第3章 Servlet基础


视频讲解

3.1 Servlet 和 JSP

Web 服务器接收到 HTTP 请求，处理完毕后会向客户端返回一个 HTTP 响应。Web 服务器接收客户端的请求有两种：一种是静态页面请求，客户端请求的页面中没有动态的内容需要处理，这些静态的页面直接作为响应返回。此时只需要能够解析 HTTP 的 Web 服务器（如 Nginx、Apache、IIS 等）即可。第二种是动态请求，客户端所请求的页面，需要在服务器端委托给一些应用程序进行处理，从而形成动态页面，最后作为 HTTP 响应返回。此时需要服务器不仅能处理 HTTP，还需要具备处理这些动态请求的能力，这种服务器称作 Web 应用服务器。之前示例中使用的 Tomcat，就是能够处理 Servlet 以及 JSP 动态页面的服务器或者称之为 Java Web 容器。

Servlet 以及 JSP 页面都是运行在服务器上的程序，并能生成动态的内容返回客户端，那么二者有什么联系和区别呢？

从技术产生的先后顺序看，Servlet 技术在前，JSP 技术在后。在早期的 Web 应用系统中，动态请求是由 Web 服务器转发给 CGI（Common Gateway Interface，公共网关接口）程序进行处理的，CGI 处理完毕后将结果拼接成 HTML 格式的文档，并返还给 Web 服务器，再通过 Web 服务器将响应返回给用户。CGI 程序一般由 C、C++、Perl 或者其他脚本语言编写，但对于每一个客户端的请求，CGI 都开启一个新的进程进行处理，对于服务器而言负担较重，执行效率低。因此 Sun 公司推出了基于 Java 的 Servlet 技术，Servlet 本质上来说是一个 Java 类，可以运行在 Tomcat 这样的容器中。对于用户的请求，Servlet 以线程的形式进行处理，执行效率更高，同时功能更为强大，对于 HTML 请求数据的提取和处理、会话跟踪、Cookie 设置等都有对应的方法。

Servlet 虽然在处理请求上非常方便，但是对于响应结果的显示却仍然采用 CGI 的方法，通过代码打印输出的方式去拼接 HTML 文档，导致如果想生成较为复杂的页面，代码量将急剧增加，同时也不便于页面整体效果的展示。

因此，Sun 公司提出了 JSP 技术，采用 HTML 模板＋嵌入 Java 代码以及标签的形式，能够简化响应页面输出的代码量，不过 JSP 的底层实现仍是基于 Servlet。在项目 Chapt_01 中，已经编写过第一个 JSP 页面，即 index.jsp。当项目运行时，第一次访问 index.jsp 页面后，该 JSP 页面编译为对应的 Servlet 类，如图 3-1 所示。Servlet 类存放在 Tomcat 服务器的 work 目录下，路径为\apache-tomcat-9.0.33\work\Catalina\localhost\Chapt_01\org\

apache\jsp。可以发现 index.jsp 已经被转化为一个 Java 类，同时也生成了对应 class 字节码文件，若再访问 index.jsp 页面，则直接读取字节码文件即可。

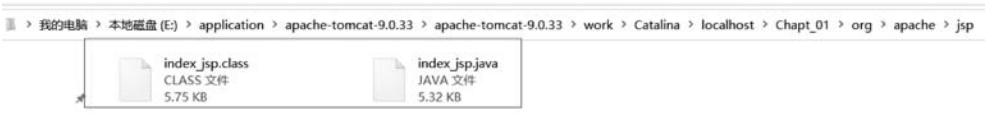

图 3-1 JSP 页面编译为对应的 Servlet 类

JSP 页面通过模板的形式方便了页面内容的输出，但如果 JSP 页面中混杂了过多的 Java 代码，将处理业务逻辑的部分都放在页面中，同样导致了代码量过大，且不利于开发人员编写和维护。因此由于两种技术各有其长处，JSP 技术的出现并没有取代 Servlet，二者可以并存合作，在开发中发挥各自的优势。

由于 Servlet 更偏向于底层的实现，因此本书先讲解 Servlet 技术的原理，然后再介绍 JSP 的使用。关于 Servlet 和 JSP 在具体开发中的使用场景，在后续章节中还会继续讨论。

3.2 Tomcat 服务器原理

在学习 Servlet 之前，先了解作为容器的 Tomcat 服务器的工作原理。

3.2.1 Tomcat 体系结构


视频讲解

Tomcat 是基于组件的 Web 应用服务器，在 2.1.2 小节介绍了 Tomcat 服务器的目录结构，在安装目录下的 conf 文件夹中，server.xml 文件是整个 Tomcat 服务器的配置文件。该配置文件给出了整个 Tomcat 服务器中各组件的设置，每个组件作为 XML 文件中的标签元素（为了方便讲解，只列出了主要的组件节点），大致结构如下：

```
<Server port = "8005" shutdown = "SHUTDOWN">
    <Listener/>…<Listener/>
    <GlobaNamingResources>…</GlobaNamingResources>
    <Service name = "Catalina">
        <Connector port = "8080" protocol = "HTTP/1.1" redirectPort = "8443"/>
        <Engine defaultHost = "localhost" name = "Catalina">
            <Host appBase = "webapps" autoDeploy = "true" name = "localhost" >
              <Context docBase = "Chapt_01" path = "/Chapt_01" reloadable = "true"
                    source = "org.eclipse.jst.jee.server:Chapt_01"/>
              <Context docBase = "Chapt_02" path = "/Chapt_02" reloadable = "true"
                    source = "org.eclipse.jst.jee.server:Chapt_02"/>
        </Host>
     </Engine>
  </Service>
</Server>
```

下面介绍 Tomcat 服务器中重要组件的作用以及相互之间的关系。

配置文件中的根节点是 Server，代表顶级服务器。该节点包含了 port="8005" shutdown="SHUTDOWN"两个属性，表示服务器通过 8005 端口号监听和关闭 Tomcat 服务器的请求。

Server 节点包含若干个 Listener、GlobaNamingResources 和 Service 等子节点。

(1) Listener 节点。该节点表示服务器运行时状态监听的配置,主要监听服务器是否会内存泄露、线程安全以及日志等信息。

(2) GlobaNamingResources 节点。该节点表示全局资源的配置,比如指定 Tomcat 服务器用户信息,这些信息存放在 conf 目录下的 tomcat-users.xml 文件中。

