

Java Servlets

--- Web Base Course , Module 3

HIGH-END IT TRAINING , SERVLETS

Summary

-  [Introduction to Web](#)
-  [Introduction to HTTP](#)
-  [Introduction to Servlet](#)
-  [Servlets Life Cycle](#)
-  [ServletRequest / ServletResponse](#)
-  [RequestDispatcher](#)
-  [Persistent State in HTTP Servlets](#)
-  [ServletConfig / ServletContext](#)
-  [Filter / Listener](#)

Web 是什么

■ Web 是什么

■ web 单词的中文意思是

- ◆ n. 网, 网络, 网状物
- ◆ vi. 结网
- ◆ vt. 结网于, 使陷入罗网

■ web 在计算机中表示基于网络的服务和程序

- ◆ web 是一个分布式的超媒体(hypermedia)信息系统
 - 它将大量的信息分布于整个因特网上
- ◆ web 的任务就是向人们提供多媒体网络信息服务

■ web 程序

- ◆ 可以通俗地理解成平时所说的网站, 涉及服务器、浏览器、网络等
- ◆ 但 web 程序又不是一般意义上的网站
 - 普通的网站以提供信息服务为主, 重在内容, 程序往往较简单
 - 而商用 web 程序往往比较复杂, 比如背后结合了数据库等技术

Web 相关概念

■ 从技术层面看，Web 技术核心有三点：

- 超文本传输 (HTTP) 协议
 - ◆ 实现万维网的信息传输
- 统一资源定位符 (URL)
 - ◆ 实现互联网信息的定位统一标识
- 超文本标记语言 (HTML)
 - ◆ 实现信息的表示与存储

Web 相关概念

■ 胖客户端程序

- 桌面程序一般也叫做胖客户端程序 (Rich Client Program)
 - ◆ 一般需要安装到计算机才能运行
- RCP 的优缺点
 - ◆ 优点：只要安装了该程序，就可以高效率地使用软件的功能
 - ◆ 缺点：必须安装程序才能使用；占用客户端资源；不易跨平台

■ 瘦客户端程序

- 与 RCP 相对应的是 瘦客户端程序 (Thin Client Program)
 - ◆ 不需要在客户端安装程序，能接入网络，有浏览器就可以使用
 - ◆ TCP 将软件功能的重点集中放在了服务器上，服务器只提供服务
 - ◆ 所谓的 SaaS (Software-as-a-service 软件即服务) 就是 TCP 应用
- TCP 的优缺点
 - ◆ 优点：客户端不必安装程序；便于跨平台实现；极少占用客户端资源
 - ◆ 缺点：必须依赖于服务器

Web 相关概念

■ C/S 结构

- 即 客户端 (Client) / 服务器端 (Server) 模式
- 这种结构中必须安装一个 RCP 程序
- RCP 程序负责与服务器进行数据交换
- 一般常见的网络程序，都是 C/S 结构
 - ◆ 比如 QQ、MSN、大型网络游戏 等等

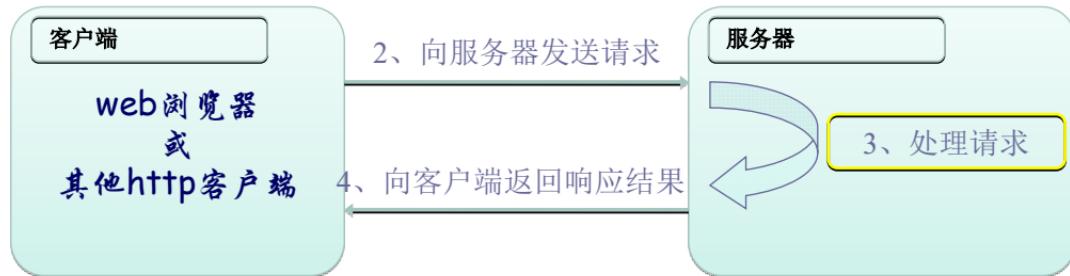
■ B/S 结构

- 指浏览器 (Browser) / 服务器 (Server) 模式
- 这种结构不需要安装 RCP 程序
- 通过任何一种浏览器访问 服务器 即可使用相关服务
- 一般网站都是 B/S 结构
 - ◆ 比如 Google 、 Baidu 、 WebQQ

Web 访问原理

访问 Web 服务器的整个过程

- 使用浏览器访问某个 URL 网址
- 找到网址后向服务器端发送请求
- 服务器接受请求并作出处理，生成处理结果
- 服务器把响应结果返回给客户端
- 浏览器接收到返回的结果，在浏览器中显式返回的结果



Web 开发

■ 什么是 Web 开发

- 一般理解为基于 B/S 架构的所有开发
- 包括服务器端开发和客户端开发

■ Web 开发技术

- CGI (Common Gateway Interface , 公共网关接口)
- ASP (微软公司, 一般与 .net 和 C# 相关联)
- JSP (Sun 公司, 一般与 Java EE 关联)
- PHP (开源, 一般是 LAMP 形式搭配使用)

HTTP 协议及其特点

■ 超文本传输协议 (HTTP)

- HTTP 协议 (HyperText Transfer Protocol, 超文本传输协议)
- 是用于从 www 服务器传输超文本到本地浏览器的传送协议
- Web 服务器和客户端浏览器通过 HTTP 在 Internet 上收发信息

■ HTTP 协议的特点

- HTTP 是一种基于请求/响应模式的协议
 - ◆ 客户端每发送一个请求，服务器就返回一个对应该请求的响应
- HTTP 使用可靠的 TCP 链接，默认端口为 80
 - ◆ 客户端与服务器的会话由客户端建立链接和发送 HTTP 请求的方式初始化
 - ◆ 服务器端不会主动联系客户端或要求与客户端建立链接
 - ◆ 会话开始后，客户端或服务器端都可以随时中断链接

HTTP 请求

HTTP 请求的构成

- 请求方法、URI、协议 / 版本
- 请求头 (Request Header)
- 请求正文

GET /sample.jsp HTTP/1.1

请求方法、URI 协议和版本号

Accept: image/gif, image/jpeg, */*

Accept-Language: zh-cn

Connection: Keep-Alive

Host: localhost

User-Agent: Mozilla/4.0(compatible; MSIE 5.01; Windows NT 5.0)

Accept-Encoding: gzip, deflate

请求头

userNamedcnhanxj&password=9527

这个空行非常重要，用于
分隔 请求头和请求正文

HTTP 响应

HTTP 响应的构成

- 协议、状态代码、描述
- 响应头 (Response Header)
- 响应正文

HTTP / 1.1 200 OK

Server: Apache Tomcat / 6.0.29

Date: Tue, 19 Jul 2011 14:33:03 GMT

Content-Type: text/html

Last-Modified: Tue, 19 Jul 2011 14:03:23 GMT

Content-Length: 112

```
<html><head><title>HTTP响应示例</title></head>
<body>Hello HTTP !</body>
</html>
```

