# 1. Web开发概念

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

学习Web开发，必须先掌握Java SE知识和基本的前端知识。

## 1.1 服务器和Web服务软件

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

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

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

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

（2）Jetty：可作为一个嵌入的服务器。

（3）JBoss：Redhat公司的产品。

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

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

## 1.2 网址中的一些概念

我们使用浏览器访问网页，如果要访问百度的首页，则访问的地址是“http://www.baidu.com/index.html”。服务器一般用80端口对外提供HTTP访问，因此完整的地址是：

|  |
| --- |
| http://www.baidu.com:80/index.html |

只不过浏览器地址栏中不显示默认的80端口，如果不是80端口，则会正常显示。当然，现在很多网站提供的是HTTPS协议，比HTTP更加安全，HTTPS默认使用的端口是443，因此现在的百度首页地址实际是（对于HTTPS协议，浏览器也不会显示默认的443端口）：

|  |
| --- |
| https://www.baidu.com:443/index.html |

我们现在对网址进行分析，其中：

（1）“http://”或者“https://”代表使用的是HTTP或者HTTPS协议；

（2）“www.baidu.com”是主机名，而“baidu.com”是域名。域名是通过DNS服务器找到其对应的IP的。一个域名下可以有多个主机，例如“music.baidu.com”、“zhidao.baidu.com”等；