(3) Service 节点。该节点表示对外提供的应用服务,至少存在一个默认名称为 Catalina 的 Service 节点。Service 节点又包含若干个 Connector 和一个 Engine 组件。

① Connector:Tomcat 服务器的核心组件。其是负责客户端交互的连接器组件,负责接收用户请求并交给 Engine 组件处理,以及将处理完毕后的响应返还给客户。可以有多个 Connector,并设置该 Connector 来接收客户请求的端口号(如默认的 8080),采用的 HTTP 版本,以及 HTTPS 转发端口号等。

② Engine:Tomcat 服务器的核心组件。其是负责处理用户请求的组件,有 defaultHost 和 name 两个属性值,其中 defaultHost 表示默认的虚拟主机名称。该组件下又包含若干 Host 元素和 Realm 元素,至少有一个 Host 元素的 name 属性和 Engine 的 defaultHost 值对应。在 Host 元素下的 Context docBase 元素则定义了一个实际的 Web 项目。Relam 元素则用于安全管理的配置,一般与 tomcat-uesrs.xml 配合使用。

对于 Server、Listener、GlobaNamingResources 等元素,如果没有特殊需要,一般不需要修改其默认配置,以免影响服务器的正常运行。


视频讲解

3.2.2 Tomcat 核心组件

Tomcat 可以根据需求,通过设置不同的监听端口配置多个 Connector,当连接器指定的端口号监听到客户端发送过来的 TCP 请求后,将分别创建一个 request 和 response 对象,然后新建一个线程,将 request 和 response 对象传送给 Engine 组件,并等待处理结果,获得响应后,将响应返还给客户端。

Engine 组件可以指定多个虚拟主机 Host 组件,Host 可以配置以下 3 个属性。

(1) appBase 属性。Web 项目的部署路径,在 2.1.2 节中设置在 webapps 路径下。

(2) autoDeploy 属性。项目是否自动部署,取值为 ture 或者 false,ture 表示自动。

(3) name 属性。虚拟主机名称,取值 localhost 表示本机,刚好对应 Engine 元素的 defaultHost 的取值。

在 Host 组件下又可以具体指定 Context 组件,实际上对应着已经在 Tomcat 服务器下运行的 Web 项目。每当有新的项目部署到服务器时,都会在 Host 组件下生成一个新的 Context 元素进行配置。例如:

```
<Context docBase="Chapt_01" path="/Chapt_01" reloadable="true"
  source="org.eclipse.jst.jee.server:Chapt_01"/>
```

说明:docBase 属性设置了 Chapt_01 项目的根目录;path 属性表示项目访问的路径,即 http://localhost:8080/Chatp_01/xxx。reloadable=true,表示服务器会检测项目文件的变动,Tomcat 服务器在运行状态下会监视 WEB-INF/classes 和 WEB-INF/lib 目录下 class 文件的改动,如果监测到 class 文件有变动,服务器会自动重新加载 Web 应用。

Context 组件实际上就是运行 Servlet 的基础容器，当用户访问该 Web 应用项目时，所有的请求都需要到该 Context 环境（即该项目）下去寻找对应的 Servlet 类去处理。

3.3 Servlet 的编写

3.3.1 Servlet 的创建


视频讲解

在 Eclipse 中新建一个名为 Chapt_03 的动态 Web 项目，由于 Servlet 是一个 Java 类，所以需要在项目的 src 目录下建立，因此需在 src 目录下新建一个 com. test. servlet 的包。在 Eclipse 中可以通过模板来创建 Servlet。Servlet 的创建步骤如下所述。

（1）右击 com. test. servlet 包，在弹出的菜单中选择 New→Other 菜单项，在弹出的对话框中，找到 Web 组件下的 Servlet 选项，选择新建 Servlet 类，如图 3-2 所示，单击 Next 按钮。

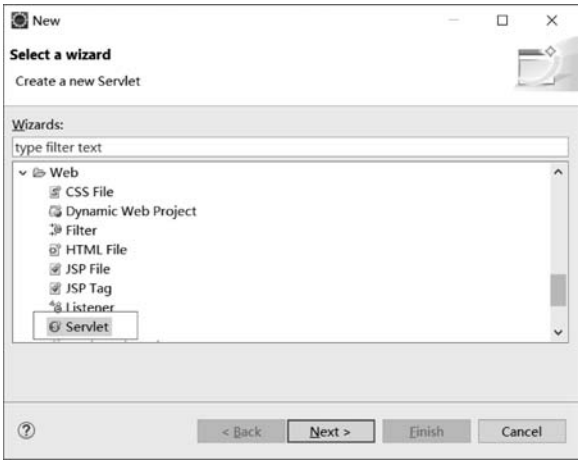

图 3-2 新建 Servlet 类

（2）Servlet 命名，如图 3-3 所示。在 Create Servlet 对话框中的 Class name 文本框中输入 FirstServlet，其他选项按照默认的即可，然后单击 Next 按钮。

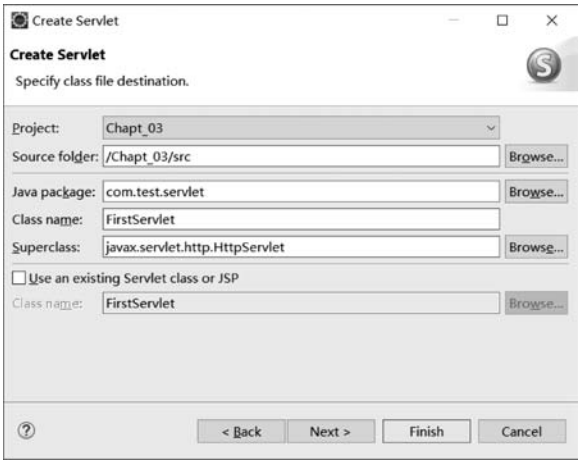

图 3-3 Servlet 命名

(3) 设置 Servlet 参数,如图 3-4 所示。在弹出的对话框中对 Servlet 的 Initialization parameters 以及 URL mappings 参数进行设置,其中 Initialization parameters 表示 Servlet 类的初始参数,URL mappings 表示访问该 Servlet 的映射路径,默认设置为/FirstServlet。如果需要设置初始参数,以及添加或者修改映射路径,就可以单击 Initialization parameters 列表或者 URL mappings 列表右边的 Add 按钮进行配置。此处先按照默认设置即可,单击 Next 按钮。