协议和版本 状态代码 描述

响应头

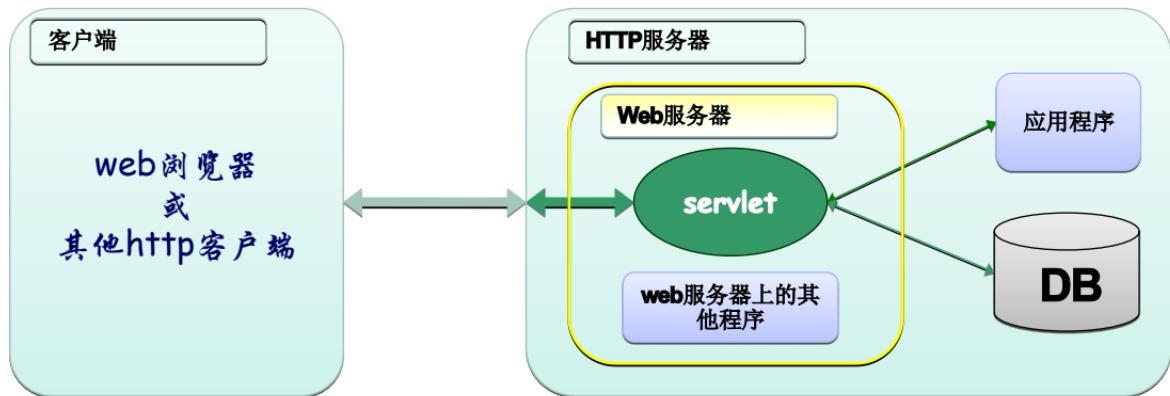
空行，分隔响应头和响应正文

响应正文

什么是Servlet

什么是Servlet

- Servlet是运行在Web服务器或者应用服务器上的Java程序
- 它是一个中间层
 - ◆ 负责连接客户端请求与HTTP服务器上的数据库或者应用程序



Servlet 功用何在

- 读取客户发送的显式数据
- 读取客户发送的隐式数据
- 生成结果
- 向客户发送显式数据
- 发送隐式的 HTTP 请求数据

选择 Web 容器

Web 服务器的选择

- Web 服务器有很多很多，开源的也有很多很多
- 我们选择业界最为流行的，Apache 组织提供的 Tomcat
 - ◆ 目前 tomcat 最新版本为 7.x
 - ◆ 我们可以选择 Tomcat 5.x 或者 Tomcat 6.x
 - ◆ 下表中列出了各个tomcat版本的区别

tomcat版本	最优jdk版本	Servlet规范	JSP规范	JavaEE规范
Tomcat 4.x	1.4	Servlet 2.3	JSP 1.2	
Tomcat 5.x	1.4 / 1.5	Servlet 2.4	JSP 2.0	Java EE 5
Tomcat 6.x	1.5 / 1.6	Servlet 2.5	JSP 2.1	Java EE 5 / 6
Tomcat 7.0	1.5 / 1.6	Servlet 3.0	JSP 2.2	Java EE 6

Servlet 开发环境

■ 获取 Tomcat

- 登陆 www.apache.org
- 在页面下部的 Apache Projects 里找到 tomcat
- 点击后进入 tomcat 页面(图标是一只黄色的猫)
- 在左边的 Downloads 点击 Tomcat 6.0
- 页面中下部找到 6.0.29 部分(最新为 6.0.29)
- 根据你的处理器配置和操作系统, 选择相应的版本下载

The screenshot shows the Apache Tomcat 6.0.29 download page. A callout bubble points to the 'Binary Distributions' section, which lists various download links for different processor architectures (Core and Deployer) and operating systems (Windows, Linux, Solaris, etc.).

6.0.29

Please see the [README](#) file for packaging information. It explains what

Binary Distributions

- Core:
 - [zip \(pgp, md5\)](#)
 - [tar.gz \(pgp, md5\)](#)
 - [32-bit Windows zip \(pgp, md5\)](#)
 - [64-bit Windows zip \(pgp, md5\)](#)
 - [64-bit Itanium Windows zip \(pgp, md5\)](#)
 - [32-bit/64-bit Windows Service Installer \(pgp, md5\)](#)
- Deployer:
 - [zip \(pgp, md5\)](#)
 - [tar.gz \(pgp, md5\)](#)

在这里找
你的处理
器和操作
系统支持
的版本

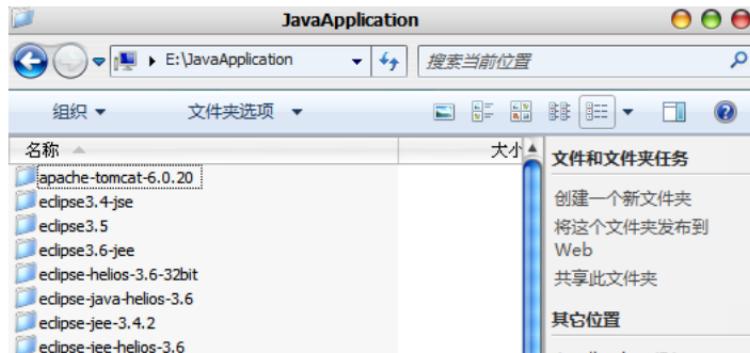
Servlet 开发环境

解压下载的压缩包

- 从官网下载的压缩包为 apache-tomcat-6.0.29.zip
- 把该压缩包解压到某一目录中

良好的编程习惯

- 我们强烈建议在磁盘上建立一个目录，比如 JavaApplication
- 该目录放置我们所用到的所有的 Java 程序
- 同时建议把 tomcat 解压到这个目录中去

