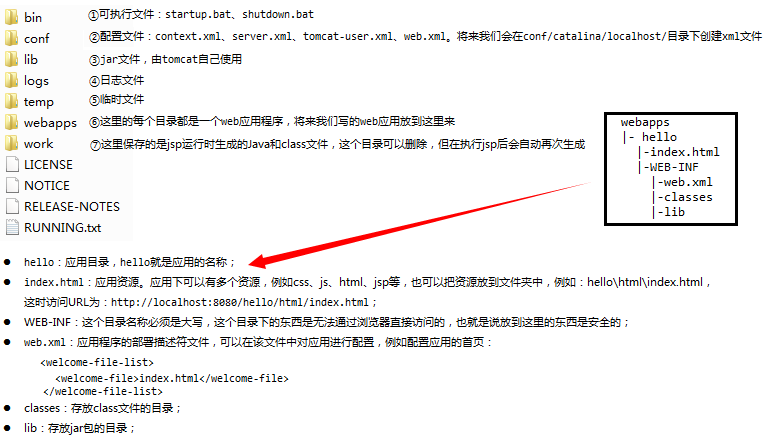
# 模拟面试卷一

### 服务器之Tomcat

1. **Tomcat的目录结构，并说说每个目录的大致功能？**



1. **Tomcat如何进行优化？**

①优化连接配置.这里以tomcat7的参数配置为例，需要修改conf/server.xml文件，修改连接数，关闭客户端dns查询。

②优化JDK，扩大tomcat使用的内存，默认为128M。

1. **Tomcat有那几种Connector运行模式？**

tomcat的运行模式有3种.修改他们的运行模式.3种模式的运行是否成功,可以看他的启动控制台,或者启动日志.或者登录他们的默认页面http://localhost:8080/查看其中的服务器状态。

1)bio：默认的模式,性能非常低下,没有经过任何优化处理和支持.

2)nio：利用java的异步io护理技术,no blocking IO技术.

想运行在该模式下，直接修改server.xml里的Connector节点,修改protocol为

<Connector port=”80″ protocol=”org.apache.coyote.http11.Http11NioProtocol”

connectionTimeout=”20000″

URIEncoding=”UTF-8″

useBodyEncodingForURI=”true”

enableLookups=”false”

redirectPort=”8443″ />

启动后,就可以生效。

3)apr：安装起来最困难,但是从操作系统级别来解决异步的IO问题,大幅度的提高性能.

必须要安装apr和native，直接启动就支持apr。下面的修改纯属多余，仅供大家扩充知识,但仍然需要安装apr和native

如nio修改模式,修改protocol为org.apache.coyote.http11.Http11AprProtocol

### Http协议

1. **什么是Http协议？说说你知道的请求头和响应头分别有哪些？**

超文本传输协议。这个协议详细规定了浏览器和万维网服务器之间互相通信的规则。通信规则规定了客户端发送给服务器的内容格式，也规定了服务器发送给客户端的内容格式。

**4.1请求头**：

①Referer：请求从哪个页面发出

\* 在页面上点击超链接或提交表单，都会有这个请求头。

\* 如果在浏览器的地址栏中直接发请求，那么就没有这个请求头。

作用：

\* 统计工作：可以统计访问来源是从哪里发出！例如统计多少访问来自baidu

\* 防盗链：下载本站资源，要查看是否Referer来自本站，如果不是那么就是盗链！

②Content-Type：application/x-www-form/urlencoded(请求体使用了URL编码)

\* 它说明的请求体的类型！！！

\* 这个头只有POST请求才有，而且默认值就是：application/x-www-form/urlencoded

③Content-Length:13

\* 只有POST请求才有这个头

\* 它说明的是请求体的字节数！

**响应头：**

* Last-Modified：最后的修改时间；
* 禁用浏览器缓存
* Expires: -1；
* Cache-Control: no-cache；
* Pragma: no-cache；
* 自动刷新====>>Refresh: 3;url=http://www.itcast.cn；

浏览器会在3秒之后请求http://www.itcast.cn：

1. **说说Http请求协议的格式以及大致内容**

POST请求是可以有请求体的，而GET请求不能有请求体。

①格式：

请求首行；

请求头信息；

空行；

请求体。

②GET请求

GET /hello/index.jsp HTTP/1.1

**Host**: localhost

**User-Agent**: Mozilla/5.0 (Windows NT 5.1; rv:5.0) Gecko/20100101 Firefox/5.0

**Accept**: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

**Accept-Language**: zh-cn,zh;q=0.5

**Accept-Encoding:** gzip, deflate

**Accept-Charset**: GB2312,utf-8;q=0.7,\*;q=0.7

**Connection**: keep-alive

**Cookie**: JSESSIONID=369766FDF6220F7803433C0B2DE36D98

-------此处是空行------

内容解说：

* GET /hello/index.jsp HTTP/1.1：**GET请求，请求服务器路径为/hello/index.jsp，协议为1.1；**
* Host:localhost：**请求的主机名为localhost；**
* User-Agent: Mozilla/5.0 (**Windows NT 5.1**; rv:5.0) Gecko/20100101 **Firefox/5.0**：**与浏览器和OS相关的信息。有些网站会显示用户的系统版本和浏览器版本信息，这都是通过获取User-Agent头信息而来的；**
* Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8：**告诉服务器，当前客户端可以接收的文档类型，其实这里包含了\*/\*，就表示什么都可以接收；**
* Accept-Language: zh-cn,zh;q=0.5：**当前客户端支持的语言，可以在浏览器的工具🡪选项中找到语言相关信息；**
* Accept-Encoding: gzip, deflate：**支持的压缩格式。数据在网络上传递时，可能服务器会把数据压缩后再发送；**
* Accept-Charset: GB2312,utf-8;q=0.7,\*;q=0.7：**客户端支持的编码；**
* Connection: keep-alive：**客户端支持的链接方式，保持一段时间链接，默认为3000ms；**
* Cookie: JSESSIONID=369766FDF6220F7803433C0B2DE36D98：**因为不是第一次访问这个地址，所以会在请求中把上一次服务器响应中发送过来的Cookie在请求中一并发送去过；这个Cookie的名字为JSESSIONID，然后在讲会话是讲究它！**

③POST请求

POST /hello/index.jsp HTTP/1.1

**Accept**: image/gif, image/jpeg, image/pjpeg, image/pjpeg, application/msword, application/vnd.ms-excel, application/vnd.ms-powerpoint, application/x-ms-application, application/x-ms-xbap, application/vnd.ms-xpsdocument, application/xaml+xml, \*/\*

**Referer**: http://localhost:8080/hello/index.jsp

**Accept-Language**: zh-cn,en-US;q=0.5

**User-Agent**: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0; InfoPath.2; .NET CLR 2.0.50727; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)

**Content-Type**: application/x-www-form-urlencoded

**Accept-Encoding**: gzip, deflate

**Host**: localhost:8080

**Content-Length**: 13

**Connection**: Keep-Alive

**Cache-Control**: no-cache

**Cookie**: JSESSIONID=E365D980343B9307023A1D271CC48E7D

keyword=hello

内容解说：

* Referer: http://localhost:8080/hello/index.jsp：**请求来自哪个页面，例如你在百度上点击链接到了这里，那么Referer:http://www.baidu.com；如果你是在浏览器的地址栏中直接输入的地址，那么就没有Referer这个请求头了；**
* Content-Type: application/x-www-form-urlencoded：**表单的数据类型，说明会使用url格式编码数据；url编码的数据都是以“%”为前缀，后面跟随两位的16进制，例如“传智”这两个字使用UTF-8的url编码用为“%E4%BC%A0%E6%99%BA”；**
* Content-Length:13：**请求体的长度，这里表示13个字节。**
* keyword=hello：**请求体内容！hello是在表单中输入的数据，keyword是表单字段的名字。**

1. **说说Http响应协议的格式以及大致内容**

①格式：

响应首行；

响应头信息；

空行；

响应体。

②响应内容：

HTTP/1.1 200 OK

Server: Apache-Coyote/1.1

Content-Type: text/html;charset=UTF-8

Content-Length: 724

Set-Cookie: JSESSIONID=C97E2B4C55553EAB46079A4F263435A4; Path=/hello

Date: Wed, 25 Sep 2012 04:15:03 GMT

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>

<head>

<base href="http://localhost:8080/hello/">

<title>My JSP 'index.jsp' starting page</title>

<meta http-equiv="pragma" content="no-cache">

<meta http-equiv="cache-control" content="no-cache">

<meta http-equiv="expires" content="0">

<meta http-equiv="keywords" content="keyword1,keyword2,keyword3">

<meta http-equiv="description" content="This is my page">

<!--

<link rel="stylesheet" type="text/css" href="styles.css">

-->

</head>

<body>

<form action="" method="post">

关键字：<input type="text" name="keyword"/>

<input type="submit" value="提交"/>

</form>

</body>

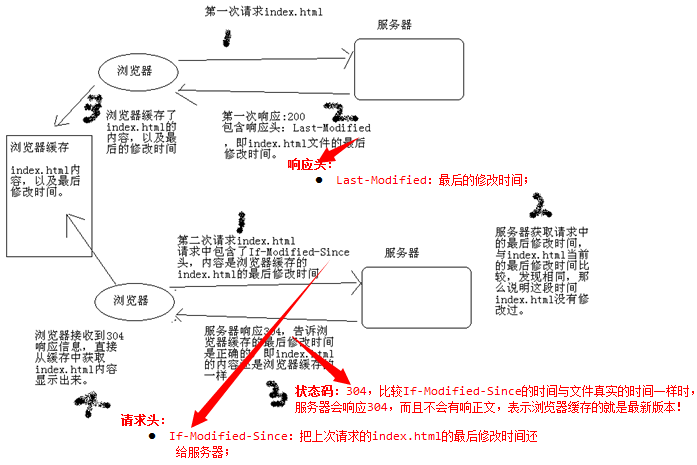
</html>

内容解说

* HTTP/1.1 200 OK：**响应协议为HTTP1.1，状态码为200，表示请求成功，OK是对状态码的解释；**
* Server: Apache-Coyote/1.1：**服务器的版本信息；**
* Content-Type: text/html;charset=UTF-8：**响应体使用的编码为UTF-8；**
* Content-Length: 724：**响应体为724字节**；
* Set-Cookie: JSESSIONID=C97E2B4C55553EAB46079A4F263435A4; Path=/hello：**响应给客户端的Cookie；**
* Date: Wed, 25 Sep 2012 04:15:03 GMT：**响应的时间，这可能会有8小时的时区差；**