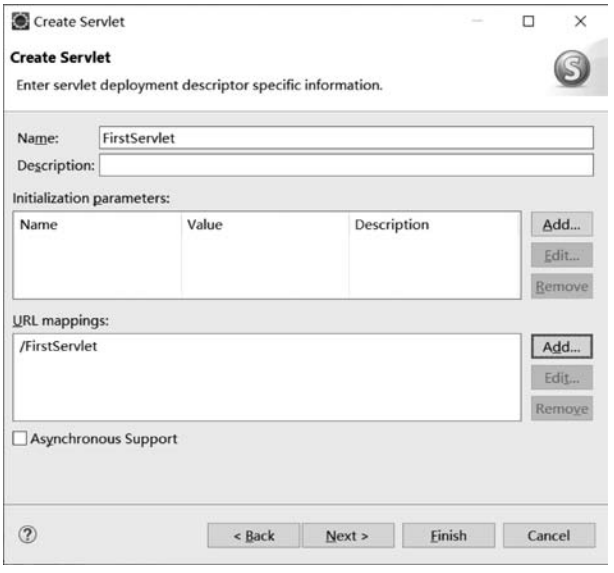

图 3-4 Servlet 参数设置

(4) 选择 Servlet 重写方法,如图 3-5 所示。在弹出的对话框中选中 FirstServlet 类需要创建的方法,模板默认会有一个父类构造器,以及继承父类的 doGet 和 doPost 方法,也可以选中其他需要继承的父类抽象方法。此处采用默认配置,单击 Finish 按钮,即完成 FirstServlet 类的创建。

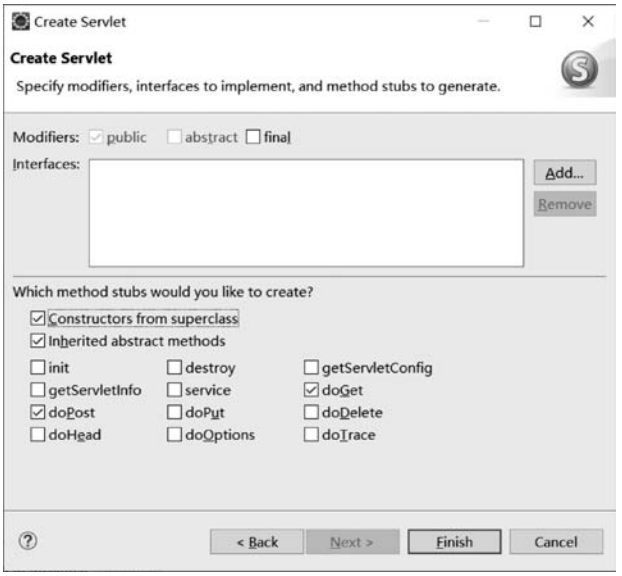

图 3-5 选择 Servlet 重写方法

此时在项目下已经生成了 FirstServlet 类，代码如下：

```
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("/FirstServlet")
public class FirstServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public FirstServlet() {
        super();
    }
    protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws
    ServletException, IOException {
        response.getWriter().append("Served at: ").append(request.
        getContextPath());
    }
    protected void doPost(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException {
        doGet(request, response);
    }
}
```

从代码中可以看出，FirstServlet 类引入了 javax.servlet.http 包中的一些类。除了父类 HttpServlet 类以外，还包括 HttpServletRequest 以及 HttpServletResponse，分别表示请求和响应，它们的实例化对象 request 和 response 分别作为 doGet 和 doPost 方法的参数。

此外，FirstServlet 类还引入了 javax.servlet.annotation.WebServlet 类，这个类是用于注解的类，可以看到在 FirstServlet 类上包含有@WebServlet("/FirstServlet")的一行注解代码。这是因为在新建项目时选择的 Dynamic Web Module Version 为 3.1，因此项目采用的是 Servlet 3.1 规范，在 Servlet 3.0 以上版本中，默认是使用注解对 Servlet 进行配置。@WebServlet("/FirstServlet")这条注解语句，其实对应了在新建 FirestServlet 时配置的 URL mapping。

如果项目选择的是 Servlet 2.5 及以下版本，就在 Servlet 新建后，需要在项目的 web.xml 配置文件的< web-app >节点下，编写如下代码以完成 FirstServlet 的配置。

```
<servlet>
    <servlet-name>FirstServlet</servlet-name>
    <servlet-class>com.test.servlet.FirstServlet</servlet-class>
  </servlet>
 <servlet-mapping>
    <servlet-name>FirstServlet</servlet-name>
     <url-pattern>/FirstServlet</url-pattern>
 </servlet-mapping>
```

其中,servlet 节点下有 servlet-name 和 servlet-class 两个元素,其值分别对应 Servlet 设置的名称和对应的具体类; servlet-mapping 节点下有 servlet-name 和 url-pattern 两个元素,配置了 FirstServlet 的访问路径。

通过模板创建 Servlet,只需要在界面中设置参数,Eclipse 就会自动生成对应的配置信息。无论采用注解还是在 web.xml 中配置,效果都是等同的。而且相同的 Servlet 配置只能选取一种方式,重复配置将会报错。当然,也可以采用手动的方式进行编写和修改,此时需要注意对应的配置语法和格式。在本书的演示中,均采用 Servlet 3.1 规范的注解方式。


视频讲解

3.3.2 Servlet 的运行

在创建完 FirstServlet 后,可进行如下操作来运行 Servlet。

(1) 右击 Chapt_03 项目,在弹出的菜单中选择 Run As→Run on Server,选择 Tomcat 9 服务器,单击 Finish 按钮,此时 Chapt_03 项目被部署到服务器中。

(2) 打开浏览器,输入网址 http://localhost:8080/Chapt_03/FirstServlet,FirstServlet 运行效果如图 3-6 所示。