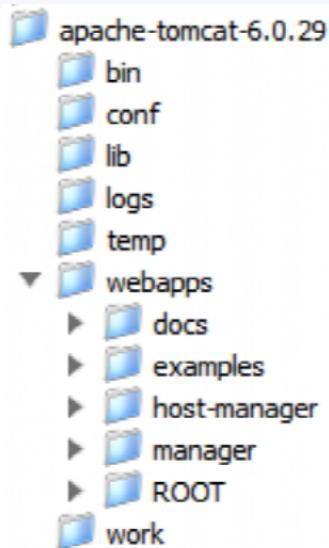


左图中，在 E 盘建立了一个文件夹 E:\JavaApplication
该文件夹下存放的全部是 Java 方面的程序
这样便于管理和查找

Servlet 开发环境

Tomcat 的目录结构

- bin 目录：存放操纵 tomcat 的命令
- conf 目录：存放配置文件
- lib 目录：存放一些 jar 文件
 - ◆ 这些 jar 文件往往是某种规范的具体实现
- logs 目录：用于存放 tomcat 的日志信息
- webapps 目录：存放 web 应用
 - ◆ 后面会看到，我们的应用程序就放在这里
- work 目录
 - ◆ 这个目录存放编译好的 class 文件等信息
 - ◆ 在后面或者 Servlets 高级部分做详细讲解
- temp 目录
 - ◆ 后面视情况讲解



Servlet 开发环境

运行tomcat依赖的环境变量

- 你必须已安装 JDK 而不仅是 JRE
- Tomcat 必须依赖 JAVA_HOME 环境变量
 - ◆ Windows 系统:
 - `JAVA_HOME=D:\Program Files\Java\jdk1.6.0_18`
 - ◆ Linux 系统:
 - `JAVA_HOME=/opt/JavaApplication/jdk1.6.0_18`
- 一般还可以指定 tomcat 的安装目录(可选)
 - ◆ Windows 系统:
 - `CATALINA_HOME=E:\JavaApplication\apache-tomcat-6.0.29`
 - ◆ Linux 系统:
 - `CATALINA_HOME=/opt/JavaApplication/apache-tomcat-6.0.29`

Servlet 开发环境

启动 tomcat

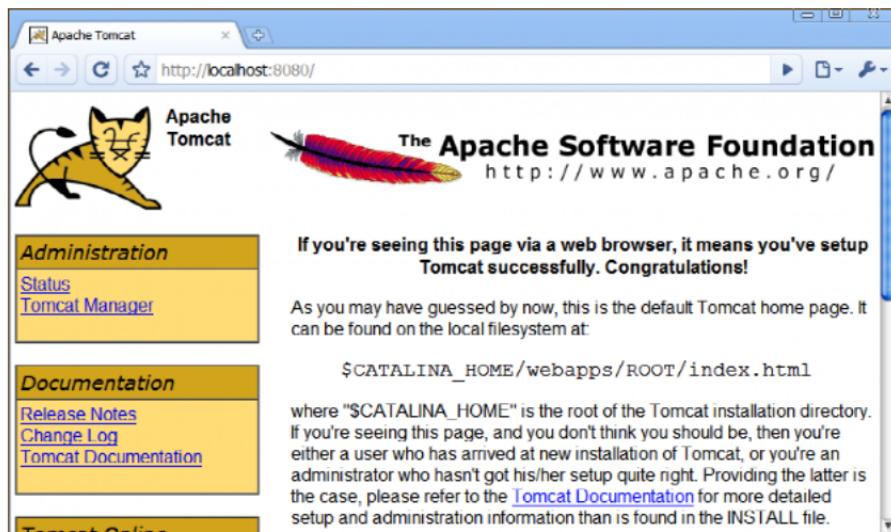
- 进入命令提示符(Linux下是打开一个终端)
 - ◆ 进入 apache-tomcat-6.0.29/bin 目录
 - 使用 catalina.bat 或 catalina.sh 启动
 - » Windows 输入 catalina.bat run , 回车, 启动 tomcat
 - » Linux 输入 catalina.sh run , 回车, 启动 tomcat
 - 使用 startup.bat 或 startup.sh 启动
 - » Windows 输入 startup.bat , 回车, 启动 tomcat
 - » Linux 输入 startup.sh , 回车, 启动 tomcat
 - ◆ 两种启动方法的区别
 - catalina 方式便于程序调试, 只打开一个命令窗口
 - startup 方式则会在当前窗口基础上, 再打开一个窗口

Servlet 开发环境

访问 tomcat 主页

- 启动后可以通过浏览器访问

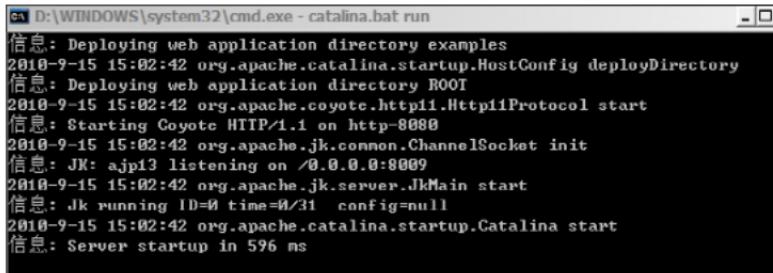
- ◆ <http://localhost:8080/>
- ◆ 其中 localhost 指的是本机，8080 是端口号



Servlet 开发环境

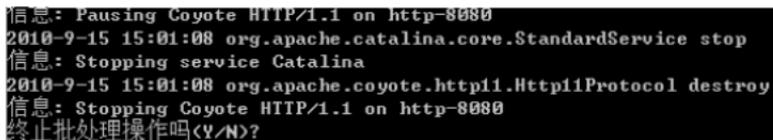
■ 关闭 tomcat

- 在已经启动 tomcat 的命令行(终端)窗口中
 - ◆ 按 Ctrl + C 或者 Ctrl + Z 或者 Ctrl + D [视操作系统而言]
 - ◆ Windows 下使用 Ctrl + C



```
D:\WINDOWS\system32\cmd.exe - catalina.bat run
信息: Deploying web application directory examples
2010-9-15 15:02:42 org.apache.catalina.startup.HostConfig deployDirectory
信息: Deploying web application directory ROOT
2010-9-15 15:02:42 org.apache.coyote.http11.Http11Protocol start
信息: Starting Coyote HTTP/1.1 on http-8080
2010-9-15 15:02:42 org.apache.jk.common.ChannelSocket init
信息: JK: ajp13 listening on /0.0.0.0:8009
2010-9-15 15:02:42 org.apache.jk.server.JkMain start
信息: Jk running ID=0 time=0/31 config=null
2010-9-15 15:02:42 org.apache.catalina.startup.Catalina start
信息: Server startup in 596 ms
```

- 程序会进入下面的提示，这时输入 y，回车即可



```
信息: Pausing Coyote HTTP/1.1 on http-8080
2010-9-15 15:01:08 org.apache.catalina.core.StandardService stop
信息: Stopping service Catalina
2010-9-15 15:01:08 org.apache.coyote.http11.Http11Protocol destroy
信息: Stopping Coyote HTTP/1.1 on http-8080
终止批处理操作吗(Y/N)?
```

