的，所以这可能引发线程安全问题。在某些特殊情形下，我们需要考虑线程安全问题。

比如现在要在程序中统计目前共有多少次的该Servlet请求。主要Tomcat不停止，就一直计数。则代码可这样写：

|  |
| --- |
| @WebServlet(urlPatterns = {**"/serve"**}) **public class** MyServlet **extends** HttpServlet {   *// 由于是单例模式，可直接在类中定义成员变量用于记录* **int visitorCount** = 0;  @Override  **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  resp.setContentType(**"text/html;charset=UTF-8"**);  *// 要在同步代码块中统计* **synchronized**(**this**) {  **visitorCount**++;  resp.getWriter().write(**"欢迎第"** + **visitorCount** + **"位访客"**);  }  } } |

即把使用到共享数据的代码块进行同步。建议不要在Servlet中使用成员变量。如果确实要使用，则必须同步。同时应该尽量缩小同步代码块的范围，避免因为同步导致并发效率降低。

## 5.4 自动加载Servlet

Servlet实例的默认创建时机为：该Servlet被第一次访问时。因此如果此Servlet类中初始化内容较多，那么用户初次访问此Servlet时将会花费较长时间。

我们可以设置在Tomcat启动时就进行某个Servlet的创建和初始化，也就是执行构造函数和init()方法。

设置方法如下：

若通过web.xml配置，则在servlet节点中配置load-on-startup标签，其中的文本写整数值。

当load-on-startup的值大于等于0时，Web容器（即Tomcat）会在启动时就加载此Servlet，若该值为负数（此值默认就为-1），则Servlet会在初次访问时加载。load-on-startup的正整数值越小，则该Servlet在Web容器启动时加载的优先级就越高。比如：

|  |
| --- |
| <**servlet**>  <**servlet-name**>Demo</**servlet-name**>  <**servlet-class**>com.zhang.Demo</**servlet-class**>  <**load-on-startup**>1</**load-on-startup**> </**servlet**> |

同样，在@WebServlet注解中也能设置loadOnStartup的值来让Servlet自动加载。比如（代码中也“重写”了构造方法和init()方法以验证Servlet被自动加载了）：

|  |
| --- |
| @WebServlet(urlPatterns = {**"/start"**}, loadOnStartup = 1) **public class** MyServlet **extends** HttpServlet {  @Override  **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** IOException {  System.***out***.println(**"Servlet已自动加载"**);  }   **public** MyServlet() {  **super**();  System.***out***.println(**"构造Servlet"**);  }   @Override  **public void** init() **throws** ServletException {  **super**.init();  System.***out***.println(**"执行init()"**);  } } |

提示：Servlet中有两个init()方法，一个是有参的，一个是无参的。若我们希望在Servlet做大量的初始化工作，那么应该重写无参的init()方法来初始化，不能重写有参的init()。因为Web容器会调用有参的init()方法，有参init()方法内部会调用无参ini()方法进行初始化，若擅自重写有有参init()方法，可能出现问题。

## 5.5 ServletConfig对象

ServletConfig对象主要用于加载Servlet的初始化参数。每个Servlet都对应一个ServletConfig对象。

首先，可在web.xml的servlet标签中设置该Servlet的初始化参数。参数可有多个，可自己定义，相当于键值对。比如给DemoServlet配置参数如下：

|  |
| --- |
| <**servlet**>  <**servlet-name**>DemoServlet</**servlet-name**>  <**servlet-class**>com.zhang.DemoServlet</**servlet-class**>  *<!-- 一个init-param标签可配置一个键值参数 -->* <**init-param**>  <**param-name**>year</**param-name**>  <**param-value**>2017</**param-value**>  </**init-param**>  *<!-- 可存在多个init-param标签来设置多个键值对 -->* <**init-param**>  <**param-name**>name</**param-name**>  <**param-value**>张三</**param-value**>  </**init-param**>  *<!-- 按照xsd约束，init-param标签需要在load-on-startup标签之前 -->* <**load-on-startup**>1</**load-on-startup**> </**servlet**> |

