# 1. Servlet进阶

## 1.1 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。

## 1.2 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();  }  } } |

即Servlet是单实例多线程的，一个Servlet类同时只能有一个实例在运行，如果有多个请求进入该Servlet，则都由该Servlet的service()方法处理。由于Servlet是多线程的，因此Servlet有线程安全问题。如果一个Servlet中有共享资源等情形，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。

## 1.3 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（优先级最高的是0）。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。

## 1.4 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。

## 1.5 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));  }  } } |

## 1.6 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。域对象就是用于通信的，在Servlet技术中，有三个域对象，分别是servletContext、session和request，这三个域对象的用法相同，底层都是使用了map集合，然后暴露给开发者的API就是getAttribute()等方法了。这些域对象的区别在于共享数据的范围不同，servletContext域对象在整个项目中都有效，至于session和request，稍后会讲解。在以后要讲的JSP中，有四大域对象，分别是application（即servletContext）、session、request和page，他们的作用范围依次减小。

## 1.7 转发、request域对象、重定向、包含

ServletContext对象中提供了一个getRequestDispatcher(String uri)方法，该方法返回的是一个“请求转发器”对象，利用该对象可以实现请求和相应的转发和包含。

调用请求转发器的forward()方法可实现请求和相应的转发，例如：

|  |
| --- |
| @WebServlet(urlPatterns = **"/oneServlet"**) **public class** DemoServlet **extends** HttpServlet {  @Override  **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  getServletContext().getRequestDispatcher(**"/anothorServlet"**).forward(req, resp);  } } |

上述实现的转发能达到什么样的效果呢？就是访问“oneServlet”后，该Servlet会把本Servlet得到的请求（req）和响应（resp）转发到“anothorServlet”中，这个“anothorServlet”中同样能够得到这个req和resp对象，进行自己的处理。

我们知道，request也是一个域对象，request对象中维护了一个Map，这个Map就是request域。request域的作用范围就在转发的页面之间（其他范围是无法访问request域的），因此利用转发，可以实现request域数据的共享。下面就是共享request域的示例（ServletDemo1转发到ServletDemo2后，还能得到原来的response和request对象）：

（1）ServletDemo1设置request域对象（域对象的操作方法和其他域对象的操作方法是一致的）。

|  |
| --- |
| @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。我们发现，转发时，浏览器显示的地址栏时不变的，虽然实际上是经过了两个不同的Servlet。

以后学习了JSP技术后，一般会在Servlet处理后，将请求和响应转发给JSP页面（JSP也相当于Servlet），最后JSP页面拿到request域数据进行显示。

那么转发和重定向（resp.sendRedirect(String url)）有何区别呢？

（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），能实现共享数据；而重定向无此功能。

简单的说，重定向只是单纯地实现页面的跳转。

实际上request对象本身就提供了getRequestDispatcher()方法，而不需要借助servletContext对象。例如：

|  |
| --- |
| @Override **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  req.getRequestDispatcher(**"/anothorServlet"**).forward(req, resp); } |

注意，如果在Servlet中执行了转发操作，就不要再通过字节流或者字符流输出数据了，因为没有任何效果，还可能导致转发显示的Servlet或JSP乱码。但是在转发前设置响应头和请求头是没问题的。

我们看到转发就是将request和response交给另一个Servlet处理。读者可能会说，我在这个Servlet的doGet()或doPost()方法中，新建需要的Servlet并调用service(request, response)方法不就好了？但这样有很多弊端，因为Servlet生命周期是WEB容器维护的，Servlet都是单实例多线程的，这样操作的话，就需要自己维护Servlet对象了，要调用init、destroy()等各种方法，麻烦而且效率低。强烈拒绝这种做法。而使用转发的方式就解决了这个问题，所有的Servlet还是由Tomcat维护，不会有任何影响。

我们一开始还说RequestDispatcher有“包含”的功能，即调用它的include()方法。包含的意思就是将这两个Servlet的内容合并再输出。比如：

（1）第一个Servlet（include到下面的第二个Servlet）：

|  |
| --- |
| @WebServlet(urlPatterns = **"/oneServlet"**) **public class** DemoServlet **extends** HttpServlet {  @Override  **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  resp.getWriter().write(**"hello1"**);  req.getRequestDispatcher(**"/anothorServlet"**).include(req, resp);  } } |

（2）第二个Servlet：

|  |
| --- |
| @WebServlet(urlPatterns = **"/anothorServlet"**) **public class** Demo2Servlet **extends** HttpServlet {  @Override  **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  resp.getWriter().write(**"hello2"**);  } } |