图 3-6 FirstServlet 运行效果

为什么会显示这样一行文本信息呢? 实际上在访问 FirstServlet 时,Tomcat 服务器按照以下步骤进行处理。

(1) 该请求中使用端口号为 8080,因此会被一直监听 8080 端口号的 Connector 组件获取。

(2) Connector 组件把请求交给 Engine 组件处理,并等待回应。

(3) Engine 查找 Host 组件,找到匹配名字为 localhost 的虚拟主机。

(4) 在 localhost 主机上,查找 Context 组件,匹配到名字为 Chapt_03 的应用。

(5) 根据请求路径/FirstServlet,在 Chapt_03 下查找 URL mapping 配置,找到对应的 FirstServlet 类去处理。

(6) 构造 HttpServletRequest 对象和 HttpServletResponse 对象,作为参数传送给 FirstServlet 的 doGet()方法,处理完毕后,将结果封装到 HttpServletResponse 对象中。

(7) Context 将 HttpServletResponse 响应返回给 Host。

(8) Host 将响应返回给 Engine。

(9) Engine 将响应返回给 Connector。

(10) Connector 将响应结果返回给浏览器客户端。

从以上步骤看,最终页面的显示结果是来自 FirstServlet 的 doGet()方法,在方法体内部只有一条语句:

```
response.getWriter().append("Served at:").append(request.getContextPath());
```

说明：response 对象的 getWriter()方法获取了一个输出流对象，向客户端进行文本的输出，后面的 append()方法表示文本的追加输出，第二个 append()方法里的参数，由 request 对象通过 getContextPath()方法获取，表示请求的上下文 Context 对象路径，即 /Chapt_03。因此，最终输出为 Served at: /Chapt_03。

3.3.3 Servlet 的运行机制

当 Servlet 运行后，最终也会编译成字节码文件，存放在 Tomcat 服务器对应项目目录下，FirstServlet 的字节码文件可以在 Chapt_03\WEB-INF\classes\com\test\servlet\路径下找到。那么当每次运行 Servlet 时，都会创建一个实例化对象吗？实际上在默认情况下，Servlet 是以单例多线程的形式运行的。下面通过例 3-1 演示 Servlet 运行状态。


视频讲解

【例 3-1】 Servlet 运行状态。

在 Chapt_03 项目下再新建一个名为 SecondServlet 的 Servlet 类，通过注解@WebServlet("/SecondServlet")，设置其映射路径为/SecondServlet，然后在其构造函数和 doGet()方法中编写代码如下：

```
public SecondServlet() {
        System.out.println("SecondServlet 对象被实例化");
    }
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
        ServletException, IOException {
         System.out.println("SecondServlet 对象 doGet 被执行");
     }
```

注意，新建 Servlet 或者 JSP 需要重启服务器，项目重新部署后才能访问。为模拟不同客户端访问同一个 Servlet 的场景，首先通过 Firefox 浏览器访问 SecondServlet，然后观察 Eclipse 的 Console 输出内容，首次运行 SecondServlet 后的输出结果如图 3-7 所示。

图 3-7 首次运行 SecondServlet 后的输出结果

在不关闭服务器的情况下，使用 Google 浏览器，再次访问 SecondServlet 后的输出结果如图 3-8 所示。

由此可见，只有第一次访问 Servlet 时，运行了构造函数和 doGet()方法，第二次访问只执行了 doGet()方法，因此只有第一次运行时创建了对象，再次访问时并没有再次实例化对象。

图 3-8　再次访问 SecondServlet 后的输出结果

Servlet 是以多线程的方式去处理每一个请求的。即使多个用户访问同一个 Servlet 对象,服务器也会各自分配一个线程去运行 doGet()方法。

在默认情况下,由于 Servlet 采用单例模式,因此存在线程安全方面的隐患,一般不要在类中直接定义成员变量。当然也可以通过实现 SingleThreadModel 接口,让每次请求都初始化一个 Servlet 去处理,此种方式本书暂不讨论。

3.3.4　Servlet 与生命周期

1. Servlet

在利用模板新建 Servlet 类时,默认需要继承 HttpServlet 这个抽象类。HttpServlet 又是继承于 GenericServlet 这个抽象类,而 GenericServlet 抽象类又实现了 Servlet 以及 ServleConfig 两个接口。因此 Servlet 可以继承或者重写一些父类方法,这些方法将伴随着 Servlet 的整个生命周期。

(1) init(ServletConfig config):该方法继承于 GenericServlet 类,是实现了 Servlet 接口声明的 init()方法。该方法在 Servlet 类被加载后被调用,ServletConfig 接口对象作为参数传递进来,从而可以获取一些初始化参数。如果有特殊初始参数配置方面的需求,可以重写该方法。

(2) service(HttpServletRequest request,HttpServletResponse response):该方法根据 request 对象的 getMethod()方法获取请求采用的方式,再去调用对应的 do×××()方法。

(3) do×××(HttpServletRequest request, HttpServletResponse response):包括 doGet()、doPost()、doPut()、doDelete()、doHead()、doTrace()、doOptions()方法。这些方法处理不同请求方式的 HTTP 请求。

(4) destroy():当 Servlet 消亡时会调用该方法。当有特殊需求时,如需要清理某些设置及参数时可以重写该方法。

2. Servlet 生命周期

Servlet 生命周期的过程:从第一次加载时调用 init()方法,接着调用 service()方法获取请求采用的方式,然后调用对应的 do×××()方法,执行该方法完毕后返回响应的结果。当 Servlet 要消亡时(如关闭了服务器),则调用 destroy()方法。

在实际开发中,一般只需要根据请求方式重写对应的 do×××()方法即可。其中使用最多的是 doGet()和 doPost()方法,分别用于处理 GET 和 POST 类型的请求。

3.4 Servlet 处理请求与响应

3.4.1 doGet()与 doPost()方法

doGet()和 doPost()方法分别处理 GET 和 POST 两种发送方式的请求，两种方法的应用场景有所区别。

GET 方式一般针对页面及资源的请求。如访问超链接或者通过 URL 进行参数传值，以及表单默认的提交，均采用 GET 方式进行请求。

POST 方式一般用于向服务器提交数据，例如当表单 method 属性设置为 POST 时，则表单采用 POST 方式进行提交。

GET 方式传递的值直接放在请求行中，与网址内容一起进行编码。POST 传递的值则放在请求体中。

doGet()和 doPost()方法都包含 HttpServletRequest 和 HttpServletResponse 类型的参数，通过 request 对象获取请求参数，进行处理后，再利用 response 对象返回响应。在编写代码时，只需在 doGet()方法体内编写处理请求的代码即可，然后在 doPost()方法内调用 doGet()方法。