（3）“:80”就指明了连接的端口，一般是80，但我们下面使用的Tomcat默认的端口是8080，因此需要指明。

（4）最后就是指明要请求的主机的资源了，就是要请求“index.html”这个资源。

# 2. Tomcat服务器

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

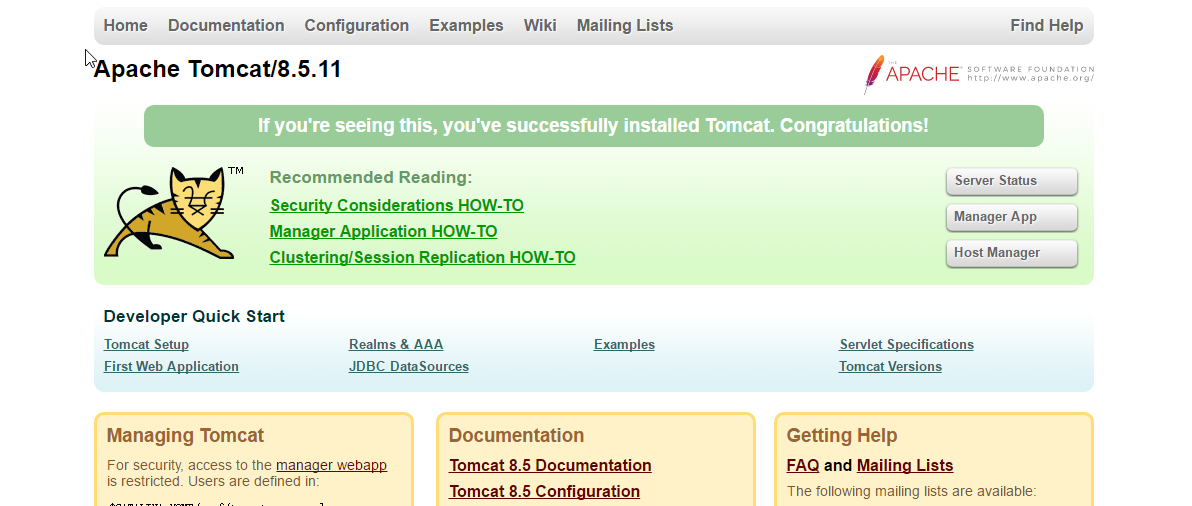
## 2.1 安装和运行

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

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

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

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



因为Tomcat默认监听的端口就是本机的8080端口。

常见的Tomcat不能启动的问题：

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

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

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

## 2.2 Tomcat的目录结构

bin：存放可执行命令。

conf：存放配置文件；其中server.xml是核心配置文件。

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

logs：存放Tomcat产生的日志文件。系统产生的异常信息都会保存在日志文件中，因此如果Tomcat不能启动或者项目出错，都能在日志文件中查看到详情。

temp：用于存放临时文件。

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

work：tomcat的运行目录。JSP运行时产生的临时文件就存放在这里。

## 2.3 体验Tomcat的作用

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

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

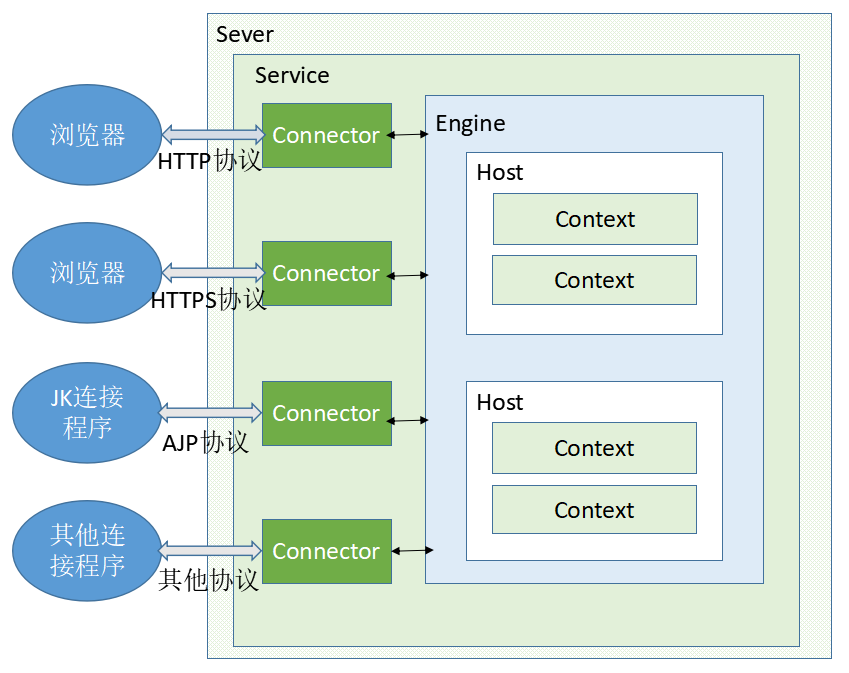
（2）开启Tomcat服务器，访问地址：[http://localhost:8080/exp/index.html 即可。如果你改过端口号，还要把8080改成你设置的端口。如果局域网其他人想访问此页面，只要将localhost换成你的机器的IP地址即可。](http://localhost:8080/exp/index.html即可。如果你改过端口号，还要把8080改成你设置的端口。如果局域网其他人想访问此页面，只要将localhost换成你的机器的IP地址即可。)

实际上浏览器访问的流程如下：

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

## 2.4 Tomcat的架构分析

Tomcat的架构图如下：



从图中可以看出：

整个Tomcat服务器通过一个Service向外界提供服务。在服务Service中维护了一个Engine和多个Connector（连接器）。

连接器Connector的作用是接收外界连接，并将连接信息交给Engine，Engine负责处理请求，最后Engine将资源返回给Connector，Connector再返回数据给客户端。Connector连接器就相当于客户端和服务器资源之间的“桥梁”。Tomcat中可配置多个连接器，每个连接器会占用端口以提供服务。例如Tomcat默认配置有一个基于HTTP的Connector和一个基于AJP协议的Connector，其中HTTP协议的Connector占用8080端口，因此我们开启服务器后可访问8080端口访问服务器资源。其中的AJP协议占用的是8009端口，JK连接程序就使用AJP协议与Tomcat通信。当然，这些默认的配置都是可以更改的。并且我们也能自行添加多个连接器（注意占用的端口不能冲突），例如（参照默认的配置）：

|  |
| --- |
| <**Connector port="8081" protocol="HTTP/1.1" connectionTimeout="20000" redirectPort="8443"** /> |

这样可以使用多个端口和Tomcat通信，既可以用浏览器访问8080端口，又可以访问8081端口。

Tomcat中，Engine只有一个，因此多个连接器与同一个Engine通信。Engine下可配置多个虚拟主机Host。例如我们配置一个名为“www.zhang.com”的主机：

|  |
| --- |
| <**Host name="www.zhang.com" appBase="zhangwebs" unpackWARs="true" autoDeploy="true"**>  *<!-- 这里的Valve标签用于配置该主机输出的日志文件 -->* <**Valve className="org.apache.catalina.valves.AccessLogValve" directory="logs" prefix="localhost\_access\_log" suffix=".txt" pattern="%h %l %u %t &quot;%r&quot; %s %b"** /> </**Host**> |