1. **说说你熟悉的响应码**

* 200：请求成功，浏览器会把响应体内容（通常是html）显示在浏览器中；
* 404：请求的资源没有找到，说明客户端错误的请求了不存在的资源；
* 500：请求资源找到了，但服务器内部出现了错误；
* 302：重定向，当响应码为302时，表示服务器要求浏览器重新再发一个请求，服务器会发送一个响应头Location，它指定了新请求的URL地址；
* 304：当用户第一次请求index.html时，服务器会添加一个名为Last-Modified响应头，这个头说明了index.html的最后修改时间，浏览器会把index.html内容，以及最后响应时间缓存下来。当用户第二次请求index.html时，在请求中包含一个名为If-Modified-Since请求头，它的值就是第一次请求时服务器通过Last-Modified响应头发送给浏览器的值，即index.html最后的修改时间，If-Modified-Since请求头就是在告诉服务器，我这里浏览器缓存的index.html最后修改时间是这个，您看看现在的index.html最后修改时间是不是这个，如果还是，那么您就不用再响应这个index.html内容了，我会把缓存的内容直接显示出来。而服务器端会获取If-Modified-Since值，与index.html的当前最后修改时间比对，如果相同，服务器会发响应码304，表示index.html与浏览器上次缓存的相同，无需再次发送，浏览器可以显示自己的缓存页面，如果比对不同，那么说明index.html已经做了修改，服务器会响应200。



### JavaWeb三大组件之servlet

1. **什么是servlet，简单的说下servlet的生命周期以及相关的方法？**

①定义：Servlet是JavaWeb的**三大组件之一**，它属于动态资源。Servlet的作用是处理请求，服务器会把接收到的请求交给Servlet来处理，在Servlet中通常需要：

* 接收请求数据；
* 处理请求；
* 完成响应。

②生命周期：服务器会在Servlet第一次被访问时创建Servlet，或者是在服务器启动时创建Servlet。如果服务器启动时就创建Servlet，那么还需要在web.xml文件中配置。也就是说默认情况下，Servlet是在第一次被访问时由服务器创建的。而且一个Servlet类型，服务器只创建一个实例对象。在Servlet被创建后，服务器会马上调用Servlet的void init(ServletConfig)方法。而且一个Servlet的一生。这个方法只会被调用一次。

当服务器每次接收到请求时，都会去调用Servlet的service()方法来处理请求。服务器接收到一次请求，就会调用service() 方法一次，所以service()方法是会被调用多次的。

Servlet是不会轻易离去的，通常都是在服务器关闭时Servlet才会离去！在服务器被关闭时，服务器会去销毁Servlet，在销毁Servlet之前服务器会先去调用Servlet的destroy()方法，我们可以把Servlet的临终遗言放到destroy()方法中

Servlet生命周期相关的方法均由tomcat容易调用！

1. **Servlet接口相关类型**

①在Servlet接口中还存在三个我们不熟悉的类型：

* **ServletRequest**：service() 方法的参数，它表示请求对象，它封装了所有与请求相关的数据，它是由服务器创建的；
* **ServletResponse**：service()方法的参数，它表示响应对象，在service()方法中完成对客户端的响应需要使用这个对象；
* **ServletConfig**：init()方法的参数，它表示Servlet配置对象，它对应Servlet的配置信息，那对应web.xml文件中的<servlet>元素。

**②ServletRequest**和**ServletResponse**是Servlet#service() 方法的两个参数，一个是请求对象，一个是响应对象，可以从ServletRequest对象中获取请求数据，可以使用ServletResponse对象完成响应。你以后会发现，这两个对象就像是一对恩爱的夫妻，永远不分离，总是成对出现。

ServletRequest和ServletResponse的实例由服务器创建，然后传递给service()方法。如果在service() 方法中希望使用HTTP相关的功能，那么可以把ServletRequest和ServletResponse强转成HttpServletRequest和HttpServletResponse。这也说明我们经常需要在service()方法中对ServletRequest和ServletResponse进行强转，这是很心烦的事情。不过后面会有一个类来帮我们解决这一问题的。【**关于servletRequest和servletResponse对象在第n条详细介绍**】

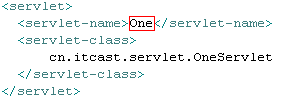
**HttpServletRequest**方法：

* String **getParameter**(String paramName)：获取指定请求参数的值；
* String **getMethod**()：获取请求方法，例如GET或POST；
* String **getHeader**(String name)：获取指定请求头的值；
* void **setCharacterEncoding**(String encoding)：设置请求体的编码！因为GET请求没有请求体，所以这个方法只只对POST请求有效。当调用request.setCharacterEncoding(“utf-8”)之后，再通过getParameter()方法获取参数值时，那么参数值都已经通过了转码，即转换成了UTF-8编码。所以，这个方法必须在调用getParameter()方法之前调用！

**HttpServletResponse**方法：

* PrintWriter **getWriter**()：获取字符响应流，使用该流可以向客户端输出响应信息。例如response.getWriter().print(“<h1>Hello JavaWeb!</h1>”)；
* ServletOutputStream **getOutputStream**()：获取字节响应流，当需要向客户端响应字节数据时，需要使用这个流，例如要向客户端响应图片；
* void **setCharacterEncoding**(String encoding)：用来设置字符响应流的编码，例如在调用setCharacterEncoding(“utf-8”);之后，再response.getWriter()获取字符响应流对象，这时的响应流的编码为utf-8，使用response.getWriter()输出的中文都会转换成utf-8编码后发送给客户端；
* void **setHeader**(String name, String value)：向客户端添加响应头信息，例如setHeader(“Refresh”, “3;url=http://www.itcast.cn”)，表示3秒后自动刷新到http://www.itcast.cn；
* void **setContentType**(String contentType)：该方法是setHeader(“content-type”, “xxx”)的简便方法，即用来添加名为content-type响应头的方法。content-type响应头用来设置响应数据的MIME类型，例如要向客户端响应jpg的图片，那么可以setContentType(“image/jepg”)，如果响应数据为文本类型，那么还要台同时设置编码，例如setContentType(“text/html;chartset=utf-8”)表示响应数据类型为文本类型中的html类型，并且该方法会调用setCharacterEncoding(“utf-8”)方法；
* void **sendError**(int code, String errorMsg)：向客户端发送状态码，以及错误消息。例如给客户端发送404：response(404, “您要查找的资源不存在！”)。

③**ServletConfig**对象对应web.xml文件中的<servlet>元素。例如你想获取当前Servlet在web.xml文件中的配置名，那么可以使用servletConfig.getServletName()方法获取！



ServletConfig对象是由服务器创建的，然后传递给Servlet的init()方法，你可以在init()方法中使用它！

* String getServletName()：获取Servlet在web.xml文件中的配置名称，即<servlet-name>指定的名称；
* ServletContext getServletContext()：用来获取ServletContext对象，ServletContext会在后面讲解；
* String getInitParameter(String name)：用来获取在web.xml中配置的初始化参数，通过参数名来获取参数值；
* Enumeration getInitParameterNames()：用来获取在web.xml中配置的所有初始化参数名称；

在<servlet>元素中还可以配置初始化参数：

|  |
| --- |
| <servlet>  <servlet-name>One</servlet-name>  <servlet-class>cn.itcast.servlet.OneServlet</servlet-class>  <init-param>  <param-name>paramName1</param-name>  <param-value>paramValue1</param-value>  </init-param>  <init-param>  <param-name>paramName2</param-name>  <param-value>paramValue2</param-value>  </init-param>  </servlet> |

在OneServlet中，可以使用ServletConfig对象的getInitParameter()方法来获取初始化参数，例如：String value1 = servletConfig.getInitParameter(“paramName1”);//获取到paramValue1

1. **谈谈与servlet接口相关的实现类**

* **GenericServlet**

①GenericServlet是Servlet接口的实现类，我们可以通过继承GenericServlet来编写自己的Servlet。

GenericServlet.java //**实现了ServletConfig接口，可以直接调用它拥有的方法！**

|  |
| --- |
| **public** **abstract** **class** GenericServlet **implements** Servlet, ServletConfig,  java.io.Serializable {  **private** **static** **final** **long** *serialVersionUID* = 1L;  **private** **transient** ServletConfig config;  **public** GenericServlet() {}  @Override  **public** **void** destroy() {}  @Override  **public** String getInitParameter(String name) {  **return** getServletConfig().getInitParameter(name);  }  @Override  **public** Enumeration<String> getInitParameterNames() {  **return** getServletConfig().getInitParameterNames();  }  @Override  **public** ServletConfig **getServletConfig**() {  **return** config;  }  @Override  **public** ServletContext getServletContext() {  **return** getServletConfig().getServletContext();  }  @Override  **public** String getServletInfo() {  **return** "";  }  **//实现了Servlet的init(ServletConfig)方法，把参数config赋给了本类的成员config，然后再调用本类自己的无参的init()方法。**  @Override  **public** **void** **init**(ServletConfig config) **throws** ServletException {  **this**.config = config;  **this**.init();  }  **//这个方法是GenericServlet自己的方法，而不是从Servlet继承下来的。当我们自定义Servlet时，如果想完成初始化作用就不要再重复init(ServletConfig)方法了，而是应该去重写init()方法。因为在GenericServlet中的init(ServletConfig)方法中保存了ServletConfig对象，如果覆盖了保存ServletConfig的代码，那么就不能再使用ServletConfig了。**  **public** **void** **init**() **throws** ServletException {}  **public** **void** log(String msg) {  getServletContext().log(getServletName() + ": " + msg);  }  **public** **void** log(String message, Throwable t) {  getServletContext().log(getServletName() + ": " + message, t);  }  @Override  **public** **abstract** **void** service(ServletRequest req, ServletResponse res)  **throws** ServletException, IOException;  @Override  **public** String getServletName() {  **return** config.getServletName();  }  } |

②关于GenericServlet的init()方法，在GenericServlet中，定义了一个ServletConfig config实例变量，并在init(ServletConfig)方法中把参数ServletConfig赋给了实例变量。然后在该类的很多方法中使用了实例变量config。

如果子类覆盖了GenericServlet的init(StringConfig)方法，那么this.config=config这一条语句就会被覆盖了，也就是说GenericServlet的实例变量config的值为null，那么所有依赖config的方法都不能使用了。如果真的希望完成一些初始化操作，那么去覆盖GenericServlet提供的init()方法，它是没有参数的init()方法，它会在init(ServletConfig)方法中被调用。

③GenericServlet还实现了ServletConfig接口，所以可以直接调用getInitParameter()、getServletContext()等ServletConfig的方法。

* **HttpServlet**

HttpServlet类是GenericServlet的子类，它提供了对HTTP请求的特殊支持，所以通常我们都会通过继承HttpServlet来完成自定义的Servlet。

