# 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。

## 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

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"**));  } } |

# 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。