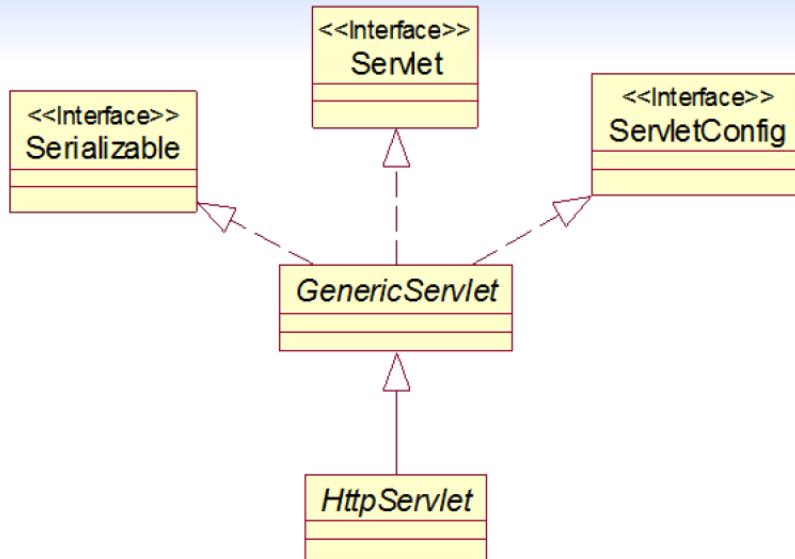
Contents

- Servlet 网络拓扑结构
- Servlet 体系结构
- Tomcat 中 web 应用的目录结构
- 实现 Servlet
 - 继承 javax.servlet.http.HttpServlet 类
 - 继承 javax.servlet.http.GenericServlet 类
 - 实现 javax.servlet.Servlet 接口
 - 开发 Servlet 的流程
- Servlet 的生命周期
 - Servlet 的执行过程
 - Servlet 从被加载至destory的时序图
 - Servlet 从被加载至destory的活动图

Servlet 网络拓扑结构



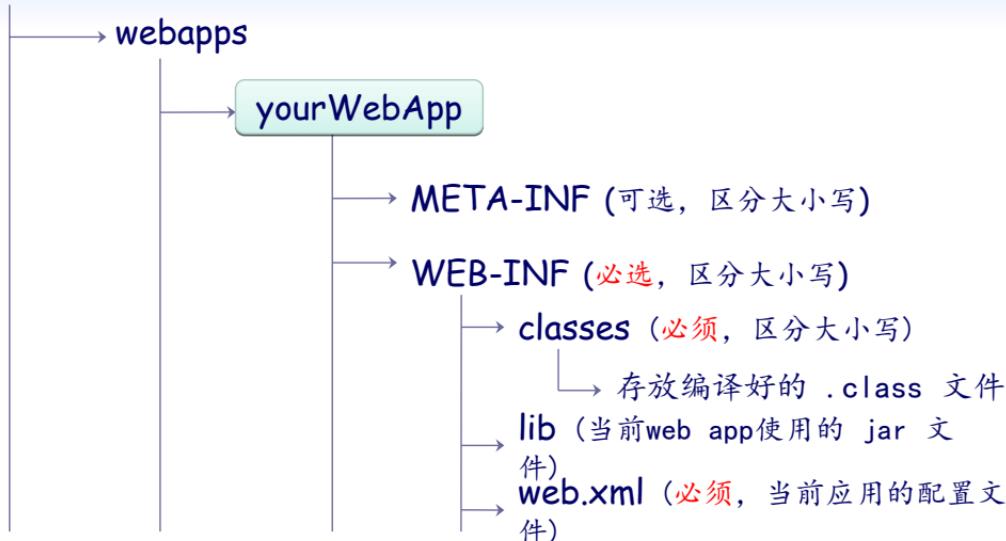
Servlet 体系结构



web 应用的目录结构

Tomcat 中 web 应用的目录结构

Tomcat 安装目录



以上目录层次要求能够默写出来

配置开发 Servlet 的编译环境

配置 Servlet 的编译环境

配置 CLASSPATH 环境变量

- ◆ Windows 系统
 - `JAVA_HOME=D:\Program Files\Java\jdk1.6.0_18`
 - `CATALINA_HOME=E:\JavaApplication\apache-tomcat-6.0.29`
 - `PATH=%JAVA_HOME%\bin;.....` (.....为原有内容，千万不要乱改！)
 - `CLASSPATH=.;%JAVA_HOME%\lib;%CATALINA_HOME%\lib\servlet-api.jar`
- ◆ Linux 系统
 - `JAVA_HOME=/opt/JavaApplication/jdk1.6.0_18`
 - `CATALINA_HOME=/opt/JavaApplication/apache-tomcat-6.0.29`
 - `PATH=%JAVA_HOME%/bin:.....` (.....为原有内容，千万不要乱改！)
 - `CLASSPATH=.;%JAVA_HOME%/lib;%CATALINA_HOME%\lib\servlet-api.jar`

为什么要配置编译环境，最重要的是哪个？？？

建立前述目录结构

建立web应用的目录层次

- 进入 apache-tomcat-6.0.29\webapps 目录
- 在其中新建一个目录 FirstServlet
- 从 ROOT 目录中复制 WEB-INF 目录到 FirstServlet 目录下
- 打开 WEB-INF\web.xml 文件
 - ◆ 删除其中内容，直至剩下以下内容
 - ◆ 建议把 ISO-8859-1 改成 UTF-8，即可以在该文件中使用中文

```
<?xml version="1.0" encoding="ISO-8859-1"?>

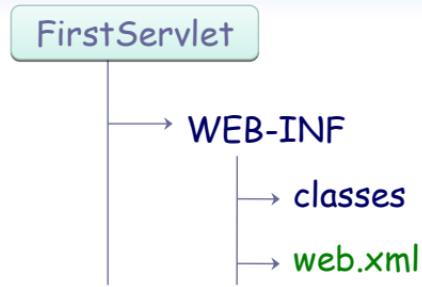
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    version="2.5">

</web-app>
```

实现Servlet - 继承 HttpServlet 类

第一个Servlet

- 进入 apache-tomcat-6.0.29\webapps 目录
- 进入 FirstServlet\WEB-INF 目录
- 新建一个 classes 的目录 (注意大小写)
- 至此, 我们的目录结构如右图
 - ◆ 右图中 FirstServlet 指 web 应用的名字
 - ◆ 绿色表示一个文件, 蓝色代表一个目录



实现Servlet

- 在 classes 目录中, 新建 HttpServletDemo.java
- 该类继承自 javax.servlet.http.HttpServlet 类
- 重写父类中的 doGet() 方法 和 doPost() 方法
- 代码详细见下页