当Engine接收到Connector的连接信息时，就会分析该连接请求的是哪个主机，如果请求的是“www.zhang.com”主机，就会将该主机下对应的资源响应回去（该主机的资源根目录是配置的“zhangwebs”）。那为什么我们之前没有配置主机名，直接通过IP地址（或localhost）也能访问Tomcat资源呢？那是因为Tomcat默认有个名为localhost的主机：

|  |
| --- |
| <**Host name="localhost" appBase="webapps" unpackWARs="true" autoDeploy="true"**>  <**Valve className="org.apache.catalina.valves.AccessLogValve" directory="logs" prefix="localhost\_access\_log" suffix=".txt" pattern="%h %l %u %t &quot;%r&quot; %s %b"** /> </**Host**> |

并且Engine节点中使用的默认主机（defaultHost）就是这个名为“localhost”的主机：

|  |
| --- |
| <**Engine name="Catalina" defaultHost="localhost"**> |

因此默认访问的就是localhost这个主机。而这个主机的appBase（应用的根目录，是相对于Tomcat根目录的相对路径）为“webapps”，所以我们之前会把应用部署在webapps目录下。现在我们配置了“www.zhang.com”这个主机，该主机对应的应用根目录appBase是“zhangwebs”，因此我们也可在Tomcat中新建“zhangwebs”目录，然后在“zhangwebs”目录下存放应用，这样我们访问“http://www.zhang.com:8081/应用名”时就能访问到“zhangwebs”下的应用。当然，前提是客户端访问“zhang.com”时，能跳转到本机IP地址，这是需要DNS解析服务器支持的。

一般域名提供商会提供相应的服务，让你选择域名和对应的二级域名分别解析到哪个IP地址上。如果不指定的话，所有的二级域名默认指向的是一级域名的IP地址，因此可以自行指定多个二级域名分别指向哪些IP地址。当Tomcat服务器接收到这些请求时，就会根据二级域名的主机名请求各个虚拟主机的内容。

上述就是HOST的作用。我们能体会到主机名有两个用途：

（1）查找IP地址

（2）在服务器中查找要访问的主机。

这里还有一个问题，就是我们访问“localhost:8080”时，并没有指明应用名，为什么也能访问资源呢？那他访问的是哪个应用呢？当不指定应用名时，Tomcat默认会请求虚拟主机中ROOT目录中的应用，也就是说我们访问“localhost:8080”时，逻辑上访问的地址是“localhost:8080/ROOT”，但实际上并不能指明ROOT目录。如果我们在zhangwebs目录下新建ROOT目录，其中有index.html页面，那我们访问“http://www.zhang.com:8081”时访问的应用就是该index.html页面。当前前提还是客户端访问该域名时能指向该服务器的IP（下面我们会使用host文件进行模拟）。

上述的架构图其实就是Tomcat的server.xml配置文件的层次结构图。

## 2.5 Tomcat的访问流程

从客户端开始。例如客户端访问“exp.zhang.com”，先找本机的host文件看有无“exp.zhang.com”对应的IP地址，若没有，就从DNS服务器解析域名得到IP。通过IP和端口就能访问到服务器上的Tomcat，Tomcat的Connector接收请求，发给Engine处理，Engine找到相应主机对应的资源并返回。

## 2.6 设置Tomcat虚拟目录的三种方式

虚拟目录就是虚拟主机中应用的目录。Tomcat中有以下三种方式部署应用（设置虚拟目录）：

第一种：将应用文件夹直接放到主机名对应的appBase应用根目录下，这是最普通的方式。例如我们将应用文件夹放入webapps目录下，访问时直接用“http://主机名:端口/应用文件夹名”即可。如果配置多个主机，将应用放入对应的应用根目录下即可，比如“zhangwebs”目录。

这种方式是将应用资源放入Tomcat目录中了，有时我们并不想将应用放入Tomcat目录中，比如Tomcat在D盘，我的应用在E盘，我想直接把E盘的应用部署到Tomcat中。Tomcat是支持这样部署的，需要在Tomcat中进行配置，一般使用以下两种方式进行配置：