这样访问oneServlet就会出现“hello1hello2”这样的文本，这就是“包含”，即包含了两个Servlet输出的内容。在实际开发中，经常使用JSP页面做包含的操作，因为许多页面有相同的内容，可以把这些内容抽取到一个JSP中，然后其他的JSP只要包含这个JSP即可，这样便于统一管理和维护，不必多处改动。这个以后学习JSP时再说。

## 1.8 Servlet的路径匹配和Tomcat的web.xml

### 1.8.1 Servlet的路径匹配

Servlet的访问路径是通过url-patterns配置的。有以下两种方式配置访问路径：

方式一：路径匹配。即通过“/路径”的形式，其中“/”表示针对项目的相对路径。访问时，直接使用“http://主机:端口/应用名/Servlet路径”即可。

方式二：后缀匹配。通常会使用通配符，例如将url-pattern配置成“\*.action”，则只要访问路径是以“.action”结尾的，一般都会进入该Servlet。例如“http://主机:端口/应用名/ABC.action”、“http://主机:端口/应用名/ABC/ABC.action”等。我们甚至可以将动态Servlet资源的路径用后缀匹配成“.html”以伪装成静态页面。

对于方式一的路径匹配，它可以配置多层，也能使用通配符。例如“/ABC/MyServlet”，访问路径就是“http://主机:端口/应用名/ABC/MyServlet”。再比如“/A/\*”，那么“http://主机:端口/应用名/A/任意路径”都可以匹配（可以多层级）。

除此之外，路径匹配还能直接写“/\*”或者“/”，他们的效果是相同的，能匹配上所有的请求路径。那么就有个问题，如果一个Servlet的访问路径配置成了“/\*”，是不是别的Servlet都无法访问了呢？

不是的，因为有如下规则：

访问路径匹配的范围越大，它的优先级越小。也就是说，你的匹配范围越大，则被访问的优先级就越小，只有别的高优先级的Servlet无法匹配成功，才会匹配你的这个Servlet。

另外还要注意，两种匹配方式不能混用，例如，不能使用“/\*.action”作为匹配路径，否则Tomcat启动时会抛出“IllegalArgumentException - Invalid <url-pattern>”的错误。

### 1.8.2 Tomcat中的web.xml

在Tomcat的conf目录下，也有一个web.xml文件。这个web.xml文件相当于是Tomcat中所有项目的web.xml文件的“父亲”，即项目中的web.xml会“继承”Tomcat中web.xml的配置，因此Tomcat中这个web.xml相当于是所有WEB项目的默认配置文件。只要我们修改了Tomcat中web.xml的配置，那么Tomcat中所有项目都会应用该配置。如果某个项目想修改默认配置，只要在本项目的web.xml中重写配置即可，因此他们类似于“子父类”关系。其实很多框架/软件都是通过这样的方式来让用户修改默认配置、实现自定义配置的。

我们来看看Tomcat的web.xml默认都配置了什么。

（1）配置了一个DefaultServlet，且该Servlet的url-pattern是“/”。相关的配置分别在103行和383行：

|  |
| --- |
| <**web-app**>  *<!-- 103行 -->* <**servlet**>  <**servlet-name**>default</**servlet-name**>  <**servlet-class**>org.apache.catalina.servlets.DefaultServlet</**servlet-class**>  *<!-- 相关配置 -->* <**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**>  *<!-- Tomcat启动时就创建默认Servlet -->* <**load-on-startup**>1</**load-on-startup**>  </**servlet**>  *<!-- 383行 -->* <**servlet-mapping**>  <**servlet-name**>default</**servlet-name**>  <**url-pattern**>/</**url-pattern**>  </**servlet-mapping**> </**web-app**> |

DefaultServlet就是Tomcat的默认Servlet。那么这个DefaultServlet为何要有范围最大的“/”进行路径匹配？因为默认的Servlet有如下作用：

1. 响应静态资源。Tomcat中静态资源也是需要Servlet来处理并响应的，这件事就是DefaultServlet来做的。例如我们将html、css、图片等静态资源放在了Tomcat的应用目录下，当我们输入正确的访问地址后，该DefaultServlet就会匹配到路径（因为开发者写的Servlet，其访问路径正常都是匹配不了这些资源的），而后读取资源并响应给用户。

2. 显示404等错误页面。以报404错误页面来说，由于DefaultServlet范围最大，当用户访问的资源，其他的Servlet都无法匹配时，就交由DefaultServlet处理了。如果DefaultServlet根据用户的请求路径并不能找到对应的静态资源，就会返回404未找到错误页面。