3.4.2 rqequest 基本信息的获取

request 对象提供了下面的方法用于获取请求中的一些重要信息。

(1) String getMethod()方法：获取请求方式，如 GET 或者 POST。

(2) String getRequestURI()方法：获取请求的 URI(Uniform Resource Identifier，统一资源标志符)。

(3) String getProtocol()方法：获取请求采用的协议。

(4) String getServerPort()方法：获取请求服务器端口号。

(5) String getServerName()方法：获取请求服务器的名称。

(6) String getContextPath()方法：获取请求的上下文路径。

(7) String getRemoteAddr()方法：获取发送请求的客户端 IP 地址。

以上信息在某些应用场景中需要用到，例如获取上下文路径可以用于绝对地址的拼接，而获取客户端 IP 地址则可以记录请求日志信息，甚至可以设置黑名单禁用部分 IP 地址。下面通过例 3-2 演示 request 对象基本信息的获取过程。

【例 3-2】 request 对象基本信息的获取过程。

新建一个 Servlet 类，取名为 RequestInfoServlet，通过注解 @ WebServlet("/RequestInfoServlet")设置映射路径为/RequestInfoServlet，然后在 doGet()方法中编写以下代码。


视频讲解

```
protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
String method = "method:" + request.getMethod() + "\r\n";
String protocol = "protocol:" + request.getProtocol() + "\r\n";
String servername = "servername:" + request.getServerName() + "\r\n";
```

```
String port = "port:" + request.getServerPort() + "\r\n";
String requestpath = "contextpath:" + request.getContextPath() + "\r\n";
String uri = "request URI:" + request.getRequestURI() + "\r\n";
String ipaddress = "ip address:" + request.getRemoteAddr();
PrintWriter out = response.getWriter()
out.append(method).append(protocol).append(servername).append(port).
append(uri).append(requestpath).append(ipaddress);
out.close();
    }
```

RequestInfoServlet 的 doGet()方法中使用了 request 的相应方法获取基本信息。访问 RequestInfoServlet 后，request 对象的基本信息如图 3-9 所示。

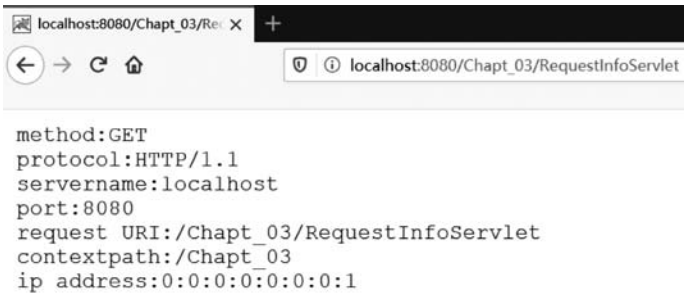

图 3-9　request 对象的基本信息

Servlet 输出时利用 response 对象使用 getWriter()或者 getOutputStream()方法以获取输出流，二者互斥不能混用。例子中使用的 PrintWriter 对象用于向客户端输出字符流，包括以下 5 个常用的方法。

(1) void print()方法：输出文本信息后不换行。

(2) void println()方法：输出文本信息后换行。

(3) void append()方法：和 print()类似，但方法可以直接追加，例 3-2 中即采用此方法，因此在相应变量后添加了\r\n 进行换行操作。

(4) void flush()方法：输出缓冲区数据，在客户端输出后清除缓冲区数据。

(5) void close()方法：关闭输出流。

3.4.3　URL 传值数据的获取

超链接是网页中常见的元素，单击超链接可以跳转到指定的 URL，该地址可以是服务器外部网址，也可以是服务器内部地址。同时超链接后面可以附带参数一同发送给服务器，从而实现页面之间信息的传递。单击超链接是通过 GET 方式提出请求，下面通过例 3-3 演示通过超链接进行 URL 传递参数以及服务器通过 Servlet 获取后再输出到客户端页面中的过程。


视频讲解

【例 3-3】 通过超链接进行 URL 传递参数。

(1) 在项目的 WebContent 下新建一个 HTML 页面，取名为 hyperlink.html，在< body >标签体内部添加一个超链接，代码如下：

```
<a href = URLServlet?a = hello&b = world>通过超链接进行 URL 传值</a>
```

该超链接指向 URLServlet，超链接指向的 URL 后面附带两个参数，访问 hyperlink.html 页面如图 3-10 所示。

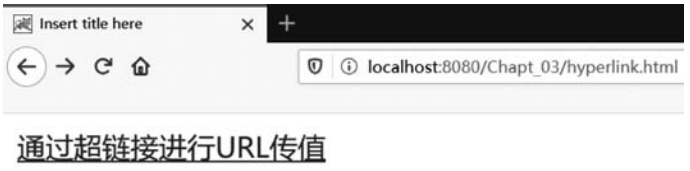

图 3-10 访问 hyperlink.html 页面

注意，超链接的 href 地址前没有加反斜杠(/)，表示采用的是相对路径，访问的是/Chapt_03/路径下的 URLServlet。如果地址前加了反斜杠，就表示使用的是绝对地址，此时必须加项目路径/Chapt_03/，代码如下：

```
<a href = /Chapt_03/URLServlet?a = hello&b = world>通过超链接进行 URL 传值</a>
```

访问路径的写法是初学时容易犯错的地方，尤其是超链接的 href 属性以及表单提交的 action 的写法，如果路径有误，则页面会报 404 错误，此时应排查采用的是相对还是绝对路径。

(2) 新建一个 Servlet，取名为 URLServlet，通过注解@WebServlet("/URLServlet")设置映射路径为/URLServlet，然后在 doGet()方法中编写代码如下：

```
String a = request.getParameter("a");
String b = request.getParameter("b");
PrintWriter out = response.getWriter();
out.println("parameter a:" + a);
out.println("parameter b:" + b);
```