实现Servlet - 继承 HttpServlet 类

■ 继承 HttpServlet 类实现 Servlet

```
public class HttpServletDemo extends HttpServlet {  
    protected void doGet( HttpServletRequest req ,  
                         HttpServletResponse resp )  
        throws ServletException, IOException {  
        this.doPost(req, resp);  
    }  
    protected void doPost( HttpServletRequest req ,  
                          HttpServletResponse resp )  
        throws ServletException, IOException {  
        resp.setContentType("text/html;charset=UTF-8"); //设置响应报头  
        PrintWriter out = resp.getWriter(); //获取输出流  
        out.print("<html><body><center>");  
        out.print("<h1>Hello , HttpServlet </h1>");  
        out.print("</center></body></html>");  
        out.flush();           out.close();  
    }  
} //限于篇幅, 省略了 import 语句, 本类中不使用 package 语句
```

实现Servlet - 继承 HttpServlet 类

■ 编译 HttpServletDemo.java

■ 就地编译

- ◆ 打开命令提示符 或 打开终端
- ◆ 进入 webapps\FirstServlet\WEB-INF\classes
- ◆ 使用 dir 命令(Windows) 或者 ls 命令(Linux) 查看文件
 - 可以查看到 HttpServletDemo.java
- ◆ 编译 HttpServletDemo.java
 - 因为没有使用 package 语句, 所以可以直接就地编译
 - javac HttpServletDemo.java
 - » 如果提示 “TestHttpServlet.java:5: 软件包 javax.servlet 不存在” 之类的信息
 - » 是因为环境变量中 servlet-api.jar 没有配置到 CLASSPATH 变量中
- ◆ 编译完后, 再用 dir 或者 ls 查看
 - 会多出一个 HttpServletDemo.class 文件
- ◆ 至此, 通过继承 HttpServlet 类实现 Servlet 完毕

实现Servlet - 继承 HttpServlet 类

修改 web.xml 文件

- 打开 FirstServlet\WEB-INF\web.xml 文件

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="这个部分的内容原样不动">
    <servlet><!-- 添加以下绿色的部分，其他部分一律不动-->
        <!-- servlet-name 可以任意指定 -->
        <servlet-name>Servlet001</servlet-name>          二者必须写一致
        <!-- servlet-class 指定Servlet全类名，严格区分大小写 -->
        <servlet-class>HttpServletDemo</servlet-class>
    </servlet>
    <servlet-mapping>
        <!-- 这里的 servlet-name 必须跟上面写一致 -->
        <servlet-name>Servlet001</servlet-name>
        <!-- url-pattern 指定的是前台访问时使用的地址
             http://localhost:8080/FirstServlet/httpServletDemo -->
        <url-pattern>/httpServletDemo</url-pattern>
    </servlet-mapping>
</web-app>

```

→ 这是刚写的那个servlet的类名

→ 这是在浏览器访问时的地址

实现Servlet - 继承 HttpServlet 类

访问我们开发的Servlet

- 打开命令提示符
- cd 进入 apache-tomcat-6.0.29\bin 目录
- 使用 catalina.bat run 或者 catalina.sh run 启动 tomcat
- 打开浏览器，访问刚开发的 Servlet
 - ◆ <http://localhost:8080/FirstServlet/httpServletDemo>
 - ◆ <http://127.0.0.1:8080/FirstServlet/httpServletDemo>



实现Servlet - 继承 HttpServlet 类

第一个 Servlet 开发总结

- 必须准备好相关的环境变量
- 在 WEB-INF 中新建 classes 目录
- 在 classes 目录中新建 HttpServletDemo.java
 - ◆ 继承自 javax.servlet.http.HttpServlet 类
 - ◆ 实现 doGet() 和 doPost() 方法
 - ◆ 就地编译 (.class 文件放在 classes 下)
- 修改 web.xml 文件
- 在命令提示符 或 终端中使用命令启动 tomcat
 - ◆ Windows 使用 catalina.bat run
 - ◆ Linux 使用 catalina.sh run
- 在浏览器中访问开发的Servlet
 - ◆ <http://localhost:8080/FirstServlet/httpServletDemo>
 - ◆ <http://127.0.0.1:8080/FirstServlet/httpServletDemo>

实现Servlet - 继承 HttpServlet 类

第一个 Servlet 开发总结

- Servlet 程序与可以在 Eclipse 中运行的 Java 程序的区别
 - ◆ 可以在 Eclipse 中或终端中运行的 Java 程序
 - 必须要有 main 方法，作为程序的入口
 - 运行该类程序，会启动一个与之对应的 JVM 线程
 - ◆ Servlet 程序
 - 不需要有 main 方法，不能在 eclipse 或 终端 中运行
 - 必须在 Servlet 容器中才能被调用
 - Servlet 容器为 Servlet 程序提供了 JavaEE 规范中述及的接口和类
- Servlet 程序与可以在 Eclipse 中运行的 Java 程序的联系
 - ◆ 一般是在 Servlet 中调用可以运行的 Java 类的非 main 方法

实现Servlet - 继承 GenericServlet 类

■ 继承 javax.servlet.GenericServlet 类实现 Servlet

- 在 classes 下新建一个类 GenericServletDemo
- 加上 package 语句，该类依然输出一句话

```
package cn.royallin.servlet;

public class GenericServletDemo extends GenericServlet {
    public void service( ServletRequest request ,
                        ServletResponse response )
        throws ServletException, IOException {
        resp.setContentType("text/html"); //设置响应报头
        PrintWriter out = resp.getWriter(); //获取输出流
        out.print("<html><body><center>");
        out.print("<h1>Hello , GenericServlet </h1>");
        out.print("</center></body></html>");
        out.flush();                 out.close();
    }
} //限于篇幅，省略了 import 语句
```

实现Servlet - 继承 GenericServlet 类

■ 继承 javax.servlet.GenericServlet 类实现 Servlet

- 编译刚刚开发的 Servlet 程序
 - ◆ javac -d . GenericServlet.java
 - ◆ 生成的 .class 将存放在 cn\royallin\servlet 目录下
 - 即: cn\royallin\servlet\GenericServlet.class
- 在 web.xml 中追加以下部分的内容