[同样，也可用注解实现相同的效果。@WebServlet中使用initParams属性配置初始化参数。查看代码可知initParams属性类型是WebInitParam数组，因此可配置多组参数。而WebInitParam也是一个注解，其中定义了name、value和description属性，也就是用于配置一个键值对。因此可将上述web.xml配置](mailto:同样，也可用注解实现相同的效果。@WebServlet中使用initParams属性配置初始化参数。查看代码可知initParams属性类型是WebInitParam数组，因此可配置多组参数。而WebInitParam也是一个注解，其中定义了name、value和description属性，也就是用于配置一个键值对。因此可将上述web.xml配置)改成注解：

|  |
| --- |
| @WebServlet(  urlPatterns = {**"/start"**},   loadOnStartup = 1,   initParams = {  @WebInitParam(name = **"year"**, value = **"2017"**),   @WebInitParam(name = **"name"**, value = **"张三"**)  } ) **public class** MyServlet **extends** HttpServlet {   } |

ServletConfig对象就是加载这些参数的，我们可以在Servlet中获取ServletConfig对象（实际上ServletConfig对象的创建也是在Servlet初始化时完成的）。

HttpServlet提供了getServletConfig()方法，该方法直接返回当前ServletConfig对象。因此我们可方便的获取参数。示例：

|  |
| --- |
| @WebServlet(  urlPatterns = {**"/start"**},  loadOnStartup = 1,  initParams = {  @WebInitParam(name = **"year"**, value = **"2017"**),  @WebInitParam(name = **"name"**, value = **"张三"**)  } ) **public class** MyServlet **extends** HttpServlet {  @Override  **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** IOException {  *// 获得ServletConfig对象* ServletConfig servletConfig = getServletConfig();  *// 例如获得year的键值* System.***out***.println(servletConfig.getInitParameter(**"year"**));  *// 也能获得所有的参数名称。然后获得所有的值* Enumeration<String> paramNames = servletConfig.getInitParameterNames();  **while** (paramNames.hasMoreElements()) {  String paramName = paramNames.nextElement();  System.***out***.println(**"键："** + paramName);  System.***out***.println(**"值："** + servletConfig.getInitParameter(paramName));  }  } } |

通过ServletConfig对象还能获取此Servlet的其他信息，比如getServletName()获得该Servlet的名称。

## 5.6 ServletContext对象

ServletContext叫做Servlet的上下文对象，它表示当前的web应用环境。一个web应用中只有一个ServletContext对象。ServletContext也能配置初始化参数，但与ServletConfig不同，ServletContext是一个全局的储存信息的空间，它的初始化数据是全局共享的。因此ServletContext在Tomcat开启时就会创建其对象。

可在web.xml中配置应用的全局初始化参数，是在context-param标签中配置的，例如：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"**>  <**context-param**>  <**param-name**>address</**param-name**>  <**param-value**>C盘</**param-value**>  </**context-param**>  <**context-param**>  <**param-name**>sync</**param-name**>  <**param-value**>是</**param-value**>  </**context-param**> </**web-app**> |

同样，在每个Servlet中都能拿到ServletContext对象并读取信息。Servlet也提供了getServletContext()方法得到该对象。并且ServletConfig对象也提供了此方法以便得到上下文对象。例如：

|  |
| --- |
| @WebServlet(  urlPatterns = {**"/start"**} ) **public class** MyServlet **extends** HttpServlet {  @Override  **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** IOException {  *// 获得servletContext对象* ServletContext servletContext = getServletContext();  *// 获得应用所有的树初始化参数* Enumeration<String> paramNames = servletContext.getInitParameterNames();  **while** (paramNames.hasMoreElements()) {  String paramName = paramNames.nextElement();  System.***out***.println(**"键："** + paramName);  System.***out***.println(**"值："** + servletContext.getInitParameter(paramName));  }  } } |

## 5.7 域对象和ServletContext域对象

在Java Web应用中，使用域对象可以实现在不同范围内的数据共享。即可把数据先保存在域对象中，不同域对象中的数据具有不同的数据共享范围。

ServletContext就是一个域对象。ServletContext域对象的作用范围是整个Web应用。即该域对象中的数据在本应用内是共享的，所有的Servlet中都能获取到数据。