HttpServlet类中提供了service(HttpServletRequest,HttpServletResponse)方法，这个方法是HttpServlet自己的方法，不是从Servlet继承来的。在HttpServlet的service(ServletRequest,ServletResponse)方法中会把ServletRequest和ServletResponse强转成HttpServletRequest和HttpServletResponse，然后调用service(HttpServletRequest,HttpServletResponse)方法，这说明子类可以去覆盖service(HttpServletRequest,HttpServletResponse)方法即可，这就不用自己去强转请求和响应对象了。

其实子类也不用去覆盖service(HttpServletRequest,HttpServletResponse)方法，因为HttpServlet还要做另一步简化操作，下面会介绍。

HttpServlet.java

|  |
| --- |
| **public** **abstract** **class** HttpServlet **extends** GenericServlet {  **protected** **void** service(HttpServletRequest req, HttpServletResponse resp)  **throws** ServletException, IOException {  ……  }  @Override  **public** **void** service(ServletRequest req, ServletResponse res)  **throws** ServletException, IOException {  HttpServletRequest request;  HttpServletResponse response;  **try** {  //强转  request = (HttpServletRequest) req;  response = (HttpServletResponse) res;  } **catch** (ClassCastException e) {  **throw** **new** ServletException("non-HTTP request or response");  }  //调用上面的service()方法  service(request, response);  }  ……  } |

此外，在HttpServlet类中，还有2个方法doGet()和doPost()

在HttpServlet的service(HttpServletRequest,HttpServletResponse)方法会去判断当前请求是GET还是POST，如果是GET请求，那么会去调用本类的doGet()方法，如果是POST请求会去调用doPost()方法，这说明我们在子类中去覆盖doGet()或doPost()方法即可。

|  |
| --- |
| **public** **class** AServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)**throws** ServletException, IOException {  System.*out*.println("hello doGet()...");  }  } |
| **public** **class** BServlet **extends** HttpServlet {  **public** **void** doPost(HttpServletRequest request, HttpServletResponse response)**throws** ServletException, IOException {  System.*out*.println("hello doPost()...");  }  } |

1. **说下实现Servlet几种方式？**

①实现javax.servlet.servlet接口

②继承javax.servlet.GenericServlet抽象类

③继承javax.servlet.HttpServlet类

通常我们只需要继承HttpServlet类即可，但是我们还得从javax.servlet.Servlet接口开始学习

1. **Servlet的一些细节问题**

* 不要在Servlet中创建成员！创建局部变量即可！
* 可以创建无状态成员！
* 可以创建有状态的成员，但状态必须为只读的！

①**Servlet不是线程安全的**，因为一个类型的Servlet只有一个实例对象，那么就有可能会出现一个Servlet同时处理多个请求，那么Servlet是否为线程安全的呢？答案是：“不是线程安全的”。这说明Servlet的工作效率很高，但也存在线程安全问题！

所以我们不应该在Servlet中便宜创建成员变量，因为可能会存在一个线程对这个成员变量进行写操作，另一个线程对这个成员变量进行读操作。

②**服务器启动的时候就创建servlet。**默认情况下，服务器会在某个Servlet第一次收到请求时创建它。也可以在web.xml中对Servlet进行配置，使服务器启动时就创建Servlet。

在<servlet>元素中配置<load-on-startup>元素可以让服务器在启动时就创建该Servlet，其中<load-on-startup>元素的值必须是大于等于的整数，它的使用是服务器启动时创建Servlet的顺序。值越小越先启动。

<servlet>

<servlet-name>hello1</servlet-name>

<servlet-class>cn.itcast.servlet.Hello1Servlet</servlet-class>

<load-on-startup>0</load-on-startup>//**非负整数**

</servlet>

③<**url-pattern**>是<servlet-mapping>的子元素，用来指定Servlet的访问路径，即URL。它必须是以“/”开头！

1. 可以在<servlet-mapping>中给出多个<url-pattern>，例如：

|  |
| --- |
| <servlet-mapping>  <servlet-name>AServlet</servlet-name>  <url-pattern>/AServlet</url-pattern>  <url-pattern>/BServlet</url-pattern>  </servlet-mapping> |

那么这说明一个Servlet绑定了两个URL，无论访问/AServlet还是/BServlet，访问的都是AServlet。

1. 还可以在<url-pattern>中使用通配符，所谓通配符就是星号“\*”，星号可以匹配任何URL前缀或后缀，使用通配符可以命名一个Servlet绑定一组URL，例如：

* <url-pattern>/servlet/\*<url-patter>：/servlet/a、/servlet/b，都匹配/servlet/\*；
* <url-pattern>\*.do</url-pattern>：/abc/def/ghi.do、/a.do，都匹配\*.do；
* <url-pattern>/\*<url-pattern>：匹配所有URL；

**他们分别为路径匹配、扩展名匹配以及啥都匹配**

通配符不能在中间，只能有两种方式：

* 路径匹配，星在后面：/servlet/\*
* 扩展名匹配，星在前面：\*.do

请注意，通配符要么为前缀，要么为后缀，不能出现在URL中间位置，也不能只有通配符。例如：/\*.do就是错误的，因为星号出现在URL的中间位置上了。\*.\*也是不对的，因为一个URL中最多只能出现一个通配符。

注意，通配符是一种模糊匹配URL的方式，如果存在更具体的<url-pattern>，那么访问路径会去匹配具体的<url-pattern>。例如：

|  |
| --- |
| <servlet>  <servlet-name>hello1</servlet-name>  <servlet-class>cn.itcast.servlet.Hello1Servlet</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>hello1</servlet-name>  <url-pattern>/servlet/hello1</url-pattern>  </servlet-mapping>  <servlet>  <servlet-name>hello2</servlet-name>  <servlet-class>cn.itcast.servlet.Hello2Servlet</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>hello2</servlet-name>  <url-pattern>/servlet/\*</url-pattern>  </servlet-mapping> |

　　当访问路径为http://localhost:8080/hello/servlet/hello1时，因为访问路径即匹配hello1的<url-pattern>，又匹配hello2的<url-pattern>，但因为hello1的<url-pattern>中没有通配符，所以优先匹配，即设置hello1。

1. **谈谈你对ServletContext对象的认识**

①服务器会为每个应用创建一个ServletContext对象：

* ServletContext对象的创建是在服务器启动时完成的；
* ServletContext对象的销毁是在服务器关闭时完成的。

一个项目只有一个ServletContext对象！我们可以在N多个Servlet中来获取这个唯一的对象，使用它可以给多个Servlet传递数据！与天地同寿！！！这个对象在Tomcat启动时就创建，在Tomcat关闭时才会死去！

 ServletContext对象的作用是在整个Web应用的动态资源之间共享数据！例如在AServlet中向ServletContext对象中保存一个值，然后在BServlet中就可以获取这个值，这就是共享数据了。

②获取ServletContext的几种方式

* ServletConfig#getServletContext()；
* GenericServlet#getServletContext();
* HttpSession#getServletContext()
* ServletContextEvent#getServletContext()

1)在Servlet中获取ServletContext对象：

* 在void init(ServletConfig config)中：

ServletContext context = config.getServletContext();，ServletConfig类的getServletContext()方法可以用来获取ServletContext对象；

2)在GenericeServlet或HttpServlet中获取ServletContext对象：

* GenericServlet类有getServletContext()方法，所以可以直接使用this.getServletContext()来获取；

|  |
| --- |
| public class MyServlet implements Servlet {  public void init(ServletConfig config) {  ServletContext context = config.getServletContext();  }  …  } |
| public class MyServlet extends HttpServlet {  public void doGet(HttpServletRequest request, HttpServletResponse response) {  ServletContext context = this.getServletContext();  }  } |

③ServletContext对象是一个域对象

ServletContext是JavaWeb四大域对象之一：

* PageContext；
* ServletRequest；
* HttpSession；
* ServletContext；

所有域对象都有存取数据的功能，因为域对象内部有一个Map，用来存储数据，下面是ServletContext对象用来操作数据的方法：

* void **setAttribute**(String name, Object value)：用来存储一个对象，也可以称之为存储一个域属性，例如：servletContext.setAttribute(“xxx”, “XXX”)，在ServletContext中保存了一个域属性，域属性名称为xxx，域属性的值为XXX。请注意，如果多次调用该方法，并且使用相同的name，那么会覆盖上一次的值，这一特性与Map相同；
* Object **getAttribute**(String name)：用来获取ServletContext中的数据，当前在获取之前需要先去存储才行，例如：String value = (String)servletContext.getAttribute(“xxx”);，获取名为xxx的域属性；
* void **removeAttribute**(String name)：用来移除ServletContext中的域属性，如果参数name指定的域属性不存在，那么本方法什么都不做；
* Enumeration **getAttributeNames**()：获取所有域属性的名称；

④获取应用初始化参数

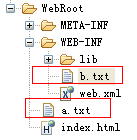
* Servlet也可以获取初始化参数，但它是局部的参数；也就是说，一个Servlet只能获取自己的初始化参数，不能获取别人的，即初始化参数只为一个Servlet准备！
* 可以配置公共的初始化参数，为所有Servlet而用！这需要使用ServletContext才能使用！

还可以使用ServletContext来获取在web.xml文件中配置的应用初始化参数！注意，应用初始化参数与Servlet初始化参数不同：

web.xml

|  |
| --- |
| <web-app ...>  ...  //**配置了两个应用初始化参数**  <context-param>  <param-name>paramName1</param-name>  <param-value>paramValue1</param-value>  </context-param>  <context-param>  <param-name>paramName2</param-name>  <param-value>paramValue2</param-value>  </context-param>  </web-app> |
| /**/获取ServletContext对象**  ServletContext context = **this**.getServletContext();  //**通过参数名获取参数值**  String value1 = context.getInitParameter("paramName1");  String value2 = context.getInitParameter("paramName2");  System.*out*.println(value1 + ", " + value2);  //**获取所有应用初始化参数名称**  Enumeration names = context.getInitParameterNames();  **while**(names.hasMoreElements()) {  System.*out*.println(names.nextElement());  } |

⑤获取资源相关的方法



1)获取真实路径（\*\*\*\*\*）

还可以使用ServletContext对象来获取Web应用下的资源，例如在hello应用的根目录下创建a.txt文件，现在想在Servlet中获取这个资源，就可以使用ServletContext来获取。

* 获取a.txt的真实路径：String realPath = servletContext.getRealPath(“/a.txt”)，realPath的值为a.txt文件的绝对路径：F:\tomcat6\webapps\hello\a.txt；
* 获取b.txt的真实路径：