（2）配置了处理JSP和JSPX页面的Servlet。相关配置在251行和388行：

|  |
| --- |
| <**web-app**>  *<!-- 251行 -->* <**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**>  *<!-- 388行 -->* <**servlet-mapping**>  <**servlet-name**>jsp</**servlet-name**>  <**url-pattern**>\*.jsp</**url-pattern**>  <**url-pattern**>\*.jspx</**url-pattern**>  </**servlet-mapping**> </**web-app**> |

关于JSP，很快就要学了，到时可以回头过来看看。其实这里就是配置JSP页面对应的Servlet，该Servlet中会处理并响应JSP页面内容。

（3）web.xml中还配置了默认的session失效时间，单位是分钟，即session默认有效时间是30分钟。该配置在581行：

|  |
| --- |
| <**session-config**>  <**session-timeout**>30</**session-timeout**> </**session-config**> |

不久我们也会学session，其中会再次提到。

（4）该web.xml中还列出了互联网中所有的MIME类型。MIME类型用来定义互联网中各种类型的资源。从594行起，列出的MIME类型占据了大量篇幅，我们截取其中一个MIME类型：

|  |
| --- |
| <**mime-mapping**>  <**extension**>mp3</**extension**>  <**mime-type**>audio/mpeg</**mime-type**> </**mime-mapping**> |

其中“extension”表示资源文件的扩展名，“mime-type”表示该类型资源对应的MIME类型。以后我们需要查找MIME类型时，可以把Tomcat的web.xml当做字典。

（5）最后，从4679行开始，定义了默认的欢迎页面：

|  |
| --- |
| <**welcome-file-list**>  <**welcome-file**>index.html</**welcome-file**>  <**welcome-file**>index.htm</**welcome-file**>  <**welcome-file**>index.jsp</**welcome-file**> </**welcome-file-list**> |

这也解释了为什么我们虽然没有在自己的项目中配置欢迎页，当访问应用根目录时，也能自动跳转到index.html页面的原因。

# 2 获取应用的资源文件

## 2.1 应用中资源的存放位置

无论是开发Java SE应用还是开发Java WEB应用，经常会在程序启动时加载必要的资源文件，例如图片、数据库连接配置文件等。

在Java SE工程中，一般将资源文件放在src目录下（或者其子文件夹）。因为Eclipse中，Java SE工程默认输出路径就是项目的bin文件夹（可在资源管理器中查看），即IDE会把src中的源码带包目录生成字节码文件，然后将字节码资源放入bin目录中，同时把src目录下的其他资源（即不是Java源文件的）原封不动地按照目录结构拷贝到bin目录下。

而在Java Web中，同样可以先将资源放在Java Web工程的src目录下，与Java SE不同的是，因为Java Web应用是在Web容器中运行的，此时IDE会把src下的资源放在Tomcat应用目录的“WEB-INF/classes”目录下，例如“Tomcat/ServletDemo/WEB-INF/classes”。因为WEB-INF/classes目录也是用来存放字节码文件的，因此src下的资源总是被拷贝到和字节码资源同一根目录。

当然，也可以将资源放在其他位置，比如放在包目录下，即和某些Java类放在一个目录下。在Java Web中，还可以将资源文件直接放在应用根目录下，例如“Tomcat/ServletDemo”（在Eclipse工程中就是放在WebContent目录下，运行时自动复制到Tomcat的应用根目录下）；或直接放在WEB-INF目录下。但有一条，这些资源文件都是放在和该应用有相对关系的路径下，而不是随便放在本机的D盘、F盘等，因为这样有利于项目的部署，所有的资源都包含在了应用中，且是用相对路径读取的，不会出错。

通过下面的学习我们就能很好地读取上述的这些文件。

## 2.2 利用servletContext加载Java Web资源文件

下面先以一个再Java Web中加载资源的例子来切入。

例：该Java Web项目在一开始运行的时候，就需要加载应用目录下的“WEB-INF/student.xml”文件，如何实现？

实现该例子有两个要点，一个是加载资源的代码要在应用刚启动时就执行，如何实现呢？根据之前所学，我们可利用Servlet的自动加载功能实现：新建一个Servlet类，重写init无参方法，设置好该Servlet的路径和loadOnStartup即可，这样init方法中的代码就会在应用启动时执行。示例代码：

|  |
| --- |
| **package** demo;  **import** javax.servlet.annotation.WebServlet; **import** javax.servlet.http.HttpServlet;  *// 一定要写上urlPatterns属性，否则不生效* @WebServlet(urlPatterns = **"/initServlet"**, loadOnStartup = 1) **public class** MyInitServlet **extends** HttpServlet {  @Override  **public void** init() {  System.***out***.println(**"初始化资源代码..."**);   } } |