（1）向域对象中设置（即保存）数据的方法是：

public void setAttribute(String name, Object object)；即保存的也是“键值对”，只是值可以是任意类型数据。

（2）从域对象中获取数据的方法是：

public Object getAttribute(String name)；即根据“键”得到Object值。

（3）将数据从域对象中删除的方法是：

public void removeAttribute(String name)。

应用的例子：

（1）ServletDemo1设置域对象数据：

|  |
| --- |
| @WebServlet(urlPatterns = **"/demo1"**) **public class** ServletDemo1 **extends** HttpServlet {  @Override  **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** IOException {  *// 获得servletContext对象* ServletContext servletContext = getServletContext();  *// 设置域对象数据。例如保存一个用户的姓名到servletContext域中* servletContext.setAttribute(**"userName"**, **"张三"**);  } } |

（2）ServletDemo2可获取域对象数据，因为在同一个应用中：

|  |
| --- |
| @WebServlet(urlPatterns = **"/demo2"**) **public class** ServletDemo2 **extends** HttpServlet {  @Override  **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  ServletContext servletContext = getServletContext();  *// 得到userName域对象值，强转成String* String userName = (String) servletContext.getAttribute(**"userName"**);  System.***out***.println(**"值："** + userName);  } } |

需要先访问/demo1，再访问/demo2，因为域对象是在/demo1的doGet()中设置的。若直接访问/demo2，只会输出null。

由此可体会到域对象的作用。

## 5.8 转发和重定向、request域对象

ServletContext对象中有方法：getRequestDispatcher()，其再调用forward()方法可实现请求和响应的转发。例如：

|  |
| --- |
| *// 将请求再转发给/servletDemo请求* getServletContext().getRequestDispatcher(**"/servletDemo"**).forward(req, resp); |

上述的直观效果是将请求结果再跳转到另一个页面或者请求。那么上述方法和重定向（resp.sendRedirect(String url)）有何区别呢？如下：

（1）转发时，浏览器地址栏的地址不会改变；而重定向的地址将变为重定向后的地址。

（2）转发只能转发到当前Web应用的链接资源，而重定向可跳转到任意地址。

（3）转发时，会转发响应和请求（response和request，之前我们将这两个参数简写为resp和req），转发到的页面或者请求可以拿到request域对象中的数据；而重定向无法保存request数据。

简单来说，若想转发页面间共享数据，只能用转发，而sendRedirect只是简单的页面跳转，不携带数据。

由上得知，request也是一种域对象，只是request域的作用范围只限于该转发的页面或请求之间，除此之外的地方不能访问到该request域中共享的数据。由于现在没学JSP页面，下面用request转发的例子就是一个Servlet转发到另一个Servlet，然后另一个Servlet能够拿到request域对象的数据。

（1）ServletDemo1设置request域对象（以前该参数简写为req）。

|  |
| --- |
| @WebServlet(urlPatterns = **"/demo1"**) **public class** ServletDemo1 **extends** HttpServlet {  @Override  **protected void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** IOException, ServletException {  *// 同样，使用setAttribute()* request.setAttribute(**"userName"**, **"张三"**);  *// 转发* getServletContext().getRequestDispatcher(**"/demo2"**).forward(reqest, response);  } } |

（2）ServletDemo2可获取：

|  |
| --- |
| @WebServlet(urlPatterns = **"/demo2"**) **public class** ServletDemo2 **extends** HttpServlet {  @Override  **protected void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  *// 获得域对象* String userName = (String) request.getAttribute(**"userName"**);  response.setContentType(**"text/html;charset=UTF-8"**);  response.getWriter().write(**"用户名："** + userName);  } } |