String realPath = servletContext.getRealPath(“/WEB-INF/b.txt”)；

2)获取资源流

不只可以获取资源的路径，还可以通过ServletContext获取资源流，即把资源以输入流的方式获取：

* 获取a.txt资源流：InputStream in = servletContext.getResourceAsStream(“/a.txt”)；
* 获取b.txt资源流：

InputStream in = servletContext.getResourceAsStream(“/WEB-INF/b.txt”)；

3)获取指定目录下所有资源路径

还可以使用ServletContext获取指定目录下所有资源路径，例如获取/WEB-INF下所有资源的路径：

|  |
| --- |
| Set set = context.getResourcePaths("/WEB-INF");  System.*out*.println(set); |
| [/WEB-INF/lib/, /WEB-INF/classes/, /WEB-INF/b.txt, /WEB-INF/web.xml] |

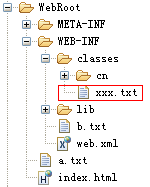
注意，本方法必须以“/”开头！！！

1. **如何获取类路径下的资源？**

获取类路径资源，类路径对一个JavaWeb项目而言，就是/WEB-INF/classes和/WEB-INF/lib/每个jar包！主要有下面两种方式进行获取：

* **Class**
* **ClassLoader**：

　　这里要讲的是获取类路径下的资源，对于JavaWeb应用而言，就是获取classes目录下的资源。



|  |
| --- |
| InputStream in = **this**.getClass().getResourceAsStream("/xxx.txt");  System.*out*.println(IOUtils.*toString*(in)); |
| InputStream in = **this**.getClass().getClassLoader().getResourceAsStream("xxx.txt");  System.*out*.println(IOUtils.*toString*(in)); |

* Class类的getResourceAsStream(String path)：
* 路径以“/”开头，相对classes路径；
* 路径不以“/”开头，相对当前class文件所有路径，例如在cn.itcast.servlet.MyServlet中执行，那么相对/classes/cn/itcast/servlet/路径；
* ClassLoader类的getResourceAsStream(String path)：
* 相对classes路径；

1. **请说一下response对象以及它的一些功能？**

response是Servlet.service方法的一个参数，类型为javax.servlet.http.HttpServletResponse。在客户端发出每个请求时，服务器都会创建一个response对象，并传入给Servlet.service()方法。response对象是用来对客户端进行响应的，这说明在service()方法中使用response对象可以完成对客户端的响应工作。

response对象的功能分为以下四种：

* 设置响应头信息；
* 发送状态码；
* 设置响应正文；
* 重定向；

##### ① 设置响应头信息

可以使用response对象的setHeader()方法来设置响应头！使用该方法设置的响应头最终会发送给客户端浏览器！

* response.setHeader(“content-type”, “text/html;charset=utf-8”)：设置content-type响应头，该头的作用是告诉浏览器响应内容为html类型，编码为utf-8。而且同时会设置response的字符流编码为utf-8，即response.setCharaceterEncoding(“utf-8”)；

response.setHeader("Refresh","5; URL=http://www.itcast.cn")：5秒后自动跳转到传智主页。

##### ② 设置发送状态码

* response.setContentType("text/html;charset=utf-8")：

等同与调用response.setHeader(“content-type”, “text/html;charset=utf-8”)；

* response.setCharacterEncoding(“utf-8”)：设置字符响应流的字符编码为utf-8；
* response.setStatus(200)：设置状态码；
* response.sendError(404, “您要查找的资源不存在”)：当发送错误状态码时，Tomcat会跳转到固定的错误页面去，但可以显示错误信息。

##### ③ 设置响应正文

向客户端输出响应正文（响应体）可以使用response的响应流，repsonse一共提供了两个响应流对象：

* PrintWriter out = response.getWriter()：获取字符流；
* ServletOutputStream out = response.getOutputStream()：获取字节流；

当然，如果响应正文内容为字符，那么使用response.getWriter()，如果响应内容是字节，例如下载时，那么可以使用response.getOutputStream()。

注意，在一个请求中，不能同时使用这两个流！也就是说，要么你使用repsonse.getWriter()，要么使用response.getOutputStream()，但不能同时使用这两个流。不然会抛出[IllegalStateException](http://java.sun.com/j2se/1.5/docs/api/java/lang/IllegalStateException.html" \o "class or interface in java.lang)异常。

**字符响应流response.getWriter()的两个问题**

1. 字符编码

在使用response.getWriter()时需要注意默认字符编码为ISO-8859-1，如果希望设置字符流的字符编码为utf-8，可以使用response.setCharaceterEncoding(“utf-8”)来设置。这样可以保证输出给客户端的字符都是使用UTF-8编码的！

但客户端浏览器并不知道响应数据是什么编码的！如果希望通知客户端使用UTF-8来解读响应数据，那么还是使用response.setContentType("text/html;charset=utf-8")方法比较好，因为这个方法不只会调用response.setCharaceterEncoding(“utf-8”)，还会设置content-type响应头，客户端浏览器会使用content-type头来解读响应数据。

2)缓冲区

response.getWriter()是PrintWriter类型，所以它有缓冲区，缓冲区的默认大小为8KB。也就是说，在响应数据没有输出8KB之前，数据都是存放在缓冲区中，而不会立刻发送到客户端。当Servlet执行结束后，服务器才会去刷新流，使缓冲区中的数据发送到客户端。

如果希望响应数据马上发送给客户端：

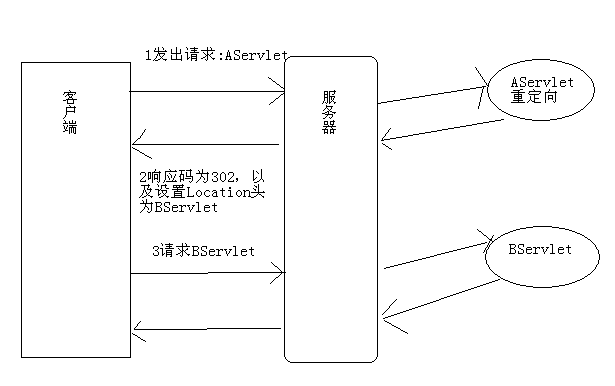
* 向流中写入大于8KB的数据；
* 调用response.flushBuffer()方法来手动刷新缓冲区；

##### ④ 重定向

1)**什么是重定向**

当你访问http://www.sun.com时，你会发现浏览器地址栏中的URL会变成http://www.oracle.com/us/sun/index.htm，这就是重定向了。

重定向是服务器通知浏览器去访问另一个地址，即再发出另一个请求。



2)**完成重定向**

响应码为200表示响应成功，而响应码为302表示重定向。所以完成重定向的第一步就是设置响应码为302。

因为重定向是通知浏览器再第二个请求，所以浏览器需要知道第二个请求的URL，所以完成重定向的第二步是设置Location头，指定第二个请求的URL地址。

|  |
| --- |
| **public** **class** AServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  //**设置响应码为302表示重定向**  response.setStatus(302);  //**设置新请求的RUL**  response.setHeader("Location", "http://www.itcast.cn");  }  } |

　　上面代码的作用是：当访问AServlet后，会通知浏览器重定向到传智主页。客户端浏览器解析到响应码为302后，就知道服务器让它重定向，所以它会马上获取响应头Location，然发出第二个请求。

3)**便捷的重定向方式**

|  |
| --- |
| **public** **class** AServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  response.sendRedirect("http://www.itcast.cn");  }  } |

response.sendRedirect()方法会设置响应头为302，以设置Location响应头。

如果要重定向的URL是在同一个服务器内，那么可以使用相对路径，例如：

|  |
| --- |
| **public** **class** AServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  response.sendRedirect("/hello/BServlet");  }  } |

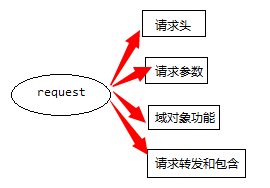
重定向的URL地址为：http://localhost:8080/hello/BServlet。

4)**重定向小结**

* 重定向是两次请求；
* 重定向的URL可以是其他应用，不局限于当前应用；
* 重定向的响应头为302，并且必须要有Location响应头；
* 重定向就不要再使用response.getWriter()或response.getOutputStream()输出数据，不然可能会出现异常；

1. **Request有哪些功能？**

request是Servlet.service()方法的一个参数，类型为javax.servlet.http.HttpServletRequest。在客户端发出每个请求时，服务器都会创建一个request对象，并把请求数据封装到request中，然后在调用Servlet.service()方法时传递给service()方法，这说明在service()方法中可以通过request对象来获取请求数据。



request的功能可以分为以下几种：

* 封装了请求头数据；
* 封装了请求正文数据，如果是GET请求，那么就没有正文；
* request是一个域对象，可以把它当成Map来添加获取数据；
* request提供了请求转发和请求包含功能。

##### ① request获取请求头数据

request与请求头相关的方法有：

* String getHeader(String name)：获取指定名称的请求头；
* Enumeration getHeaderNames()：获取所有请求头名称；
* int getIntHeader(String name)：获取值为int类型的请求头。

##### ② request获取请求参数

最为常见的客户端传递参数方式有两种：

* 浏览器地址栏直接输入：一定是GET请求；
* 超链接：一定是GET请求；
* 表单：可以是GET，也可以是POST，这取决与<form>的method属性值；

**GET请求和POST请求的区别：**

* GET请求：
* 请求参数会在浏览器的地址栏中显示，所以不安全；
* 请求参数长度限制长度在1K之内；
* GET请求没有请求体，无法通过request.setCharacterEncoding()来设置参数的编码；
* POST请求：
* 请求参数不会显示浏览器的地址栏，相对安全；
* 请求参数长度没有限制；

|  |
| --- |
| //**hello是应用名，ParamServlet是Servlet绑定URL路径,号后面的是请求参数**  <a href=*"/hello/ParamServlet?p1=v1&p2=v2"*>超链接</a>  <hr/>  <form action=*"/hello/ParamServlet"* method=*"post"*>  参数1：<input type=*"text"* name=*"p1"*/><br/>  参数2：<input type=*"text"* name=*"p2"*/><br/>  <input type=*"submit"* value=*"提交"*/>  </form> |
|  |

下面是使用request获取请求参数的API：

* String getParameter(String name)：通过指定名称获取参数值；

|  |
| --- |
| //**点击超链接是GET请求，所以会执行doGet()方法**  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  String v1 = request.getParameter("p1");  String v2 = request.getParameter("p2");  System.*out*.println("p1=" + v1);  System.*out*.println("p2=" + v2);  }  //**提交表单是POST请求，所以会调用doPost()方法**  **public** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  String v1 = request.getParameter("p1");  String v2 = request.getParameter("p2");  System.*out*.println("p1=" + v1);  System.*out*.println("p2=" + v2);  } |