单击 hyperlink.html 中的超链接，页面跳转到 URLServlet，获取超链接和 URL 传递的参数，如图 3-11 所示。

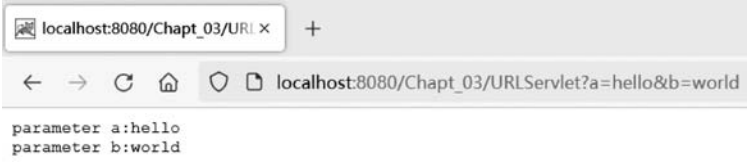

图 3-11 获取超链接和 URL 传递的参数

此时参数附加在 URL 后面，并出现在地址栏中，说明超链接的确是通过 GET 方式进行的请求。在 URLServlet 类的 doGet()方法中，使用了 request 对象的 getParameter(String name)方法。该方法的作用就是通过参数名来获取对应的参数值。该方法使用较为频繁，在获取表单提交的数据时也会采用。

3.4.4 表单中单值元素数据的获取

表单提交是 Web 应用中常见的功能,本节介绍 Servlet 处理表单中单一元素数据的方法。单值元素是指该表单元素提交数据给服务器时,只包含一个变量。表单中的单值元素包括文本框、密码框、单选按钮、下拉框以及多行文本框等。下面通过例 3-4 演示 Servlet 获取表单中单值元素的数据并处理的操作步骤。


视频讲解

【例 3-4】 Servlet 获取表单中单值元素的数据。

(1) 新建一个 HTML 页面,取名为 single.html,在< body >标签体内部编写一个包含上述元素的表单,代码如下:

```
< form action = "GetSingleServlet" method = "post">
    用户名:< input type = "text" name = "username" >< br >
    密    码:
    < input type = "password" name = "password" >< br >
    性别(单选):
    < input type = "radio" name = "gender" value = "male"
    checked = "checked" >男
    < input type = "radio" name = "gender" value = "female">女< br >
    所在区域:< select name = "country">
                < option value = "China">中国</option >
                < option value = "USA">美国</option >
                < option value = "GB">英国</option >
              </select >< br >
    请输入个人信息:< br >
    < textarea name = "information" rows = "5" cols = "30"></textarea >< br >
    < input type = "submit" value = "提交" >
    < input type = "reset" value = "重置" >
</form >
```

说明:表单 action 属性的值表示表单提交后,所有元素数据由 GetSingleServlet 处理,注意此时采用的是相对路径的写法;method 属性的值表示提交方式,默认为 GET 方式,本例使用 POST 方式进行提交。

(2) 新建 GetSingleServlet,通过注解@WebServlet("/GetSingleServlet")设置映射路径为/GetSingleServlet,在 doGet()方法中的代码如下:

```
String username = request.getParameter("username");
String password = request.getParameter("password");
String gender = request.getParameter("gender");
String country = request.getParameter("country");
String information = request.getParameter("information");
PrintWriter out = response.getWriter();
out.println("username:" + username);
out.println("password:" + password);
out.println("gender:" + gender);
out.println("country:" + country);
out.println("information:" + information);
```

与获取 URL 传值方式一样，使用 request 对象的 getParameter()方法，以表单中 input 元素的 name 属性为参数，获取 input 元素的 value 属性值，从而得到表单元素提交的数据。

(3) 打开浏览器访问 single.html 并填写表单数据，如图 3-12 所示。

图 3-12 访问 single.html 并填写表单数据

单击“提交”按钮将表单数据提交给 GetSingleServlet 处理，获取单值元素并输出到页面，如图 3-13 所示。

图 3-13 获取单值元素并输出到页面

3.4.5 表单中多值元素数据的获取

表单中如果有多个元素的 name 属性值相同，当表单提交时，这些同名元素将以数组的形式向服务器发送数据，这样的元素称为多值元素。典型的多值元素包括复选框、多选列表框以及其他同名元素组合。下面通过例 3-5 演示 Servlet 获取表单中多值元素的数据并处理的操作步骤。

【例 3-5】 Servlet 获取表单中多值元素的数据并处理。


视频讲解

(1) 新建一个 HTML 页面，取名为 multiple.html，在< body >标签体内部编写一个包含上述元素的表单，代码如下：

```
<form action = "GetMultipleServlet" method = "post">
  勾选你的兴趣爱好(可多选):<br>
  <input name = "hobbies" type = "checkbox" value = "reading">阅读
  <input name = "hobbies" type = "checkbox" value = "dancing">跳舞
  <input name = "hobbies" type = "checkbox" value = "singing">唱歌
  <input name = "hobbies" type = "checkbox" value = "sport">运动<br>
```

```
    选择精通的语言(可多选):<br>
    <select name="languages" multiple>
      <option value="Chinese">汉语</option>
      <option value="English">英语</option>
      <option value="French">法语</option>
      <option value="Russian">俄语</option>
      </select><br>
      填写你擅长的其他技能:<br>
      技能 1:<input type="text" name="skills"><br>
      技能 2:<input type="text" name="skills"><br>
      技能 3:<input type="text" name="skills"><br>
      <input type="submit" value="提交" >
      <input type="reset" value="重置" >
</form>
```

说明：multiple.html 页面中的下拉框<select>有一个属性 multiple，表示这个下拉框和 CheckBox 一样，可以多选。另外还有 3 个 name 属性值都是 skills 的文本框，这是一个文本框的组合。表单提交方式为 POST，提交给 GetMultipleServlet 去处理。

由于多选框、多选下拉列表和组合文本框的 value 取值有多个，因此需要采用 request 对象的 getParameterValues()方法，参数仍然是这些多值元素的 name 属性值，但该方法返回的不再是单一的字符串，而是一个 String 类型的数组。

(2) 新建一个 Servlet 类，取名为 GetMultipleServlet，通过注解 @ WebServlet ("/GetMultipleServlet")设置映射路径为/GetMultipleServlet，在 doGet()方法中编写代码如下：

```
String[] hobbies = request.getParameterValues("hobbies");
String[] languages = request.getParameterValues("languages");
String[] skills = request.getParameterValues("skills");
PrintWriter out = response.getWriter();
out.println("your hobbies include:");
for(int i = 0;i < hobbies.length;i++) {
out.println(hobbies[i]);
}
out.println();
out.println("you can speak:");
for(int i = 0;i < languages.length;i++) {
out.println(languages[i]);
}
out.println();
out.println("your skills include:");
for(int i = 0;i < skills.length;i++) {
out.println(skills[i]);
}
```