```
<!-- 前面是原有的内容，一律不动 -->
<!-- GreenericServletDemo -->
<servlet>
    <servlet-name>cn.royallin.servlet.GenericServletDemo</servlet-name>
    <servlet-class>cn.royallin.servlet.GenericServletDemo</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>cn.royallin.servlet.GenericServletDemo</servlet-name>
    <url-pattern>/genericServletDemo</url-pattern>
</servlet-mapping>
</web-app> <!-- 最后面的 web-app 不动 -->
```

实现Servlet - 继承 GenericServlet 类

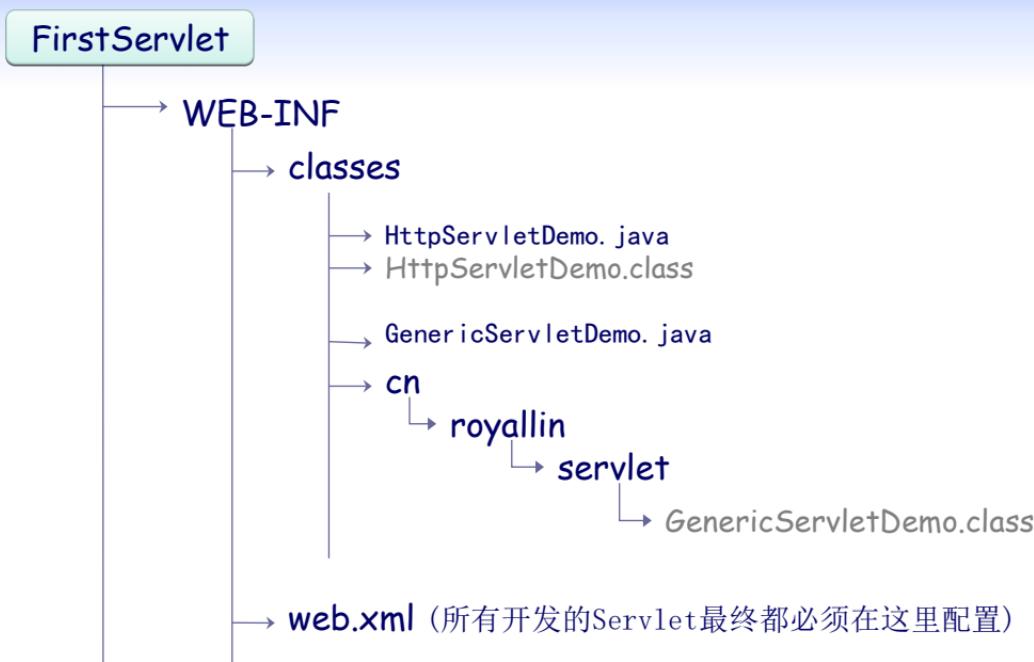
访问刚刚开发的Servlet

- 启动 tomcat
- 使用浏览器访问， Chrome 浏览效果如下



实现Servlet - 继承 GenericServlet 类

- 回顾我们的 FirstServlet 应用的目录层次



实现Servlet - 实现 Servlet 接口

通过实现 javax.servlet.Servlet 接口实现 Servlet

- 同上面两个 Servlet 程序一样，在 classes 中新建一个类
 - ◆ 该类类名为 ServletInterfaceDemo
 - ◆ 实现 javax.servlet.Servlet 接口
 - ◆ 把 javax.servlet.Servlet 接口里的方法全部实现，包括
 - 三个核心方法：
 - » destroy()
 - » init(ServletConfig config)
 - » service(ServletRequest request, ServletResponse response)
 - 两个辅助方法
 - » getServletConfig()
 - » getServletInfo()
 - ◆ 同 GenericServletDemo.java 一样，重点实现 service 方法
 - ◆ 后面两页是详细代码！

实现Servlet - 实现 Servlet 接口

通过实现 javax.servlet.Servlet 接口实现 Servlet

```
package cn.royallin.servlet;
//此处省略了 import 语句
public class ServletInterfaceDemo implements Servlet {

    public void destroy() {
        System.out.println(" ServletInterfaceDemo destroy() 执行 ");
    }

    public ServletConfig getServletConfig() {
        return null;
    }

    public String getServletInfo() {
        return null;
    }
}
```

实现Servlet - 实现 Servlet 接口

通过实现 javax.servlet.Servlet 接口实现 Servlet

```
public void init(ServletConfig config) throws ServletException {  
    System.out.println(" ServletInterfaceDemo init() 执行 ");  
}  
  
public void service (ServletRequest request, ServletResponse  
response)  
        throws ServletException, IOException {  
    System.out.println(" ServletInterfaceDemo service() 执行 ");  
    response.setContentType("text/html");//设置响应报头  
    PrintWriter out = response.getWriter(); //获取输出流  
    out.print("<html><body><center>");  
    out.print("<h1>Hello , Servlet </h1>");  
    out.print("</center></body></html>");  
    out.flush();      out.close();  
}  
}// ServletInterfaceDemo 类全部结束
```

实现Servlet - 实现 Servlet 接口

- 通过实现 javax.servlet.Servlet 接口实现 Servlet
- 在 web.xml 中配置 ServletInterfaceDemo