* String[] getParameterValues(String name)：当多个参数名称相同时，可以使用方法来获取；

|  |
| --- |
| <a href=*"/hello/ParamServlet?name=zhangSan&name=liSi"*>超链接</a> |
| **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  //**获取所有名为name的参数值，输出结果为：[zhangSan, liSi]**  String[] names = request.getParameterValues("name");  System.*out*.println(Arrays.*toString*(names));  } |

* Enumeration getParameterNames()：获取所有参数的名字；

|  |
| --- |
| <form action=*"/hello/ParamServlet"* method=*"post"*>  参数1：<input type=*"text"* name=*"p1"*/><br/>  参数2：<input type=*"text"* name=*"p2"*/><br/>  <input type=*"submit"* value=*"提交"*/>  </form> |
| **public** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  //**获取所有参数名称，输出结果为：p1和p2**  Enumeration names = request.getParameterNames();  **while**(names.hasMoreElements()) {  System.*out*.println(names.nextElement());  }  } |

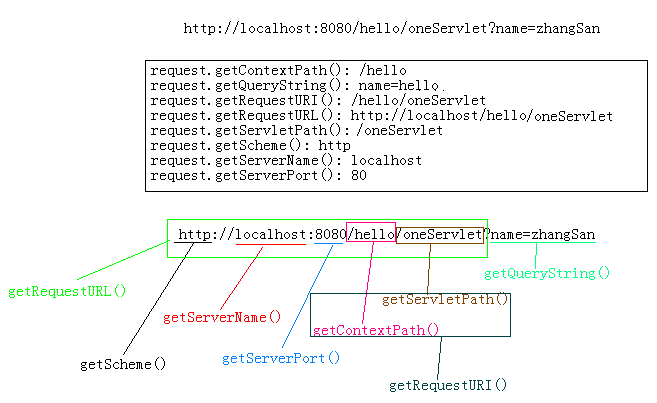
* Map getParameterMap()：获取所有参数封装到Map中，其中key为参数名，value为参数值，因为一个参数名称可能有多个值，所以参数值是String[]，而不是String。

|  |
| --- |
| <a href=*"/day05\_1/ParamServlet?p1=v1&p1=vv1&p2=v2&p2=vv2"*>超链接</a> |
| Map<String,String[]> paramMap = request.getParameterMap();  **for**(String name : paramMap.keySet()) {  String[] values = paramMap.get(name);  System.*out*.println(name + ": " + Arrays.*toString*(values));  } |
| p2: [v2, vv2]  p1: [v1, vv1] |

##### ③ request获取请求相关的其它方法

request中还提供了与请求相关的其他方法，有些方法是为了我们更加便捷的方法请求头数据而设计，有些是与请求URL相关的方法。

* int getContentLength()：获取请求体的字节数，GET请求没有请求体，没有请求体返回-1；
* String getContentType()：获取请求类型，如果请求是GET，那么这个方法返回null；如果是POST请求，那么默认为application/x-www-form-urlencoded，表示请求体内容使用了URL编码；
* String getMethod()：返回请求方法，例如：GET
* Locale getLocale()：返回当前客户端浏览器的Locale。java.util.Locale表示国家和言语，这个东西在国际化中很有用；
* String getCharacterEncoding()：获取请求编码，如果没有setCharacterEncoding()，那么返回null，表示使用ISO-8859-1编码；
* void setCharacterEncoding(String code)：设置请求编码，只对请求体有效！注意，对于GET而言，没有请求体！！！所以此方法只能对POST请求中的参数有效！
* String getContextPath()：返回上下文路径，例如：/hello
* String getQueryString()：返回请求URL中的参数，例如：name=zhangSan
* String getRequestURI()：返回请求URI路径，例如：/hello/oneServlet
* StringBuffer getRequestURL()：返回请求URL路径，例如：http://localhost/hello/oneServlet，即返回除了参数以外的路径信息；
* String getServletPath()：返回Servlet路径，例如：/oneServlet
* String getRemoteAddr()：返回当前客户端的IP地址；
* String getRemoteHost()：返回当前客户端的主机名，但这个方法的实现还是获取IP地址；
* String getScheme()：返回请求协议，例如：http；
* String getServerName()：返回主机名，例如：localhost
* int getServerPort()：返回服务器端口号，例如：8080



可以使用request.getRemoteAddr()方法获取客户端的IP地址，然后判断IP是否为禁用IP。

|  |
| --- |
| String ip = request.getRemoteAddr();  System.*out*.println(ip);  **if**(ip.equals("127.0.0.1")) {  response. getWriter().print("您的IP已被禁止！");  } **else** {  response.getWriter().print("Hello!");  } |

##### ④ request域方法

request是域对象！在JavaWeb中一共四个域对象，其中ServletContext就是域对象，它在整个应用中只创建一个ServletContext对象。request其中一个，request可以在一个请求中共享数据。

一个请求会创建一个request对象，如果在一个请求中经历了多个Servlet，那么多个Servlet就可以使用request来共享数据。现在我们还不知道如何在一个请求中经历之个Servlet，后面在学习请求转发和请求包含后就知道了。

**下面是request的域方法**：

* void **setAttribute**(String name, Object value)：用来存储一个对象，也可以称之为存储一个域属性，例如：servletContext.setAttribute(“xxx”, “XXX”)，在request中保存了一个域属性，域属性名称为xxx，域属性的值为XXX。请注意，如果多次调用该方法，并且使用相同的name，那么会覆盖上一次的值，这一特性与Map相同；
* Object **getAttribute**(String name)：用来获取request中的数据，当前在获取之前需要先去存储才行，例如：String value = (String)request.getAttribute(“xxx”);，获取名为xxx的域属性；
* void **removeAttribute**(String name)：用来移除request中的域属性，如果参数name指定的域属性不存在，那么本方法什么都不做；
* Enumeration getAttributeNames()：获取所有域属性的名称；

##### ⑤ 请求转发和请求包含

无论是请求转发还是请求包含，都表示由多个Servlet共同来处理一个请求。例如Servlet1来处理请求，然后Servlet1又转发给Servlet2来继续处理这个请求。

**1)请求转发**

在AServlet中，把请求转发到BServlet：

|  |
| --- |
| **public** **class** AServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  System.*out*.println("AServlet");  //**获取“调度器”，其中参数为BServlet绑定的URL，即BServlet的<url-pattern>值。**  RequestDispatcher rd = request.getRequestDispatcher("/BServlet");  //**调用“调度器”的转发方法，该方法等同与告诉服务器，去调用BServlet的service()方法一样。**  rd.forward(request, response);  }  } |
| **public** **class** BServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  System.*out*.println("BServlet");  }  } |
| Aservlet  BServlet |

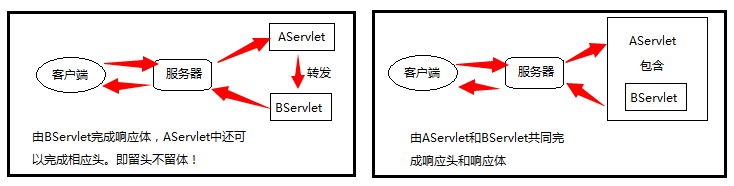
**2)请求包含**

在AServlet中，把请求包含到BServlet：

|  |
| --- |
| **public** **class** AServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  System.*out*.println("AServlet");  RequestDispatcher rd = request.getRequestDispatcher("/BServlet");  //**请求包含**  rd.include(request, response);  }  } |
| **public** **class** BServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  System.*out*.println("BServlet");  }  } |
| Aservlet  BServlet |

1. **请求转发和请求包含的异同？**

* 如果在AServlet中请求转发到BServlet，那么在AServlet中就不允许再输出响应体，即不能再使用response.getWriter()和response.getOutputStream()向客户端输出，这一工作应该由BServlet来完成；如果是使用请求包含，那么没有这个限制；
* 请求转发虽然不能输出响应体，但还是可以设置响应头的，例如：response.setContentType(”text/html;charset=utf-8”);
* 请求包含大多是应用在JSP页面中，完成多页面的合并；
* 请求转发大多是应用在Servlet中，转发目标大多是JSP页面；



1. **请求转发和重定向的异同？**

* 请求转发是一个请求，而重定向是两个请求；
* 请求转发后浏览器地址栏不会有变化，而重定向会有变化，因为重定向是两个请求；
* 请求转发的目标只能是本应用中的资源，重定向的目标可以是其他应用；
* 请求转发对AServlet和BServlet的请求方法是相同的，即要么都是GET，要么都是POST，因为请求转发是一个请求；
* 重定向的第二个请求一定是GET；

1. **请你简单给我说一下请求响应的一个大概流程(servlet)**



1. **说说你知道的与路径相关的一些操作**

* 超链接
* 表单
* 转发
* 包含
* 重定向
* <url-pattern>
* ServletContext获取资源
* Class获取资源
* ClassLoader获取资源

1. **谈谈你对客户端路径、服务器端路径以及<url-pattern>路径的认识**

##### ① <url-pattern>路径

它必须使用“/”开头，并且相对的是当前应用。

##### ② 客户端路径

超链接、表单、重定向都是客户端路径，客户端路径可以分为三种方式：

* 绝对路径；
* 以“/”开头的相对路径；
* 不以“/”开头的相对路径；

例如：http://localhost:8080/hello1/pages/a.html中的超链接和表单如下：

|  |
| --- |
| 绝对路径：<a href=*"http://localhost:8080/hello2/index.html"*>链接1</a>  客户端路径：<a href=*"/hello3/pages/index.html"*>链接2</a>  相对路径：<a href=*"index.html"*>链接3</a>  <hr/>  绝对路径：  <form action=*"http://localhost:8080/hello2/index.html"*>  <input type=*"submit"* value=*"表单1"*/>  </form>  客户端路径：  <form action=*"/hello2/index.html"*>  <input type=*"submit"* value=*"表单2"*/>  </form>  相对路径：  <form action=*"index.html"*>  <input type=*"submit"* value=*"表单3"*/>  </form> |

* 链接1和表单1：没什么可说的，它使用绝对路径；
* 链接2和表单2：以“/”开头，相对主机，与当前a.html的主机相同，即最终访问的页面为**http://localhost:8080**/hello2/index.html；
* 链接3和表单3：不以“/”开头，相对当前页面的路径，即a.html所有路径，即最终访问的路径为：**http://localhost:8080/hello1/pages/**index.html；

重定向1：

|  |
| --- |
| **public** **class** AServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  response.sendRedirect("**/hello/index.html**");  }  } |

　　假设访问AServlet的路径为：http://localhost:8080/hello/servlet/AServlet

　　因为路径以“/”开头，所以相对当前主机，即**http://localhost:8080**/hello/index.html。

重定向2：

|  |
| --- |
| **public** **class** AServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  response.sendRedirect("**index.html**");  }  } |