以上代码通过 getParameterValues()方法获取多值元素数据，通过 for 循环将数组中的信息输出到客户端。

(3) 打开浏览器,访问 multiple.html 页面并填写表单数据,如图 3-14 所示。

图 3-14 访问 multiple.html 页面并填写表单数据

单击“提交”按钮,将表单数据提交给 GetMultipleServlet 处理,获取多值元素并输出到页面,如图 3-15 所示。

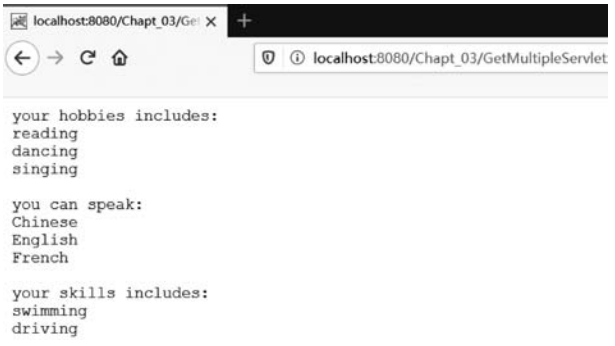

图 3-15 获取多值元素并输出到页面

3.5 中文传输乱码问题


视频讲解

例 3-4 演示了 Servlet 处理表单的操作,URL 传值以及表单提交的数据都是英文和数值。如果提交的表单数据是中文,那么是否能正确显示呢? 重新访问 single.html 页面,表单输入中文数据并提交,如图 3-16 所示。

跳转到 Servlet 处理后,获取 POST 方式提交的中文参数,如图 3-17 所示,此时出现了中文参数乱码问题。

如果提交方式为 GET,是否也会有这个问题呢? 修改 single.html 中 form 的 action 属性,修改为 GET 并重新提交。跳转到 Servlet 处理后,获取 GET 方式提交的中文参数如图 3-18 所示,同样出现了中文参数乱码问题。

实际上,这个例子中的参数是由客户端发出 request,到服务器由 Servlet 处理,然后 response 输出到客户端显示的。之所以出现乱码,是由于中文参数在 request、response 以

图 3-16　表单输入中文数据并提交

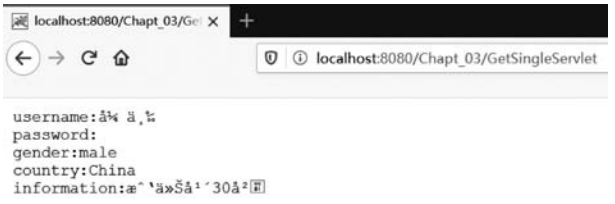

图 3-17　获取 POST 方式提交的中文参数

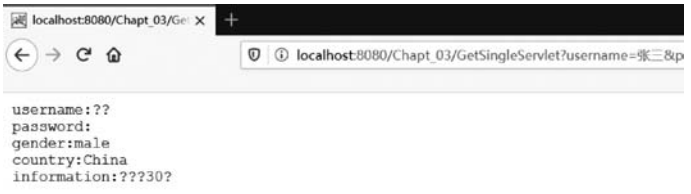

图 3-18　获取 GET 方式提交的中文参数

及最终客户端浏览器编码的方式不同而造成的。

3.5.1　请求参数编码

下面分 GET 和 POST 两种方式讨论请求参数的编码问题。

1. GET 方式

其传递的参数是放在 HTTP 请求行中,并与网址内容一起进行编码后,再传递给服务器进行处理。在 Tomcat 服务器中有一个名为 URICoding 的参数,即用于指定其请求地址的编码方式。在 Tomcat 8 以上版本中,这个参数的默认值是 UTF-8,由于 UTF-8 编码支持中文,因此通过 GET 方式传递参数到服务器并输出就不会出现乱码的情况。在 GetSingleServlet 中添加如下代码,用于输出 request 传递参数到服务器后的内容。

```
System.out.println(username);
System.out.println(information);
```

重新提交表单，在 Console 中可以看到中文参数显示正常且没有乱码。中文参数通过 GET 方式传递到服务器的效果，如图 3-19 所示。

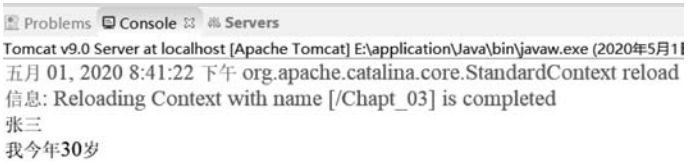

图 3-19 中文参数通过 GET 方式传递到服务器的效果

在 Tomcat 7 及以下版本中，这个参数默认为 ISO8859-1，这可以通过修改 server.xml 文件中的 Connector 组件部分的配置来进行修改默认。其代码如下：

```
<Connector connectionTimeout="20000" port="8080"
protocol="HTTP/1.1" redirectPort="8443" URIEncoding="UTF-8" />
```

如果不修改服务器的 URICoding 编码方式，就需要对 GET 方式传递的每个参数都单独转换编码格式，例如：

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

2. POST 方式

将表单提交方式修改为 POST 方式，重新提交表单，利用 Console 进行观察，发现出现了中文乱码。中文参数通过 POST 方式传递到服务器的效果如图 3-20 所示。

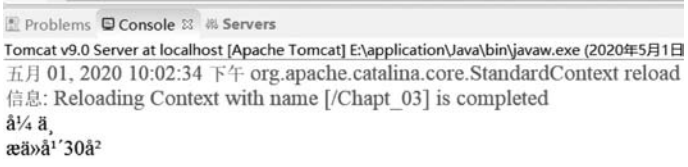

图 3-20 中文参数通过 POST 方式传递到服务器的效果

由于通过 POST 方式提交的数据是存放在 HTTP 的请求体中的，而服务器对 POST 请求的数据编码的默认方式不是 UTF-8，因此会在服务器端出现中文乱码。为此可以重新指定服务器对请求体的编码方式来避免上述问题，修改 GetSingleServlet 类的代码，在使用 getParameter()方法获取参数前，加上如下一行代码，用于指定 POST 方式提交参数的编码格式。

```
request.setCharacterEncoding("UTF-8");
```