```
<!-- 前面是原有的内容，一律不动 -->
<!-- ServletInterfaceDemo -->
<servlet>
    <servlet-name>cn.royallin.servlet.ServletInterfaceDemo</servlet-name>
    <servlet-class>cn.royallin.servlet.GenericServletDemo</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>cn.royallin.servlet.ServletInterfaceDemo</servlet-name>
    <url-pattern>/servletInterfaceDemo</url-pattern>
</servlet-mapping>
</web-app> <!-- 最后面还是以 /web-app 结束 -->
```

实现Servlet - 实现 Servlet 接口

通过实现 javax.servlet.Servlet 接口实现 Servlet

- 启动或重启动 tomcat
- 在浏览器中访问，Chrome 浏览器访问结果如下



三种方式实现 Servlet 总结

总结前面的三种 Servlet 的开发方式

- 由浅入深的学习，逐步掌握 Servlet 开发，逐步接近原理
 - ◆ 第一个Servlet的开发
 - 继承 javax.servlet.http.HttpServlet 类
 - 这是目前开发 Servlet 最常用的一种方法，也是比较简单的一种方法
 - ◆ 第二个Servlet的开发
 - 继承了 HttpServlet 的父类 javax.servlet.GenericServlet
 - 重写了 service() 方法
 - ◆ 第三个Servlet的开发
 - 实现 javax.servlet.Servlet 接口
 - 实现所有抽象的方法
 - » init() 方法 和 destory() 方法中加上打印语句
 - 着重实现 service() 方法，业务逻辑在此完成
 - 这是最原始的一种开发 Servlet 的
- 必须熟悉 tomcat 中 webapp 应用的目录层次

Annotation Supported

■ Servlet 3.0 开始允许使用注解来标注 Servlet

■ @WebServlet

- ◆ 在一个实现过 Servlet 接口的类上标注，声明这是一个Servlet
- ◆ @WebServlet 的常用属性
 - String name 指定当前Servlet 的名称，相当于 xml 中的 servlet-name
 - String[] value 指定当前 Servlet 对应的 url (与 url-pattern 对应)
 - String[] urlPatterns 与 value 作用相同
 - int loadOnStartup 作用与 load-on-startup ，设定 Servlet 的加载顺序
 - boolean asyncSupported 指定是否支持异步操作
 - WebInitParam[] initParams 用于设置 Servlet 初始化参数

■ @WebInitParam

- ◆ 必须与 @WebServlet 、@Filter 等联合使用
 - 用于配置 Servlet 或 Filter 的初始化参数，等同于 init-param
- ◆ @WebInitParam 的属性
 - String name : 用于设置初始化参数的名称
 - String value : 用于设置初始化参数的值

Servlets Life Cycle

Servlet 的生命周期

加载和实例化

- ◆ 容器通过反射加载相应的 Servlet 类并实例化该类的一个实例
 - 对 Servlet 类的加载可能发生在引擎启动时
 - 也可能是用户第一次访问相应的 Servlet 时
- ◆ 容器只为相应的 Servlet 初始化一个实例

初始化：调用 init() 方法

- ◆ Servlet 第一次被调用时，调用 init() 方法一次
- ◆ 以后除非被 reload 或者 destory 后重新访问，才会重新调用该方法
 - 之所以重新调用
 - » 是因为原有的 Servlet 生命周期已结束
 - » 一个新的 Servlet 生命周期开始

处理请求：调用 service() 方法

- 业务逻辑一般在该方法中完成，该方法可以被多次调用
- 只要有一个用户访问当前 Servlet，就会调用该方法（一个连接一个线程）

Servlets Life Cycle

Servlet 的生命周期

服务结束：调用 `destroy()` 方法

- ◆ 有三种情况会调用 `destroy` 方法(只调用一次)
 - tomcat 将重启
 - reload 该 webapp
 - 关闭或重启计算机(Servlet 容器也随之关闭)

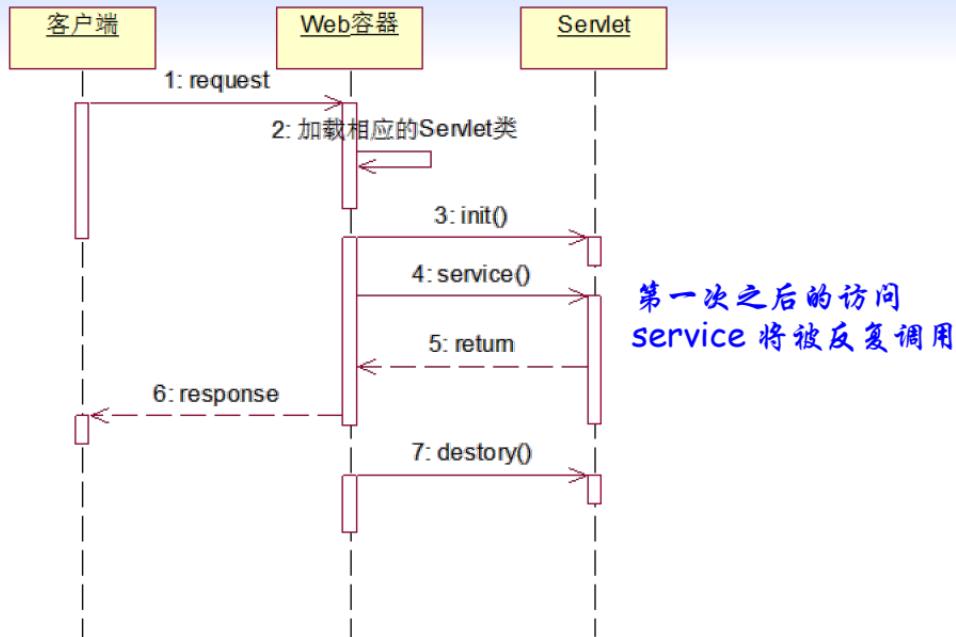
使用 Servlet 验证执行过程

用 `ServletInterfaceDemo` 中的打印顺序验证 Servlet 执行过程

- ◆ `init()` 可能在 用户第一次请求时在控制台打印