假设访问AServlet的路径为：http://localhost:8080/hello/servlet/AServlet

因为路径不以“/”开头，所以相对当前路径，

即**http://localhost:8080/hello/servlet/**index.html

强烈建议使用“/”开头的路径，这说明在页面中的超链接和表单都要以“/”开头，后面是当前应用的名称，再是访问路径：

<form action="/hello/servlet/AServlet"></form>

<a href="/hello/b.html">链接</a>

其中/hello是当前应用名称，这也说明如果将来修改了应用名称，那么页面中的所有路径也要修改，这一点确实是个问题。这一问题的处理方案会在学习了JSP之后讲解！

在Servlet中的重定向也建议使用“/”开头。同理，也要给出应用的名称！例如：

|  |
| --- |
| response.sendRedirect("/hello/BServlet"); |

其中/hello是当前应用名，如果将来修改了应用名称，那么也要修改所有重定向的路径，这一问题的处理方案是使用request.getContextPath()来获取应用名称。

|  |
| --- |
| response.sendRedirect(request.getContextPath() + "/BServlet"); |

##### ③ 服务器端路径

服务器端路径必须是相对路径，不能是绝对路径。但相对路径有两种形式：

* 以“/”开头；
* 不以“/”开头；

其中请求转发、请求包含都是服务器端路径，服务器端路径与客户端路径的区别是：

* 客户端路径以“/”开头：相对当前主机；
* 服务器端路径以“/”开头：相对当前应用；

转发1：

|  |
| --- |
| **public** **class** AServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  request.getRequestDispatcher("/BServlet").forward(request, response);  }  } |

假设访问AServlet的路径为：http://localhost:8080/hello/servlet/AServlet

因为路径以“/”开头，所以相对当前应用，即http://localhost:8080/hello/BServlet。

转发2：

|  |
| --- |
| **public** **class** AServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  request.getRequestDispatcher("BServlet").forward(request, response);  }  } |

假设访问AServlet的路径为：http://localhost:8080/hello/servlet/AServlet

因为路径不以“/”开头，所以相对当前应用，

即http://localhost:8080/hello/servlet/BServlet。

1. **获取资源的一些路径问题**

##### ① ServletContext获取资源

**必须是相对路径**，可以“/”开头，也可以不使用“/”开头，但无论是否使用“/”开头都**是相对当前应用路径。**

例如在AServlet中获取资源，AServlet的路径路径为：http://localhost:8080/hello/servlet/AServlet：

|  |
| --- |
| **public** **class** AServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  String path1 = **this**.getServletContext().getRealPath("a.txt");  String path2 = **this**.getServletContext().getRealPath("/a.txt");  System.*out*.println(path1);  System.*out*.println(path2);  }  } |

path1和path2是相同的结果：http://localhost:8080/hello/a.txt

##### ② Class获取资源

**Class获取资源也必须是相对路径**，可以“/”开头，也可以不使用“/”开头。

|  |
| --- |
| **package** cn.itcast;  **import** java.io.InputStream;  **public** **class** Demo {  **public** **void** fun1() {  InputStream in = Demo.**class**.getResourceAsStream("/a.txt");  }  **public** **void** fun2() {  InputStream in = Demo.**class**.getResourceAsStream("a.txt");  }  } |

其中fun1()方法获取资源时以“/”开头，那么相对的是当前类路径，即/hello/WEB-INF/classes/a.txt文件；

其中fun2()方法获取资源时没有以“/”开头，那么相对当前Demo.class所在路径，因为Demo类在cn.itcast包下，所以资源路径为：/hello/WEB-INF/classes/cn/itcast/a.txt。

##### ③ ClassLoader获取资源

**ClassLoader获取资源也必须是相对路径**，可以“/”开头，也可以不使用“/”开头。但无论是否以“/”开头，**资源都是相对当前类路径**。

|  |
| --- |
| **public** **class** Demo {  **public** **void** fun1() {  InputStream in = Demo.**class**.getClassLoader().getResourceAsStream("/a.txt");  }  **public** **void** fun2() {  InputStream in = Demo.**class**.getClassLoader().getResourceAsStream("a.txt");  }  } |

　　fun1()和fun2()方法的资源都是相对类路径，即classes目录，即/hello/WEB-INF/classes/a.txt

1. **一些编码问题**

##### ① 请求编码

1)直接在地址栏中给出中文

请求数据是由客户端浏览器发送服务器的，请求数据的编码是由浏览器决定的。例如在浏览器地址栏中给出：http://localhost:8080/hello/AServlet?name=传智，那么其中“传智”是什么编码的呢？不同浏览器使用不同的编码，所以这是不确定的！

* IE：使用GB2312；
* FireFox：使用GB2312；
* Chrome：使用UTF-8；

　　通常没有哪个应用要求用户在浏览器地址栏中输入请求数据的，所以大家只需了解一下即可。

2)在页面中发出请求

通常向服务器发送请求数据都需要先请求一个页面，然后用户在页面中输入数据。页面中有超链接和表单，通过超链接和表单就可以向服务器发送数据了。

因为页面是服务器发送到客户端浏览器的，所以这个页面本身的编码由服务器决定。而用户在页面中输入的数据也是由页面本身的编码决定的。

index.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <title>index.html</title>  //这就相当于给客户端发送了响应头content-type，指定当前页面的编码为utf-8  <meta http-equiv=*"content-type"* content=*"text/html; charset=UTF-8"*>  </head>  <body>  <form action=*"/hello/servlet/AServlet"*>  名称:<input type=*"text"* name=*"name"*/>  <input type=*"submit"* value=*"提交"*/>  </form>  <a href=*"/hello/servlet/AServlet?name=传智"*>链接</a>  </body>  </html> |

当用户在index.html页面中输入数据时，都是UTF-8列表的。因为这个页面本身就是UTF-8编码的！

**页面的编译就是页面中输入数据的编码。**

3)GET请求解读编码

当客户端通过GET请求发送数据给服务器时，使用request.getParameter()获取的数据是被服务器误认为ISO-8859-1编码的，也就是说客户端发送过来的数据无论是UTF-8还是GBK，服务器都认为是ISO-8859-1，这就说明我们需要在使用request.getParameter()获取数据后，再转发成正确的编码。

例如客户端以UTF-8发送的数据，使用如下转码方式：

String name = request.getParameter(“name”);

name = new String(name.getBytes(“iso-8859-1”), “utf-8”);

4)POST请求解读编码

　　当客户端通过POST请求发送数据给服务器时，可以在使用request.getParameter()获取请求参数之前先通过request.setCharacterEncoding()来指定编码，然后再使用reuqest.getParameter()方法来获取请求参数，那么就是用指定的编码来读取了。

也就是说，如果是POST请求，服务器可以指定编码！但如果没有指定编码，那么默认还是使用ISO-8859-1来解读。

request.setCharacterEncoding(“utf-8”);

String name = request.getParameter(“name”);

##### ② 响应编码

响应：服务器发送给客户端数据！响应是由response对象来完成，如果响应的数据不是字符数据，那么就无需去考虑编码问题。当然，如果响应的数据是字符数据，那么就一定要考虑编码的问题了。

response.getWriter().print(“传智”);

上面代码因为没有设置repsonse.getWriter()字符流的编码，所以服务器使用默认的编码（ISO-8859-1）来处理，因为ISO-8859-1不支持中文，所以一定会出现编码的。

所以在使用response.getWriter()发送数据之前，一定要设置response.getWriter()的编码，这需要使用response.setCharacterEncoding()方法：

response.setCharacterEncoding(“utf-8”);

response.getWriter().print(“传智”);

上面代码因为在使用response.getWriter()输出之前已经设置了编码，所以输出的数据为utf-8编码。但是，因为没有告诉浏览器使用什么编码来读取响应数据，所以很可能浏览器会出现错误的解读，那么还是会出现乱码的。当然，通常浏览器都支持来设置当前页面的编码，如果用户在看到编码时，去设置浏览器的编码，如果设置的正确那么乱码就会消失。但是我们不能让用户总去自己设置编码，而且应该直接通知浏览器，服务器发送过来的数据是什么编码，这样浏览器就直接使用服务器告诉他的编码来解读！这需要使用content-type响应头。

response.setContentType(“text/html;charset=utf-8”);

response.getWriter().print(“传智”);

　　上面代码使用setContentType()方法设置了响应头content-type编码为utf-8，这不只是在响应中添加了响应头，还等于调用了一次response.setCharacterEncoding(“utf-8”)，也就是说，通过我们只需要调用一次response.setContentType(“text/html;charset=utf-8”)即可，而无需再去调用response.setCharacterEncoding(“utf-8”)了。

在静态页面中，使用<meta>来设置content-type响应头，例如：

<meta http-equiv="content-type" content="text/html; charset=UTF-8">

##### ③ URL编码

通过页面传输数据给服务器时，如果包含了一些特殊字符是无法发送的。这时就需要先把要发送的数据转换成URL编码格式，再发送给服务器。

其实需要我们自己动手给数据转换成URL编码的只有GET超链接，因为表单发送数据会默认使用URL编码，也就是说，不用我们自己来编码。

例如：“传智”这两个字通过URL编码后得到的是：“%E4%BC%A0%E6%99%BA”。URL编码是先需要把“传智”转换成字节，例如我们现在使用UTF-8把“传智”转换成字符，得到的结果是：“[-28, -68, -96, -26, -103, -70]”，然后再把所有负数加上256，得到[228, 188, 160, 230, 153, 186]，再把每个int值转换成16进制，得到[E4, BC, A0, E6, 99, BA]，最后再每个16进制的整数前面加上“%”。

通过URL编码，把“传智”转换成了“%E4%BC%A0%E6%99%BA”，然后发送给服务器！服务器会自动识别出数据是使用URL编码过的，然后会自动把数据转换回来。

当然，在页面中我们不需要自己去通过上面的过程把“传智”转换成“%E4%BC%A0%E6%99%BA”，而是使用Javascript来完成即可。当后面我们学习了JSP后，就不用再使用Javascript了。

|  |
| --- |
| <script type=*"text/javascript"*>  **function** \_go() {  location = "/day05\_2/AServlet?name=" + encodeURIComponent("传智+播客");  }  </script> |
| <a href=*"javascript:\_go();"*>链接</a> |

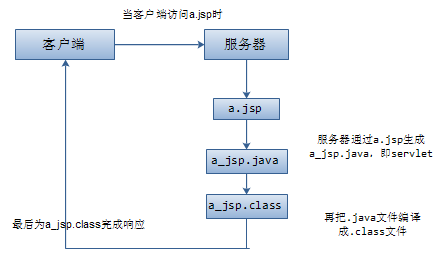
因为URL默认只支持ISO-8859-1，这说明在URL中出现中文和一些特殊字符可能无法发送到服务器。所以我们需要对包含中文或特殊字符的URL进行URL编码。