注意：虽然配置该Servlet的路径是没有意义的（因为目前看，该Servlet只管加载资源，没有处理请求），但是urlPatterns也是要配置的，否则还是不会自动执行。以后学习了过滤器Filter和监听器Listener后，可使用Filter或者Listener来让应用启动时执行代码。

实现该例子的第二个要点就是用代码的方式找到该资源并进行加载了。下面我们一步一步来探究。

在Eclipse工程中，该资源文件的目录就是“D:\EclipseWorkspace\ServletDemo\WebContent\WEB-INF\student.xml”下，考虑用以下方式拿到文件：

|  |
| --- |
| File file = **new** File(**"D:/EclipseWorkspace/ServletDemo/WebContent/WEB-INF/student.xml"**); |

但这样其实不对，因为之前说过项目是在Tomcat中运行的，Eclipse工程中的这些资源文件实际上在部署到Tomcat时，就被复制到Tomcat应用中对应的目录了，因此我们应该用student.xml文件在Tomcat中的路径（假设Tomcat在E盘）：

|  |
| --- |
| File file = **new** File(**"E:/Tomcat/ServletDemo/WEB-INF/student.xml"**); |

但这样还是不科学的，路径是在程序中写死的，这样在其他机器部署项目就会遇到麻烦，这个之前也说过，因此我们应该使用相对路径寻找资源文件（一般是相对于项目有关的路径，例如相对于项目的根目录、相对于某个字节码目录等。因为不论把项目放在何处，资源和项目之前的位置关系是不变的！），这是寻找资源的真正解决之道！

在Java项目中，不论是Java SE还是Java Web，默认的相对路径就是JVM的执行目录。例如我们在项目中写如下代码：

|  |
| --- |
| File f1 = **new** File(**"student.xml"**); File f2 = **new** File(**"demo/student.xml"**); |

他们都表示在JVM执行目录下寻找资源，即会在JVM执行目录下寻找“student.xml”文件和JVM执行目录中demo文件夹下的student.xml文件。那么JVM的执行目录是哪个目录呢？JVM的执行目录就是运行“java”命令的地方。你可以在任何地方运行java命令来执行字节码文件，只要你设置了正确的classpath路径即可。

在Eclipse的Java SE项目中，JVM的执行位置就是项目的根目录，因此会在项目的根目录寻找资源文件。但是在Eclipse中，Java SE的字节码文件放在bin目录下，因此如果我们自己在bin目录下运行java命令的话，JVM就会在bin目录下寻找资源。而对于Java Web应用，JVM是在Tomcat的bin目录执行的，因此当执行上述的代码时，JVM会在Tomcat的bin目录寻找资源。

上一段说的都是经过验证的，读者可自行验证（验证Java Web时，最好自行将项目部署到Tomcat中验证，因为IDE可能会自行设置JVM启动路径，总之在应用真正部署运行时，JVM的启动路径一般就是Tomcat的bin目录）。

既然这样，为什么我们之前在Java SE中，一般直接写“new File("src/student.xml")”也能加载成功呢？这是因为我们开发和运行时都在Eclipse项目中进行的，运行时使用的其实就是src目录下的student.xml资源文件。实际上这样做也是不对的，因为项目的运行实际是依靠字节码文件的，真正运行Java SE时只要bin目录下的资源即可。Eclipse也会把src下的资源复制到bin目录中，因此我们以后使用资源应该相对于classpath路径或者字节码路径查找资源，而不是依赖Eclipse环境的bin目录和src目录等。至于到底如何做，在下一章中我们详细讲解。

我们回过头来再看例子中的要求。例子要求我们找到应用目录下的“WEB-INF/student.xml”文件，而经过上述的分析我们的得出的结论是：

要通过相对路径的方法寻找，但通过JVM默认的相对路径显然也是无法拿到我们需要的资源的。因此我们进一步研究其他使用相对路径的方法。

我们想要的一个加载资源的方式是：该方式查找资源的相对路径是Tomcat的该应用根目录，即该方式会在“Tomcat的应用目录”下查找资源，例如会在“D:/Tomcat/webapps/MyDemo”下查找资源。

实际上ServletContext对象提供了用于获取资源的方法getResourceAsStream和getRealPath，这些方法就是基于应用的根目录来查找资源的：

（1）getResourceAsStream(资源路径)：以流的方式获取资源，返回InputStream。

（2）getRealPath(资源路径)：获得资源在本机中的绝对路径，返回String字符串。

上述参数中，资源路径前加不加“/”都可，都代表相对于应用根目录。例如：

|  |
| --- |
| @WebServlet(urlPatterns = **"/initData"**, loadOnStartup = 1) **public class** MyServlet **extends** HttpServlet {  @Override  **public void** init() **throws** ServletException {  *// 以流的方式获取应用目录下的student.xml资源。student.xml前也可加上"/"，效果相同* InputStream is = getServletContext().getResourceAsStream(**"student.xml"**);  *// 获得应用目录下，demo文件夹中的student.xml资源。demo前的"/"也可省略，效果相同。* String path = getServletContext().getRealPath(**"/demo/student.xml"**);  System.***out***.println(path);  } } |