第二种：在Tomcat的server.xml配置文件中，在对应主机Host节点下，添加Context的节点配置。在Context中需要配置两个属性，一个是path，即指明该应用在主机中的访问路径，另一个是docBase，即指明该应用的目录。例如在名为“localhost”的主机中添加如下Context节点：

|  |
| --- |
| <**Context path="/myCtx" docBase="E:\MyExtProject"** /> |

那么启动Tomcat后直接访问“localhost:8080/myCtx”即可。

第三种：在Tomcat的conf/Catalina目录下，还有主机名目录，在对应的主机名目录（例如localhost）下，新建一个xml文件，文件名就是待访问的应用名，文件内容写Context标签，加上docBase应用位置这个属性即可。例如新建一个myCtx2.xml文件，内容为：

|  |
| --- |
| <**Context docBase="E:\MyExtProject"** /> |

那么启动Tomcat后直接访问“localhost:8080/myCtx2”即可。

## 2.7 山寨百度

最后我们基于上述的知识，做一个“山寨百度”的实战。需要实现的效果：本机浏览器输入“http://www.baidu.com”后，直接跳转到我们在Tomcat上山寨的“百度”页面。

下面是实现的步骤：

（1）首先，为了能使我们浏览器访问“www.baidu.com”时访问的是本机的IP，应该修改本机的hosts文件，因为DNS服务器我们是修改不了的。Windows中，hosts文件所在目录是：“C:\Windows\System32\drivers\etc”。由于hosts文件是系统重要文件，修改时需要管理员权限，因此我们用管理员权限打开notepad++，然后在notepad++中编辑此文件，添加一行内容为：

|  |
| --- |
| 127.0.0.1 www.baidu.com |

让请求“www.baidu.com”时指向本机IP。

（2）在Tomcat中，配置Connector的HTTP端口为80，并配置一个名为“www.baidu.com”的虚拟主机，应用根目录为baidu：

|  |
| --- |
| <**Host name="www.baidu.com" appBase="baidu" unpackWARs="true" autoDeploy="true"**>  <**Valve className="org.apache.catalina.valves.AccessLogValve" directory="logs" prefix="localhost\_access\_log" suffix=".txt" pattern="%h %l %u %t &quot;%r&quot; %s %b"** /> </**Host**> |

（3）在Tomcat目录中新建baidu目录，并在baidu目录下创建ROOT应用目录，在ROOT目录中新建index.html文件，内容如下：

|  |
| --- |
| <!DOCTYPE **html**> <**html lang="zh-CN"**> <**head**>  <**meta charset="UTF-8"**>  <**title**>百度一下，你就知道</**title**>  <**style type="text/css"**>  .**center** {  **text-align**: **center**;  }  </**style**> </**head**> <**body**> <**div class="center"**>  <**h1**>百度一下，你就知道</**h1**>  <**input type="text"**><**button**>百度一下</**button**> </**div**> </**body**> </**html**> |

这样，开启Tomcat，本机直接访问“www.baidu.com”就能访问到我们的山寨页面了。

总结一下：修改hosts是为了访问“www.baidu.com”时能跳转到本机IP，然后就是配置Tomcat的端口和主机，并在主机对应的“baidu”文件夹下创建ROOT文件夹，再将index.html放入ROOT中，因为ROOT是默认访问的应用。实际不配置“www.baidu.com”主机名也行，因为默认也会访问“localhost”主机名，这里这样做是为了加深对Tomcat配置的理解。

# 3. 动态资源开发技术：Servlet

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

## 3.1 第一个Servlet程序

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

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

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

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

现在我们正式写好第一个Servlet类：

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

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

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

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

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

温馨小提示：如果我们不依赖IDE的自动编译，自己如何手动基于jar包（例如本例中的servlet-api.jar）来编译该MyServlet.java文件呢？应该编译时指定classpath：