服务器会自动识别数据是否使用了URL编码，如果使用了服务器会自动把数据解码，无需我们自己动手解码。

1. **Jsp原理**

①jsp是特殊的servlet

当JSP页面首次被访问时，容器（Tomcat）会先把JSP编译成Servlet，然后再去执行Servlet。所以JSP其实就是一个Servlet！



②jsp真身存放目录

JSP生成的Servlet存放在${CATALANA}/work目录下，我经常开玩笑的说，它是JSP的“真身”。我们打开看看其中的内容，了解一下JSP的“真身”。

你会发现，在JSP中的静态信息（例如<html>等）在“真身”中都是使用out.write()完成打印！这些静态信息都是作为字符串输出给了客户端。

JSP的整篇内容都会放到名为\_jspService的方法中！你可能会说<@page>不在“真身”中，<%@page>我们明天再讲。

a\_jsp.java的\_jspService()方法：

|  |
| --- |
| public void **\_jspService**(final javax.servlet.http.HttpServletRequest **request**,  final javax.servlet.http.HttpServletResponse **response**)  throws java.io.IOException, javax.servlet.ServletException {  final javax.servlet.jsp.PageContext **pageContext**;  javax.servlet.http.HttpSession **session** = null;  final javax.servlet.ServletContext **application**;  final javax.servlet.ServletConfig **config**;  javax.servlet.jsp.JspWriter **out** = null;  final java.lang.Object **page** = this;  javax.servlet.jsp.JspWriter \_jspx\_out = null;  javax.servlet.jsp.PageContext \_jspx\_page\_context = null;  try {  response.setContentType("text/html;charset=UTF-8");  pageContext = \_jspxFactory.getPageContext(this, request, response,  null, true, 8192, true);  \_jspx\_page\_context = pageContext;  application = pageContext.getServletContext();  config = pageContext.getServletConfig();  session = pageContext.getSession();  out = pageContext.getOut();  \_jspx\_out = out;    //**JSP页面中的内容都会在这个位置出现！这时上面所说的对象已经创建完了，所以在JSP页面中是可以使用的。**  …  } |

1. **Jsp脚本有哪些？Jsp注释是怎么回事？**

##### ① SP脚本一共三种形式：

* <%...%>：内容会直接放到“真身”中；方法能放什么，他就能放什么。
* <%=…%>：内容会放到out.print()中，作为out.print()的参数；
* <%!…%>：内容会放到\_jspService()方法之外，被类直接包含；

前面已经讲解了<%...%>和<%=…%>，但还没有讲解<%!...%>的作用！

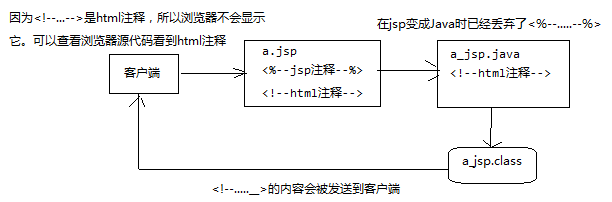
现在我们已经知道了，JSP其实就是一个类，一个Servlet类。<%!...%>的作用是在类中添加方法或成员的，所以<%!...%>中的内容不会出现在\_jspService()中。【自己可以测试查看jsp真身。】

|  |
| --- |
| <%!  **private** String name;  **public** String hello() {  **return** "hello JSP!";  }  %> |

##### ② jsp注释

<%-- ... --%>中的内容在JSP编译成.java时会被忽略的，即JSP注释。

也可以在JSP页面中使用html注释：<!-- … -->，但这个注释在JSP编译成的.java中是存在的，它不会被忽略，而且会被发送到客户端浏览器。但是在浏览器显示服务器发送过来的html时，因为<!-- … -->是html的注释，所以浏览器是不会显示它的。



1. **什么是会话跟踪技术？**

我们需要先了解一下什么是会话！可以把会话理解为客户端与服务器之间的一次会晤，在一次会晤中可能会包含多次请求和响应。例如你给10086打个电话，你就是客户端，而10086服务人员就是服务器了。从双方接通电话那一刻起，会话就开始了，到某一方挂断电话表示会话结束。在通话过程中，你会向10086**发出多个请求，那么这多个请求都在一个会话中。**

在JavaWeb中，客户向某一服务器发出第一个请求开始，会话就开始了，直到客户关闭了浏览器会话结束。

在一个会话的多个请求中共享数据，这就是会话跟踪技术。例如在一个会话中的请求如下：

* 请求银行主页；
* 请求登录（请求参数是用户名和密码）；
* 请求转账（请求参数与转账相关的数据）；
* 请求信誉卡还款（请求参数与还款相关的数据）。

在这上会话中当前用户信息必须在这个会话中共享的，因为登录的是张三，那么在转账和还款时一定是相对张三的转账和还款！这就说明我们必须在**一个会话过程中有共享数据的能力。**

我们知道HTTP协议是无状态协议，也就是说每个请求都是独立的！无法记录前一次请求的状态。但HTTP协议中可以使用Cookie来完成会话跟踪！

在JavaWeb中，使用session来完成会话跟踪，session底层依赖Cookie技术。

会话跟踪技术使用cookie或session完成

1. **什么是cookie？Cookie有什么规范？**

##### ① 什么是cookie

Cookie是由服务器创建，然后通过响应发送给客户端的一个键值对。客户端会保存Cookie，并会标注出Cookie的来源（哪个服务器的Cookie）。当客户端向服务器发出请求时会把所有这个服务器Cookie包含在请求中发送给服务器，这样服务器就可以识别客户端了！

##### ② HTTP的cookie规范如下：

* Cookie大小上限为4KB；
* 一个服务器最多在客户端浏览器上保存20个Cookie；
* 一个浏览器最多保存300个Cookie；

上面的数据只是HTTP的Cookie规范，但在浏览器大战的今天，一些浏览器为了打败对手，为了展现自己的能力起见，可能对Cookie规范“扩展”了一些，例如每个Cookie的大小为8KB，最多可保存500个Cookie等！但也不会出现把你硬盘占满的可能！

注意，不同浏览器之间是不共享Cookie的。也就是说在你使用IE访问服务器时，服务器会把Cookie发给IE，然后由IE保存起来，当你在使用FireFox访问服务器时，不可能把IE保存的Cookie发送给服务器

1. **Cookie和HTTP头以及cookie的覆盖**

Cookie是通过HTTP请求和响应头在客户端和服务器端传递的：

* Cookie：请求头，客户端发送给服务器端；
* 格式：Cookie: a=A; b=B; c=C。即多个Cookie用分号离开；
* Set-Cookie：响应头，服务器端发送给客户端；
* 一个Cookie对象一个Set-Cookie：

//**一个Cookie是一个键一个值。**

Set-Cookie: a=A

Set-Cookie: b=B

Set-Cookie: c=C

**Cookie的覆盖：**

如果服务器端发送重复的Cookie那么会覆盖原有的Cookie，例如客户端的第一个请求服务器端发送的Cookie是：Set-Cookie: a=A；第二请求服务器端发送的是：Set-Cookie: a=AA，那么客户端只留下一个Cookie，即：a=AA。

1. **Cookie的生命**

##### ① 什么是Cookie的生命

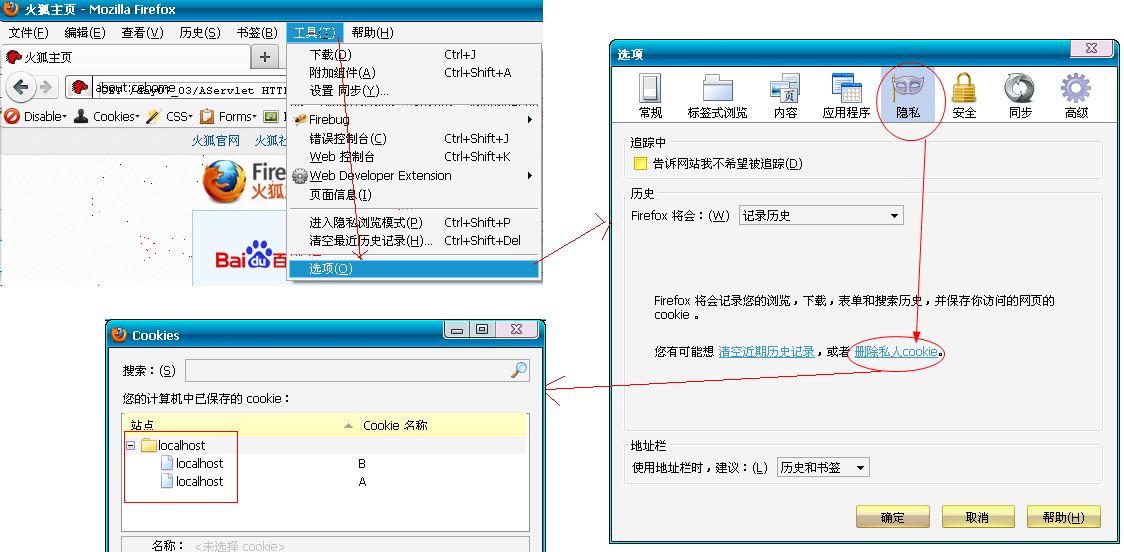
　　Cookie不只是有name和value，Cookie还是生命。所谓生命就是Cookie在客户端的有效时间，可以通过setMaxAge(int)来设置Cookie的有效时间。

* cookie.setMaxAge(-1)：cookie的maxAge属性的默认值就是-1，表示只在**浏览器内存**中存活。一旦关闭浏览器窗口，那么cookie就会消失。
* cookie.setMaxAge(60\*60)：表示cookie对象可存活1小时。当生命大于0时，浏览器会把Cookie保存到硬盘上，就算关闭浏览器，就算重启客户端电脑，cookie也会存活1小时；
* cookie.setMaxAge(0)：cookie生命等于0是一个特殊的值，它表示cookie被作废！也就是说，如果原来浏览器已经保存了这个Cookie，那么可以通过Cookie的setMaxAge(0)来删除这个Cookie。无论是在浏览器内存中，还是在客户端硬盘上都会删除这个Cookie。