此时访问/demo1就会看到效果。

除了servletContext对象提供了getRequestDispatcher()方法，本身request也提供了该方法，因此直接使用request也可。

同样，servletContext和request对象都提供了getContextPath()方法，用于获得当前Web应用的路径，并且提供getRealPath(String path)来获得path路径对应的本机的真实路径。因为当前Web应用路径是“虚拟路径”，即对于客户来说，就是一个网络地址，而从服务器来看，当前Web应用路径在本机上是有真实路径的，比如在C盘的Tomcat路径下webapps目录中。根据不同需要使用这两个方法。一般用来操作本机时使用getRealPath()，当处理客户访问的资源时，用getContextPath()。

request对象还常用如下方法获得客户机相关信息：

（1）getRemoteAddr()：获得客户机的IP地址；

（2）getRemotePort()：获得客户机使用的端口号；

（3）getLocalAddr()：获得服务器的IP地址；

可自行再看看request提供的其他方法。

# 6 获取应用的资源文件

在开发中，不管是Java SE应用还是Java EE应用，经常会将资源文件放在项目的src目录下，比如数据库连接的配置文件等。当IDE编译Java类时，会自动把放在src目录下的文件复制一份到项目的编译目录下。

即对于Java SE项目，会直接把文件复制到IDE的bin目录下，即项目的根路径下，而对于Java EE项目，会把文件复制到WEB-INF/classes目录下，总之都是复制到所有包的根目录下。由于有这样的机制，我们使用资源文件就很方便，因为文件和项目路径是“相对路径”关系，若把项目移动到其他地方，他们之间的关系是不变的。

所以我们要知道如何获取该路径。当然，有时直接把配置文件放在某个包的路径下，比如在bin/com/zhang/test目录下。那么也需要得到路径等。还能通过方法直接获得目录下指定的文件的文件输入流，很方便load进Properties对象来使用。

（1）对于Java SE应用，使用如下方法：

|  |
| --- |
| **package** com.zhang.test;  **public class** Demo {  **public static void** main(String[] args) {  *// 获得bin目录（即src目录下移过去的）的绝对路径。toString()之前是URL对象* System.***out***.println(ClassLoader.*getSystemResource*(**""**).toString());  *// 获得bin目录下的文件的输入流* System.***out***.println(ClassLoader.*getSystemResourceAsStream*(**"db.properties"**));    *// 获得本类所在包的路径。即bin/com/zhang/test/* System.***out***.println(Demo.**class**.getResource(**""**));  *// 获得本类所在包路径中文件的输入流* System.***out***.println(Demo.**class**.getResourceAsStream(**"db.properties"**));   } } |

（2）对于Java EE应用，使用如下方法：

|  |
| --- |
| @WebServlet(  urlPatterns = {**"/start"**} ) **public class** MyServlet **extends** HttpServlet {  @Override  **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** IOException {  *// 获得classes目录的绝对路径。必须写"/"，不能写空字符串* System.***out***.println(**this**.getClass().getResource(**"/"**));  *// 获得该目录下文件的输入流* System.***out***.println(**this**.getClass().getClassLoader().getResourceAsStream(**"db.properties"**));   *// 获得当前包目录的路径。this.getClass()和写成本类名.class是一样的。* System.***out***.println(**this**.getClass().getResource(**""**));  *// 获得当前包目录中文件的输入流* System.***out***.println(**this**.getClass().getResourceAsStream(**"db.properties"**));  } } |

对于Java Web应用，还能用ServletContext中的getRealPath(String path)方法获取本应用目录下资源路径，用getResourceAsStream(String path)方法获得本应用下资源流。其中path参数传递的是相对于本应用目录（比如webapps/myapps）的路径。比如想获得项目中src下的文件（编译项目时会编译到WEB-INF/classes下），这时path传递：WEB-INF/classes/yourfile.txt。

例子：

|  |
| --- |
| @WebServlet(  urlPatterns = {**"/start"**} ) **public class** MyServlet **extends** HttpServlet {  @Override  **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** IOException {  *// 得到ServletContext对象* ServletContext servletContext = getServletContext();  *// 得到应用真实路径* System.***out***.println(servletContext.getRealPath(**""**));  *// 得到项目下的文件* System.***out***.println(servletContext.getRealPath(**"db.properties"**));  *// 得到classes下的文件* System.***out***.println(servletContext.getRealPath(**"WEB-INF/classes/db.prop"**));  *// 如果想获得流，就用getResourceAsStream()* System.***out***.println(servletContext.getResourceAsStream(**"WEB-INF/classes/db.properties"**));  } } |