再重新提交表单后，后台输出中文参数为正常。

3.5.2 响应编码

当 Servlet 处理完毕后，通过 response 对象进行中文的输出时，服务器也有默认的编码方式，

可以通过 response.getCharacterEncoding()方法获取,同时也提供了 setCharacterEncoding()方法用于更改编码方式。在 GetSingleServlet 类中,添加代码如下:

```
//获取 response 默认编码
System.out.println(response.getCharacterEncoding());
//设置 Tomcat 的 response 编码方式为 UTF-8
response.setCharacterEncoding("UTF-8");
//查看修改后的 response 编码方式
System.out.println(response.getCharacterEncoding());
```

注意,修改编码方式的语句需要放在执行输出的语句之前。重新提交表单,在 Console 中观察 response 编码方式的输出结果,如图 3-21 所示。

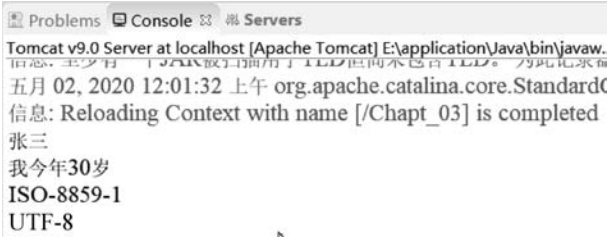

图 3-21　response 编码方式的输出结果

3.5.3　客户端编码

在修改了 response 编码方式后,中文输出是否不再乱码呢? 然而在查看页面后发现,仍然是乱码。这是因为 setCharacterEncoding()方法只是设定了响应信息的编码格式,但输出到客户端时,文本信息最终显示的编码方式是由客户端浏览器决定的。在 GetSingleServlet 输出页面下,按 F12 键,打开开发者工具,在控制台选项卡下,查看中文乱码提示信息,如图 3-22 所示。

图 3-22　中文乱码提示信息

因此当要输出中文时,还要告诉浏览器以何种编码方式进行显示。可以使用 response 的 setHeader()或者 setContentType()方法设置响应输出的信息类型以及编码方式。再次修改 GetSingleServlet 类的代码,在输出语句前添加代码如下:

```
response.setHeader("Content-type","text/plain;charset=UTF-8");
```

或者添加代码如下：

```
response.setContentType("text/plain;charset=UTF-8").
```

这两句代码的作用相同，都是通过设置 HTTP response 响应头的方式告诉客户端浏览器：输出信息的格式是纯文本类型，编码方式采用 UTF-8。重新提交表单，修改响应头后中文输出正常，如图 3-23 所示。

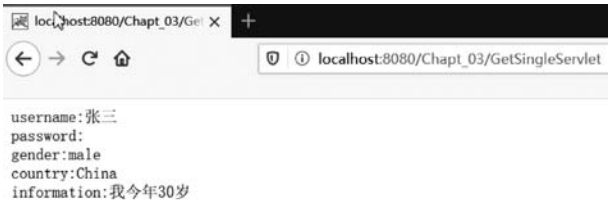

图 3-23 修改响应头后中文输出正常

setContentType()方法的第一个参数是指定访问资源的类型，比如 text/html 表示 HTML 格式，image/gif 表示 GIF 格式的图片，而 application/pdf 表示 PDF 格式。关于资源类型代码，可以查询 https://www.w3cschool.cn/http/ahkmgfmz.html。

setHeader()方法能够设置 HTTP 响应的很多参数，非常实用，在后续章节中还会介绍其用法。

3.6 Servlet 生成 HTML 页面

在上述的例子中，Servlet 的响应都是输出普通的文本类型的字符串，页面输出的是纯文本信息。那么如何通过 Servlet 输出一个由 HTML 元素组成的页面呢？下面通过例 3-6 来演示使用 Servlet 输出 10 以内的加法表格的操作步骤。


视频讲解

【例 3-6】 使用 Servlet 输出 10 以内的加法表格。

(1) 新建一个 Servlet，取名为 HtmlServlet，通过注解@WebServlet("/HtmlServlet")设置映射路径为/HtmlServlet，在其 doGet()方法中编写代码如下：

```
PrintWriter out = response.getWriter();
out.println("<html><head>");
out.println("<style type='text/css'>");              //引入内嵌 CSS 样式
out.println("table{border-collapse:collapse;}");     //设置表格为单一边框
out.println("td{border:1px solid black;}");          //单元格边框黑色实线宽度 1px
out.println("</style></head>");
out.println("<body><table>");
for(int i=1;i<=9;i++) {                              //for 循环语句输出包含加法算式的单元格
  out.println("<tr>");
  for(int j=1;j<=i;j++) {
      int add1 = i-j+1;
      int add2 = j;
      int sum = add1 + add2;
```

```
            out.println("< td >" + add1 + " + " + add2 + " = " + sum + "</td >");
        }
        out.println("</tr >");
    }
    out.println("</table ></body ></html >");
```

(2) 访问 HtmlServlet,Servelt 输出加法表格的效果如图 3-24 所示。

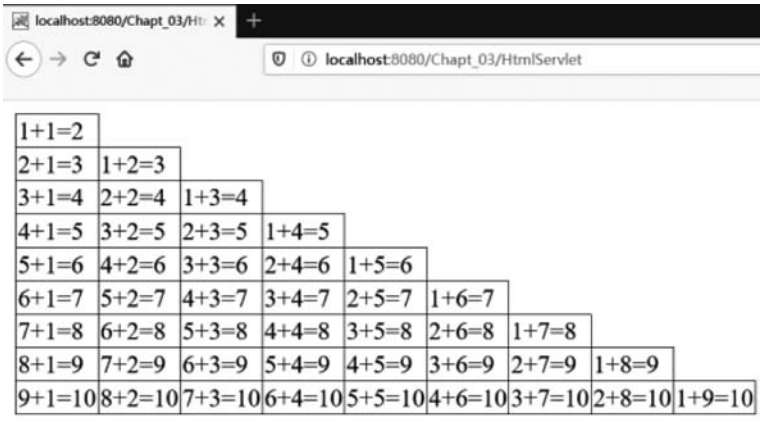

图 3-24　Servlet 输出加法表格的效果

从上面的例子可以看出,Servlet 通过输出流的方式可以逐句生成 HTML 标签以及 CSS 样式,从而生成动态页面。这种方式过于低效,相对于 JSP 页面,而 Servlet 的优势在于业务逻辑的处理。在后续章节学习完 JSP 后,可以结合 Servlet 和 JSP 技术共同开发。