|  |
| --- |
| javac -cp .;C:\Tomcat\lib\servlet-api.jar -d . MyServlet.java |

或者在执行javac之前设置一下临时classpath：

|  |
| --- |
| set classpath=%classpath%;C:\Tomcat\lib\servlet-api.jar |

其实自己要清楚IDE只是帮助我们完成了这些步骤，而并不是通过什么神奇的手段。相关具体信息可以参见书籍《Java学习笔记》第二章。

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

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?> <!-- Schema约束 -->* <**web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://xmlns.jcp.org/xml/ns/javaee"  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"  id="WebApp\_ID" version="3.1"**>  *<!-- 欢迎页列表。可配置多个欢迎页面。如果存在欢迎页面，则访问该项目根目录时，会直接显示该欢迎页。 -->* <**welcome-file-list**>  <**welcome-file**>index.html</**welcome-file**>  <**welcome-file**>index.jsp</**welcome-file**>  </**welcome-file-list**>  *<!-- servlet标签用于配置程序中的一个servlet -->* <**servlet**>  *<!-- 定义该Servlet的名称 -->* <**servlet-name**>MyFirstServlet</**servlet-name**>  *<!-- 指定Servlet所在的类 -->* <**servlet-class**>com.zhang.test.MyServlet</**servlet-class**>  </**servlet**>  *<!-- servlet-mapping用于配置一个Servlet的访问路径 -->* <**servlet-mapping**>  *<!-- servlet的名称，和上面的一致 -->* <**servlet-name**>MyFirstServlet</**servlet-name**>  *<!-- 指定该servlet的访问路径，就是浏览器访问的路径。一定要写上斜杠 -->* <**url-pattern**>/hello</**url-pattern**>  </**servlet-mapping**> </**web-app**> |

web.xml中指明了可以用/hello路径来访问此Servlet。

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

我们一般说Tomcat是Servlet/WEB容器，从上面可见一斑。因为Tomcat能运行处理Servlet类和JSP页面（JSP实际也是Servlet）。Servlet的本质就是具备一定功能的java 类，利用Servlet能产生动态的HTML页面。

## 3.2 在Eclipse中开发Servlet

### 3.2.1 在Eclipse配置Tomcat

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

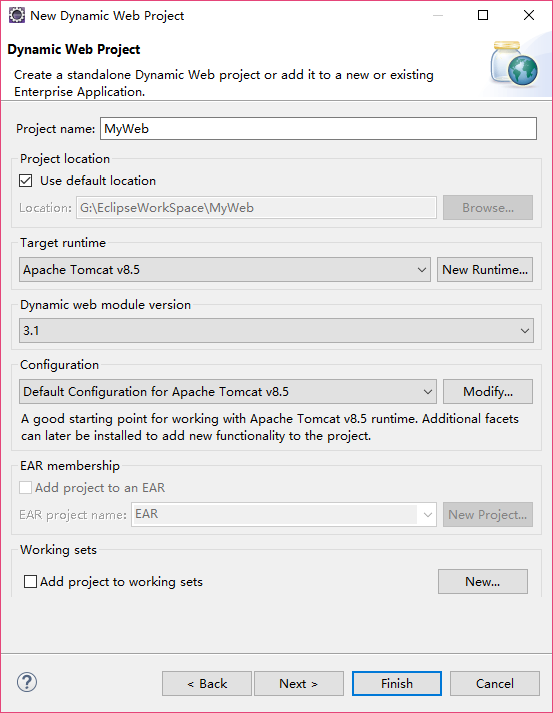
首先在Eclipse中配置好Tomcat。点击Window -> Proferences ，选择左边树状菜单中Server中的“Runtime Environments”一项。点击右侧界面的“Add...”按钮，选择Apache下的对应的Tomcat版本（这里选择v8.5），随后点击“Next”，浏览选择我们Tomcat的安装目录。最后Finish完成。

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

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

### 3.2.2 新建Java Web项目

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



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

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

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

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

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

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

## 3.3 使用Servlet注解

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

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

例子：

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

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

# 4. Servlet处理请求和响应

## 4.1 HTTP协议

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

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

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

说明：

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

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

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

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

（5）POST后面使用的是URI路径。需要注意URI和URL的区别：

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

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

同样，HTTP响应也包含了响应行、响应头和实体内容，比如：

|  |
| --- |
| HTTP/1.1 200 OK -- 响应行（下面开始是响应头）  Server: Apache-Coyote/1.1 -- 服务器名称  Content-Length: 24 -- 表示服务器发送给浏览器的数据长度  Date: Fri, 30 Jan 2015 01:54:57 GMT -- 服务器时间。注意服务器返回的时间时GMT时间，即格林尼治当地标准时间，该时间和我们东八区的时间相差8小时。因此不要以为是服务器的时间错误。  Content-Type: text/html; charset=UTF-8 -- 表示服务器响应数据的类型和编码  Set-Cookie: name=eric; path=/search -- 表示服务器设置给浏览器的Cookie信息  -- 一个空行，下面是实体内容，比如HTML数据  <html>This is HTML.</html> -- 实体内容 |