例如应用目录是MyWeb，那么“is”获取的就是“E:/Tomcat/webapps/MyWeb/student.xml”资源的输入流，而“path”就是“E:/Tomcat/webapps/MyWeb/demo/student.xml”。

这样示例中的问题就解决了，只要使用：

|  |
| --- |
| InputStream is = getServletContext().getResourceAsStream(**"/WEB-INF/student.xml"**); |

即可。

我们现在就可总结出ServletContext一共有三个作用：

（1）获得web.xml中的全局配置信息；

（2）ServletContext是Servlet中的三大域对象之一；

（3）ServletContext可用于获得项目中的资源。

现在再来扩展一下获取资源的方式。上面我们直接获取的是应用根目录的资源，这个在IDE（Eclipse）中表现为WebContent目录下的资源。那么现在我在Java Web项目的src目录下存放了资源文件，该如何获得呢？前面我们说过，在Java Web项目中，src下的资源文件会被复制到应用的WEB-INF/classes目录，即字节码所在目录，因此如果src下有一个student.xml文件，那么使用ServletContext的查找方式为：

|  |
| --- |
| InputStream is = getServletContext().getResourceAsStream(**"/WEB-INF/classes/student.xml"**); |

那么如果student.xml在包中呢（即和某字节码文件同目录），例如在com.servlet.demo中，那么查找的方式就为：

|  |
| --- |
| InputStream is = getServletContext().getResourceAsStream(**"/WEB-INF/classes/com/servlet/demo/student.xml"**); |

看起来，上述拿到WEB-INF/classes目录中的资源文件比较麻烦，但是可行的。实际上，我们可以利用JDK提供的方法，更容易地拿到这些文件，这些方法同样适用于Java SE项目中，可以解决上述提到的通用的问题。下面一章就专门讲解通用的获取资源的方法。

## 2.3 通用的加载资源的方法

现在系统地介绍在Java中如何获得特定位置的资源。这个方法都是基于字节码、classpath等相对路径获取资源的，而不是基于绝对路径。因此这样获取资源就能不依赖于具体的机器文件系统环境，而只依赖于自身项目。这里介绍的方法在Java SE和Java Web中都是通用的。

（一）利用字节码对象提供的方法获取资源

字节码对象上提供了用于获取资源文件的方法，例如：

（1）getResourceAsStream(相对路径)：以流的方式获取资源，返回类型是InputStream。如果找不到资源就返回null。

（2）getResource(相对路径)：返回资源的URL，返回类型是URL。如果找不到资源就返回null。

上述方法基于的相对路径是相同的：如果相对路径以“/”开头，就代表从字节码目录开始寻找资源资源；如果相对路径不以“/”开头，表示从当前类的字节码文件所在路径查找。

例如：

|  |
| --- |
| **package** com.resource.demo;  **import** java.io.IOException; **import** java.io.InputStream;  **import** r.R; *// R类的包是r* **public class** Main {  **public static void** main(String[] args) **throws** IOException {  *// 说明：在Java SE Eclipse中，字节码目录是bin，不过使用该方法和bin目录无关。即使自行执行该字节码文件，也能找到该文件  // 查找字节码目录下的student.xml文件。在Eclipse中是bin目录下，在Web中是EB-INF/classes目录下。* InputStream is1 = Main.**class**.getResourceAsStream(**"/student.xml"**);  System.***out***.println(is1);  *// 查找Main类字节码文件目录下的student.xml文件，即查找com.resource.demo目录下的student.xml资源。* InputStream is2 = Main.**class**.getResourceAsStream(**"student.xml"**);  System.***out***.println(is2);  *// 注意这个是查找R类字节码文件目录下的student.xml文件。  // 即查找r目录下的student.xml资源（因为R类的包就是r）。因为这个是使用R类字节码对象的方法，是跟R类有关的。* InputStream is3 = R.**class**.getResourceAsStream(**"student.xml"**);  System.***out***.println(is3);  } } |