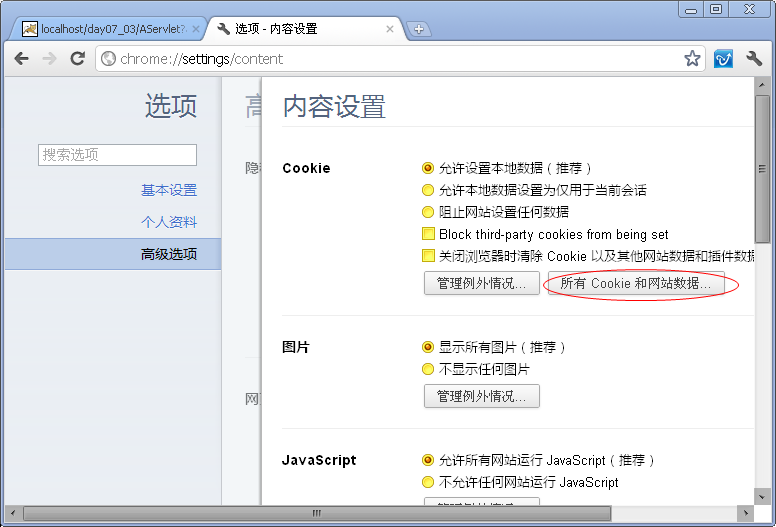
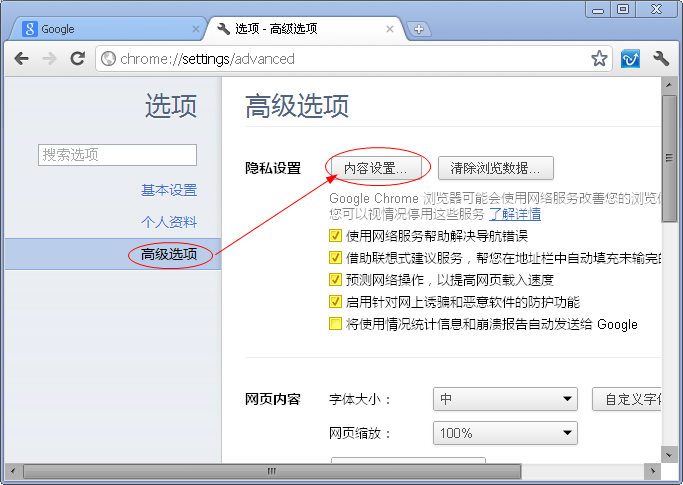
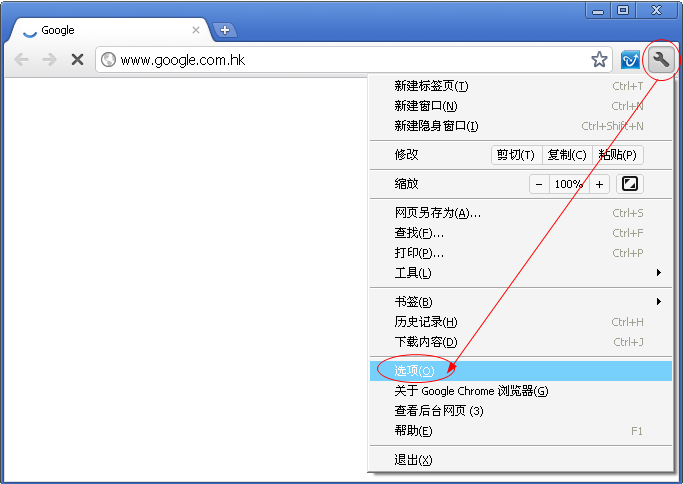
##### ② 浏览器查看Cookie

下面是浏览器查看Cookie的方式：

* IE查看Cookie文件的路径：C:\Documents and Settings\Administrator\Cookies；
* FireFox查看Cooke：



* Google查看Cookie：



##### ③ 案例：显示上次访问时间

* 创建Cookie，名为lasttime，值为当前时间，添加到response中；
* 在AServlet中获取请求中名为lasttime的Cookie；
* 如果不存在输出“您是第一次访问本站”，如果存在输出“您上一次访问本站的时间是xxx”；

AServlet.java

|  |
| --- |
| **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  response.setContentType("text/html;charset=utf-8");  //**创建Cookie对象，名为lasttime，值为当前时间**  Cookie cookie = **new** Cookie("lasttime", **new** Date().toString());  //**设置Cookie在客户端的有效时间为1小时**  cookie.setMaxAge(60 \* 60);  //**添加Cookie到response中**  response.addCookie(cookie);    //**获取请求中的Cookie**  Cookie[] cs = request.getCookies();  String s = "您是首次访问本站！";  **if**(cs != **null**) {  **for**(Cookie c : cs) {  **if**(c.getName().equals("lasttime")) {  s = "您上次的访问时间是：" + c.getValue();  }  }  }    response.getWriter().print(s);  } |

1. **Cookie的path**

每个Cookie都有自己的路径！！！路径与是否归还Cookie相关！！！

Cookie的默认路径（在没有设置路径时），是当前Servlet所在路径！！[http://localhost:8080](http://localhost:8080/day11_3/servlet/BServlet)**[/day11\_3/servlet](http://localhost:8080/day11_3/servlet/BServlet)**[/BServlet](http://localhost:8080/day11_3/servlet/BServlet) 【/day11\_3/servlet为cookie默认路径】

##### ① 什么是Cookie的路径

现在有WEB应用A，向客户端发送了10个Cookie，这就说明客户端无论访问应用A的哪个Servlet都会把这10个Cookie包含在请求中！但是也许只有AServlet需要读取请求中的Cookie，而其他Servlet根本就不会获取请求中的Cookie。这说明客户端浏览器有时发送这些Cookie是多余的！

可以通过设置Cookie的path来指定浏览器，在访问什么样的路径时，包含什么样的Cookie。

##### ② Cookie路径与请求路径的关系

下面我们来看看Cookie路径的作用：

下面是客户端浏览器保存的3个Cookie的路径：

a:　/cookietest；

b:　/cookietest/servlet；

c:　/cookietest/jsp；

下面是浏览器请求的URL：

A:　http://localhost:8080/cookietest/AServlet；

B:　http://localhost:8080/cookietest/servlet/BServlet；

C:　http://localhost:8080/cookietest/jsp/CServlet；

* 请求A时，会在请求中包含a；
* 请求B时，会在请求中包含a、b；
* 请求C时，会在请求中包含a、c；

也就是说，请求路径如果包含了Cookie路径，那么会在请求中包含这个Cookie，否则不会请求中不会包含这个Cookie。

* A请求的URL包含了“/cookietest”，所以会在请求中包含路径为“/cookietest”的Cookie；
* B请求的URL包含了“/cookietest”，以及“/cookietest/servlet”，所以请求中包含路径为“/cookietest”和“/cookietest/servlet”两个Cookie；
* B请求的URL包含了“/cookietest”，以及“/cookietest/jsp”，所以请求中包含路径为“/cookietest”和“/cookietest/jsp”两个Cookie；

##### ③ 设置Cookie的路径

设置Cookie的路径需要使用setPath()方法，例如：

cookie.setPath(“/cookietest/servlet”);

如果没有设置Cookie的路径，那么Cookie路径的默认值当前访问资源所在路径，例如：

* 访问http://localhost:8080/cookietest/AServlet时添加的Cookie默认路径为/cookietest；
* 访问http://localhost:8080/cookietest/servlet/BServlet时添加的Cookie默认路径为/cookietest/servlet；
* 访问http://localhost:8080/cookietest/jsp/BServlet时添加的Cookie默认路径为/cookietest/jsp；

1. **Cookie的应用**

##### ① Cookie中保存中文

Cookie中根本不能保存中文！需要把中文转换成URL编码后才能保存。这也说明读取Cookie之后，还要使用URL解码！！！

Cookie的name和value都不能使用中文，如果希望在Cookie中使用中文，那么需要先对中文进行URL编码，然后把编码后的字符串放到Cookie中。

　向客户端响应中添加Cookie

|  |
| --- |
| //**使用URL编码**  String name = URLEncoder.*encode*("姓名", "UTF-8");  String value = URLEncoder.*encode*("张三", "UTF-8");  //**编码后的字符串保存到cookie中**  Cookie c = **new** Cookie(name, value);  c.setMaxAge(3600);  response.addCookie(c); |

从客户端请求中获取Cookie

|  |
| --- |
| response.setContentType("text/html;charset=utf-8");  Cookie[] cs = request.getCookies();  **if**(cs != **null**) {  **for**(Cookie c : cs) {  //**把cookie的name和value使用URL解码后再打印**  String name = URLDecoder.*decode*(c.getName(), "UTF-8");  String value = URLDecoder.*decode*(c.getValue(), "UTF-8");  String s = name + ": " + value + "<br/>";  response.getWriter().print(s);  }  } |

##### ② 显示曾经浏览过的商品

index.jsp

|  |
| --- |
| <body>  <h1>商品列表</h1>  <a href=*"/day06\_3/GoodServlet?name=ThinkPad"*>ThinkPad</a><br/>  <a href=*"/day06\_3/GoodServlet?name=Lenovo"*>Lenovo</a><br/>  <a href=*"/day06\_3/GoodServlet?name=Apple"*>Apple</a><br/>  <a href=*"/day06\_3/GoodServlet?name=HP"*>HP</a><br/>  <a href=*"/day06\_3/GoodServlet?name=SONY"*>SONY</a><br/>  <a href=*"/day06\_3/GoodServlet?name=ACER"*>ACER</a><br/>  <a href=*"/day06\_3/GoodServlet?name=DELL"*>DELL</a><br/>    <hr/>  您浏览过的商品：  <%  Cookie[] cs = request.getCookies();  **if**(cs != **null**) {  **for**(Cookie c : cs) {  **if**(c.getName().equals("goods")) {  out.print(c.getValue());  }  }  }  %>  </body> |

GoodServlet

|  |
| --- |
| **public** **class** GoodServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  String goodName = request.getParameter("name");  String goods = CookieUtils.*getCookValue*(request, "goods");    **if**(goods != **null**) {  String[] arr = goods.split(", ");  Set<String> goodSet = **new** LinkedHashSet(Arrays.*asList*(arr));  goodSet.add(goodName);  goods = goodSet.toString();  goods = goods.substring(1, goods.length() - 1);  } **else** {  goods = goodName;  }  Cookie cookie = **new** Cookie("goods", goods);  cookie.setMaxAge(1 \* 60 \* 60 \* 24);  response.addCookie(cookie);    response.sendRedirect("/day06\_3/index.jsp");  }  } |

CookieUtils

|  |
| --- |
| **public** **class** CookieUtils {  **public** **static** String getCookValue(HttpServletRequest request, String name) {  Cookie[] cs = request.getCookies();  **if**(cs == **null**) {  **return** **null**;  }  **for**(Cookie c : cs) {  **if**(c.getName().equals(name)) {  **return** c.getValue();  }  }  **return** **null**;  }  } |