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

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

HTTP请求可以使用“Fiddle”软件进行查看。Fiddle能拦截本机所有的HTTP请求和响应，Fiddle界面的左侧是连接列表，右侧选择查看的详情。一般在“Inspectors”中查看详情，其中的“Raw”选项卡中显示了HTTP的原始数据。

## 4.2 Servlet获得请求头和请求参数

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

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

（1）用req获得请求行信息：

req.getMethod()：获得请求方式（POST 或GET）

req.getRequestURI()：获得请求的URI。

req.getRequestURL()，获得请求的URL。

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

（2）用req获得请求头信息：

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

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

（3）获取请求参数：

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

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

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

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

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

## 4.3 请求对象HttpServletRequest的应用

上述讲了request对象的常用方法，现在进行实际应用。

例1：获得请求头信息。根据user-agent请求头判断用户的浏览器。

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

例2：判断请求的来源页面。很多页面要求用户浏览了广告后才能看本页面内容。

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

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

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

下面写访问地址为view的Servlet：

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

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

例3：获取请求参数。

首先写以下请求数据的页面：

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

其中也可改成POST请求。这里在Servlet中写好两种方式的请求处理：

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

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

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

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

另外，在doPost()中，可以通过流来读取实体内容：

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

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

## 4.4 Servlet 获得响应头和设置响应数据

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

我们在Servlet中设置响应的数据同样通过resp对象。主要有以下方法：

（1）响应行

resp.setStatus(int code)：设置响应的状态码。

（2）响应头

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

（3）实体内容

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

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

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

## 4.5 响应对象HttpServletResponse的应用

例1：状态码和响应头应用。

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

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

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

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

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

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

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

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

例2：响应数据的编码问题。

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

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

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

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

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

实际上在HTML的meta标签中，其中的http-equiv属性就可用来模拟Content-Type响应头和Refresh响应头，例如：

|  |
| --- |
| <!DOCTYPE **html**> <**html lang="zh-CN"**> <**head**>  <**meta http-equiv="Content-Type" content="text/html;charset=UTF-8"** />  <**meta http-equiv="Refresh" content="3;http://www.baidu.com"**>  <**title**>Document</**title**> </**head**> <**body**> 内容 </**body**> </**html**> |

# 5. Servlet进阶

## 5.1 Servlet的缺省路径

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

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

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

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

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

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

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

即Tomcat是先查找动态资源，后查找静态资源。

## 5.2 Servlet的生命周期

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

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

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

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

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

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

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

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

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

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

## 5.3 Servlet线程安全问题

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

（1）HttpServlet类

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

（2）模拟容器的多线程Demo类

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

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。

## 5.4 自动加载Servlet

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

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

设置方法如下：

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

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

## 5.5 ServletConfig对象

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

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

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

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

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

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

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

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

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

## 5.6 ServletContext对象

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

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

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

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

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

## 5.7 域对象和ServletContext域对象

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

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

（1）向域对象中设置（即保存）数据的方法是：

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

（2）从域对象中获取数据的方法是：

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

（3）将数据从域对象中删除的方法是：

public void removeAttribute(String name)。

应用的例子：

（1）ServletDemo1设置域对象数据：

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

（2）ServletDemo2可获取域对象数据，因为在同一个应用中：

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

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

由此可体会到域对象的作用。

## 5.8 转发和重定向、request域对象

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

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

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

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

（2）转发只能转发到当前Web应用的链接资源，而重定向可跳转到任意地址。

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

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

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

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

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

（2）ServletDemo2可获取：

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

此时访问/demo1就会看到效果。

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

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

request对象还常用如下方法获得客户机相关信息：

（1）getRemoteAddr()：获得客户机的IP地址；

（2）getRemotePort()：获得客户机使用的端口号；

（3）getLocalAddr()：获得服务器的IP地址；

可自行再看看request提供的其他方法。

# 6 获取应用的资源文件

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

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

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

（1）对于Java SE应用，使用如下方法：

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

（2）对于Java EE应用，使用如下方法：

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

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

例子：

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

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

## 7.1 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。

## 7.2 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。