其中需要注意的就是使用不同的字节码对象，查找的路径也是不一样的，是按照字节码对象的包目录来的（对于不加“/”的情况来说）。

再例如：

|  |
| --- |
| **package** com.resource.demo;  **import** java.io.IOException; **import** java.net.URL;  **import** r.R; *// R类的包是r* **public class** Main {  **public static void** main(String[] args) **throws** IOException {  *// 查找字节码目录下的student.xml文件。* URL u1 = Main.**class**.getResource(**"/student.xml"**);  System.***out***.println(u1);  *// 查找Main类字节码文件目录下的student.xml文件。* URL u2 = Main.**class**.getResource(**"student.xml"**);  System.***out***.println(u2);  *// 查找R类字节码文件目录下的student.xml文件。* URL u3 = R.**class**.getResource(**"student.xml"**);  System.***out***.println(u3);  } } |

直接输出url对象，得到的URL路径信息类似于：

|  |
| --- |
| file:/C:/eclipse-workspace/ResourceDemo/bin/r/student.xml |

如果想要获得真正的本机绝对路径信息，将上述的字符串按需截取即可。然后拿到路径后就可按照自己的要求加载了。

（二）使用类的加载器(ClassLoader)提供的方法加载资源

使用字节码对象Class的getClassLoader()方法可以获取类的加载器对象，例如：

|  |
| --- |
| ClassLoader cl = Main.**class**.getClassLoader(); *// 获得Main类字节码的类加载器* |

那么类加载器的作用是什么呢？我们知道字节码文件是要放到JVM中运行的，因此JVM就要加载字节码文件，就要使用IO流去读取字节码文件。类加载器主要就是用来读取字节码文件这些资源的。类加载器中提供了读取资源的方法，而我们也可利用他们来读取我们需要的资源。

一般使用ClassLoader中以下方法：

（1）getResourceAsStream(相对路径)：以流的方式获取资源，返回类型是InputStream。如果找不到资源就返回null。

（2）getResource(相对路径)：返回资源的URL，返回类型是URL。如果找不到资源就返回null。

参数中，“相对路径”不能以“/”开头，上述方法会从字节码根目录查找资源，这点和使用字节码文件对象的方法效果相反。

最后再注意以下问题：

（1）Class字节码对象和ClassLoader对象中提供的方法，原则上是用来加载字节码文件的，它们一旦第一次加载完资源，就会将资源存放在内存中，直到虚拟机关闭。因此在程序运行期间，如果该资源的内容发生了变化，那么即使你多次调用getResourceAsStream()方法，你读到的数据还是初次读取时读到的数据，因为JVM只会再从内存中读取数据。这样，在有些场景是无法及时获得新数据的。

解决办法就是使用getResource()方法获得资源路径，以后用路径读取资源（例如用路径创建File对象再读取），因为资源的位置一般不会改变。

（2）因为Class和ClassLoader原则上是用来加载字节码文件的，所以他们的缓存设计得很小，因此不要使用Class和ClassLoader的getResourceAsStream()方法读取较大的文件。

基于以上两点，强烈建议多考虑使用getResource()方法，而不是getResourceAsStream()方法。

# 3. Servlet的设计分析：从Servlet到HttpServlet

## 3.1 Servlet接口

Servlet就是能生成动态资源的Java类（有Servlet容器如Tomcat处理）。在Java EE官方文档（API）中，我们发现Servlet是一个接口，其描述如下：

|  |
| --- |
| public interface Servlet  Defines methods that all servlets must implement.  A servlet is a small Java program that runs within a Web server. Servlets receive and respond to requests from Web clients, usually across HTTP, the HyperText Transfer Protocol.  To implement this interface, you can write a generic servlet that extends javax.servlet.GenericServlet or an HTTP servlet that extends javax.servlet.http.HttpServlet.  This interface defines methods to initialize a servlet, to service requests, and to remove a servlet from the server. These are known as life-cycle methods and are called in the following sequence:  1.The servlet is constructed, then initialized with the init method.  2.Any calls from clients to the service method are handled.  3.The servlet is taken out of service, then destroyed with the destroy method, then garbage collected and finalized.  In addition to the life-cycle methods, this interface provides the getServletConfig method, which the servlet can use to get any startup information, and the getServletInfo method, which allows the servlet to return basic information about itself, such as author, version, and copyright. |

从上述的文档中我们可以知道：

（1）有三种方式创建Servlet类：

方式一：实现Servlet接口；

方式二：继承GenericServlet类；

方式三：继承HttpServlet类。

我们之前直接使用的是方式三，通过下面的讲解，实际上所有的Servlet类都实现了Servlet接口，只是一般使用HttpServlet更方便。

（2）文档中说明了Servlet的生命周期及对应执行的方法是init()、service()和destroy()。这和我们之前通过HttpServlet讲的是一致的，其实这些生命周期就是Servlet接口中定义的。我们可直接新建一个类实现Servlet接口，然后实现其中的方法：

|  |
| --- |
| **package** demo.servlet; **import** javax.servlet.\*; **import** java.io.IOException; **public class** SimpleServlet **implements** Servlet {  **public void** init(ServletConfig config) **throws** ServletException {  System.***out***.println(**"初始化"**);  }    **public void** service(ServletRequest req, ServletResponse res) **throws** ServletException, IOException {  System.***out***.println(**"提供服务"**);  }    **public void** destroy() {  System.***out***.println(**"销毁Servlet"**);  }   **public** String getServletInfo() {  **return null**;  }   **public** ServletConfig getServletConfig() {  **return null**;  } } |

在web.xml中配置了Servlet路径映射后放到Tomcat中运行，是正常执行的。下面我们分析GenericServlet和HttpServlet，会知道他们都直接和间接实现了Servlet接口。

（3）Servlet接口中除了声明了上述的三个声明周期方法，还提供了getServletInfo()和getServletConfig()这两个方法。

getServletInfo()返回Servlet版本、作者信息等，但一般没什么作用，直接返回空字符串即可。

getServletConfig()返回ServletConfig对象。哪里有这个对象？其实这是在init()方法中传递过来的，我们只要在该Servlet中维护这个servletConfig，并在getServletConfig()中返回即可，例如：

|  |
| --- |
| **public void** init(ServletConfig config) **throws** ServletException {  **this**.**config** = config; }  **public** ServletConfig getServletConfig() {  **return config**; } |

Java EE官当文档中说，ServletConfig是：

|  |
| --- |
| A servlet configuration object used by a servlet container to pass information to a servlet during initialization. |

也就是初始化Servlet时就将ServletConfig传进来的。ServletConfig就是有关此Servlet的配置信息，可以在web.xml的该Servlet配置中定义配置信息，这些都在之前讲过了。当然，ServletConfig中还提供了getServletContext()方法，即返回ServletContext，这是获得全局的应用配置信息。

## 3.2 Servlet使用优化：GenericServlet和HttpServlet

GenericServlet和HttpServlet是Java EE设计的Servlet类，目的是简化Servlet的使用。因此API文档中说也可以使用他们创建Servlet类。下面我们来分析他们这些类的设计，以便对Servlet有更深的了解。

（1）GenericServlet

GenericServlet目标是简化Servlet使用，其中实现好了一些方法、维护了ServletConfig变量、增强了一些方法。下面我们就编写自己的MyGenericServlet，模拟GenericServlet实现。

第一步：维护好servletConfig对象、实现一些方法：

|  |
| --- |
| **package** demo.servlet;  **import** javax.servlet.\*; **import** java.io.IOException;  *// 该类需要实现Servlet接口* **public abstract class** MyGenericServlet **implements** Servlet {  *// 维护servletConfig对象* **private** ServletConfig **config** = **null**;  **public void** init(ServletConfig config) **throws** ServletException {  **this**.**config** = config; *// 给config赋值* }   **public** ServletConfig getServletConfig() {  **return config**; *// 将config返回即可* }   *// service方法用于处理请求和响应，每个Servlet类中处理逻辑是不同的。因此这里不实现该方法，使用abstract关键字，让用户重写。这样，该MyGenericServlet也要写成abstract。* **public abstract void** service(ServletRequest req, ServletResponse res) **throws** ServletException, IOException;   *// 直接返回一个空字符串。用户如果想返回一些信息，直接覆盖此方法。* **public** String getServletInfo() {  **return ""**;  }   *// Servlet销毁时，默认什么也不做。如果需要做什么，用户可覆盖此方法* **public void** destroy() {   } } |

这样我们写的Servlet类就直接继承MyGenericServlet类，通常只要重写service方法即可。例如在service中得到ServletContext对象：

|  |
| --- |
| **package** demo.servlet;  **import** javax.servlet.\*; **import** java.io.IOException;  **public class** MyServlet **extends** MyGenericServlet {  **public void** service(ServletRequest req, ServletResponse res) **throws** ServletException, IOException {  System.***out***.println(**"进入了service"**);  ServletContext sc = getServletConfig().getServletContext();  } } |

ServletConfig接口中的方法也是很常用的，能不能让我直接调用getServletContext()方法呢？想实现这样的小功能也很简单，只要在MyGenericServlet中添加这些方法即可，例如：

|  |
| --- |
| public void getServletContext(){  return getServletConfig().getServletContext();  } |

实际上有个更好的方法，直接让MyGenericServlet再实现ServletConfig接口，实现其中的方法即可。

另外我们还要解决一个初始化Servlet的小问题。如果我们使用MyGenericServlet类，如何初始化Servlet呢？如果我们使用实现Servlet接口的方式，只要将初始化代码写在要实现的init()方法中即可，但现在继承MyGenericServlet，如果重写init()方法，那么MyGenericServlet中的init()就会被覆盖，其中维护servletConfig的代码就失效了，需要用户自己维护。这样就不太友好。如何解决这个问题？我们可以这样做：在MyGenericServlet的init(ServletConfig)方法中，再调用一个无参数的init()方法，该无参init()方法在MyGenericServlet中是空方法，专门用于给子类重写，以便初始化资源。这样，子类如果要初始化资源，直接重写init()无参方法即可，而不会影响正常工作。

这样，MyGenericServlet代码如下：

|  |
| --- |
| **package** demo.servlet;  **import** javax.servlet.\*; **import** java.io.IOException; **import** java.util.Enumeration;  *// 再实现ServletConfig接口* **public abstract class** MyGenericServlet **implements** Servlet, ServletConfig {  **private** ServletConfig **config** = **null**;  **public void** init(ServletConfig config) **throws** ServletException {  **this**.**config** = config;  init(); *// 调用init()方法* }    *// 空的init()方法* **public void** init() {}   **public** ServletConfig getServletConfig() {  **return config**;  }    **public abstract void** service(ServletRequest req, ServletResponse res) **throws** ServletException, IOException;    **public** String getServletInfo() {  **return ""**;  }    **public void** destroy() {   }   **public** String getServletName() {  **return** getServletConfig().getServletName();  }   **public** ServletContext getServletContext() {  **return** getServletConfig().getServletContext();  }   **public** String getInitParameter(String name) {  **return** getServletConfig().getInitParameter(name);  }   **public** Enumeration<String> getInitParameterNames() {  **return** getServletConfig().getInitParameterNames();  } } |

我们查看Tomcat提供的GenericServlet源码，其中基本和我们写的思路是一致的，只是添加了序列化、日志等功能。（强烈建议查看此类源码）

使用GenericServlet的示例代码：

|  |
| --- |
| **package** demo.servlet;  **import** javax.servlet.\*; **import** java.io.IOException;  **public class** MyServlet **extends** GenericServlet {  @Override  **public void** init() **throws** ServletException {  System.***out***.println(**"初始化资源，这不是servlet的生命周期方法init(ServletConfig sc)"**);  }   @Override  **public void** service(ServletRequest req, ServletResponse res) **throws** ServletException, IOException {  System.***out***.println(**"service方法"**);  } } |

现在就能理解为何我们之前在HttpServlet中也重写无参的init()来初始化资源。

（2）HttpServlet

HttpServlet是基于GenericServlet、针对Web中常用的HTTP协议做的优化。

优化1：Web中基本都是HTTP协议，因此，将service方法中的ServletRequest和ServletResponse强转成HttpServletRequest和HttpServletResponse。强转是没问题的，因为如果是在WEB中，使用的就是HTTP协议，本身也是HttpServletRequest和HttpServletRepsonse对象。

优化2：按照请求方式的不同，执行不同的方法，例如doGet、doPost等，这样用户可根据需要使用不同的方法。

因此，具体的优化代码示例如下：

|  |
| --- |
| **package** demo.servlet;  **import** javax.servlet.\*; **import** javax.servlet.http.\*; **import** java.io.IOException;  **public class** MyHttpServlet **extends** GenericServlet {  **public void** service(ServletRequest req, ServletResponse res) **throws** ServletException, IOException {  *// 优化1：把req和res直接强转成http的* HttpServletRequest request = (HttpServletRequest) req;  HttpServletResponse response = (HttpServletResponse) res;  *// 优化2：根据请求方式的不同，进入不同的处理方法。由具体类型的service方法负责处理* service(request, response);  }   **public void** service(HttpServletRequest req, HttpServletResponse res) {  *// 1. 获得请求的方式* String method = req.getMethod();  *// 2. 根据请求方式的不同，调用不同的请求方法* **if** (**"GET"**.equalsIgnoreCase(method)) {  doGet(req, res);  } **else if** (**"POST"**.equalsIgnoreCase(method)) {  doPost(req, res);  } **else** {  *// ...其他方式* }  }   *// 下面的方法空实现即可，让用户选择覆盖。如果使用abstract，那么用户就必须覆盖方法，这对用户不太友好。* **public void** doPost(HttpServletRequest req, HttpServletResponse res) {    }   **public void** doGet(HttpServletRequest req, HttpServletResponse res) {    } } |

我们查看真实的HttpServlet源码，发现是类似的。这就是GenericServlet和HttpServlet的设计思路，一般开发中直接继承HttpServlet即可。