- ◆ `service()` 在每个用户请求时打印(每个用户一个线程)
- ◆ 在控制台(命令提示符或终端)中关闭 tomcat , 将调用 `destroy()`

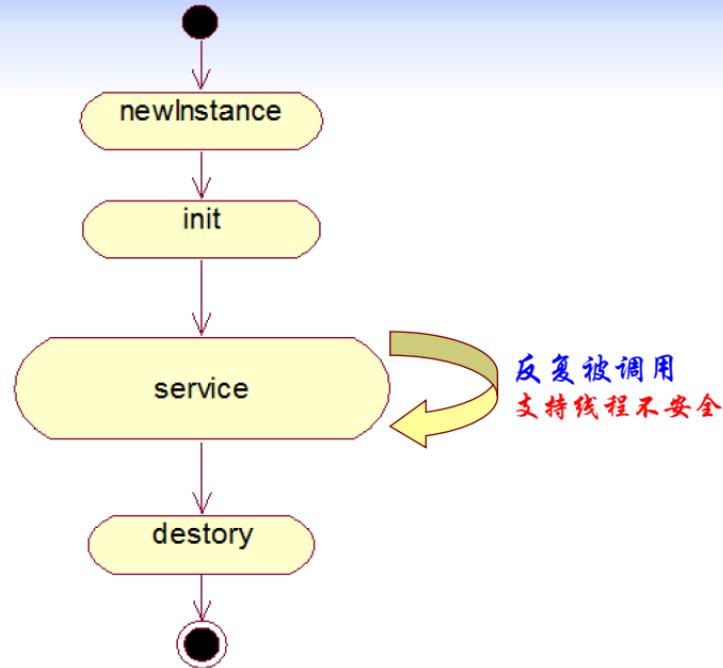
Servlets Life Cycle

Servlet第一次被访问至销毁的时序图



Servlets Life Cycle

活动图



Contents

■ 基于 HTTP 的请求响应机制

- HTTP 请求的类型
- 表单的提交方式
- Servlet 容器响应 web 客户端请求的过程

■ ServletRequest

- 获取表单数据
- 获取请求报头数据
- 获取和设置请求编码
- 获取客户端上传的文件

■ ServletResponse

- 服务器响应代码
- 服务器响应报头

HTTP 请求方法

■ HTTP/1.1 协议中共定义的八种方法

- 这些方法用来表明 Request-URI 指定的资源的不同操作方式
- 这些方法分别为
 - ◆ OPTIONS
 - 返回服务器针对特定资源所支持的 HTTP 请求方法
 - 也可以利用向 Web 服务器发送“*”的请求来测试服务器的功能性
 - ◆ HEAD
 - 向服务器索要与 GET 请求相一致的响应，只不过响应体将不会被返回
 - 此方法可以在不必传输整个响应内容的情况下，获取包含在响应头中的元信息
 - ◆ GET
 - 向特定的资源发出请求
 - ◆ POST
 - 向指定资源提交数据进行处理请求（例如提交表单或者上传文件）
 - 数据被包含在请求体中
 - POST 请求可能会导致新的资源的建立和/或已有资源的修改。

HTTP 请求方法

■ HTTP/1.1 协议中共定义的八种方法

■ 续前页

- ◆ PUT
 - 向指定资源位置上传其最新内容
- ◆ DELETE
 - 删除指定资源
- ◆ TRACE
 - 回显服务器收到的请求
- ◆ CONNECT
 - HTTP/1.1 协议中预留给能够将连接改为管道方式的代理服务器

表单的提交方式

表单的提交方式一般有两种

■ Get

- ◆ 使用MIME类型 application/x-www-form-urlencoded 的 urlencoded 文本的格式传递参数
- ◆ urlencoding是一种字符编码，保证被传送的参数由遵循规范的文本组成
 - 例如一个空格的编码是"%20"。附加参数还能被认为是一个查询字符串

■ Post

- ◆ 与 GET 类似， POST 参数也是被 URL编码的
- ◆ 然而，变量名/变量值不作为 URL 的一部分被传送，而是放在实际的 HTTP 请求消息内部被传送

表单的提交方式

Get 和 Post 的区别与联系

从安全角度，post 请求比 get 请求安全

- ◆ get 请求会把提交的属性显示在地址栏上
 - 请求地址之后会以 ? 开始，以后多个参数以 & 连接，如
» `http://localhost:8080/mystruts/index.do?action=HelloAction&jsp=index`
 - ◆ post 则不会显示，从该角度看，post 较为安全

从提交内容上看

- ◆ get 可提交内容少，所提交数据最大一般不超过 2 Kb
- ◆ post 可提交内容多，所提交数据理论上没有限制
 - 实际应用中，一般建议不超过 64 Kb

从速度上讲

- ◆ get 较快，它要求服务器立即处理请求
- ◆ post 较慢，可能形成一个队列请求

■ 多任务因特网邮件扩充

- 即 Multipurpose Internet Mail Extension , 简称 MIME
- 用来设定文档内容
- 一些常见的内容类型如下：
 - ◆ text/html: HTML文档
 - ◆ text/plain: 纯文本文件
 - ◆ image/jpeg: jpeg图像文件
 - ◆ image/GIF: gif图像文件
 - ◆ image/TIFF: TIFF格式的图形文件
 - ◆ application/rtf: 多信息文本格式文档
 - ◆ application/zip: PKZIP或WinZIP压缩文件
 - ◆ video/mpeg: mpeg视频文件
 - ◆ video/quicktime: QuickTime视频文件

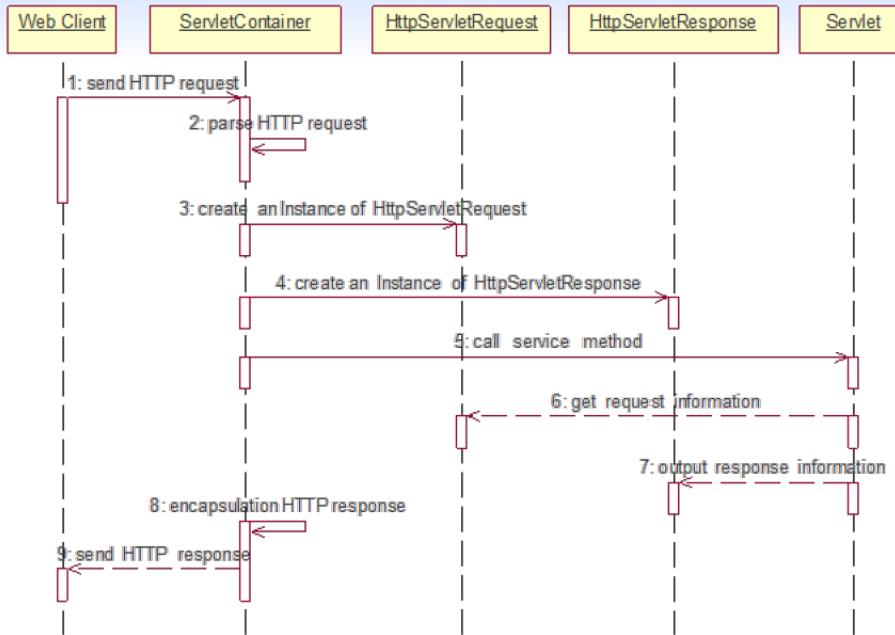
ServletRequest & ServletResponse

■ 基于 HTTP 协议的请求响应机制的信息交换过程

- 建立连接：
 - ◆ 客户端与服务器建立 TCP 连接
- 发送请求：
 - ◆ 建立连接后，客户端把请求消息发送到服务器的相应端口上
- 处理请求
 - ◆ Servlet 容器接受到 HTTP 请求
 - ◆ 解析 HTTP 请求并封装相应的对象
- 发送响应：
 - ◆ 服务器在处理完客户端请求之后，要向客户端发送响应消息
- 关闭连接：
 - ◆ 客户端和服务器双方都可以通过关闭 Socket 来结束 TCP/IP 对话

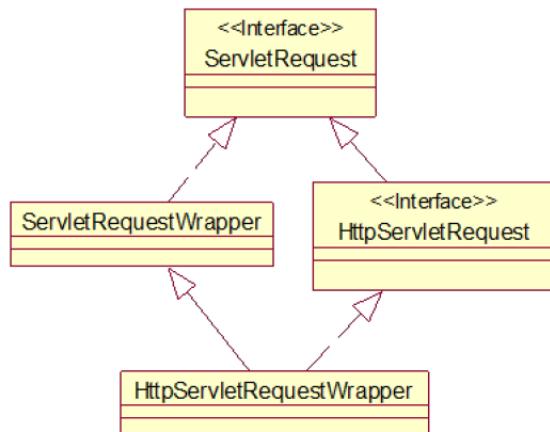
ServletRequest & ServletResponse

Servlet 容器响应 Web 客户端请求的时序图



ServletRequest

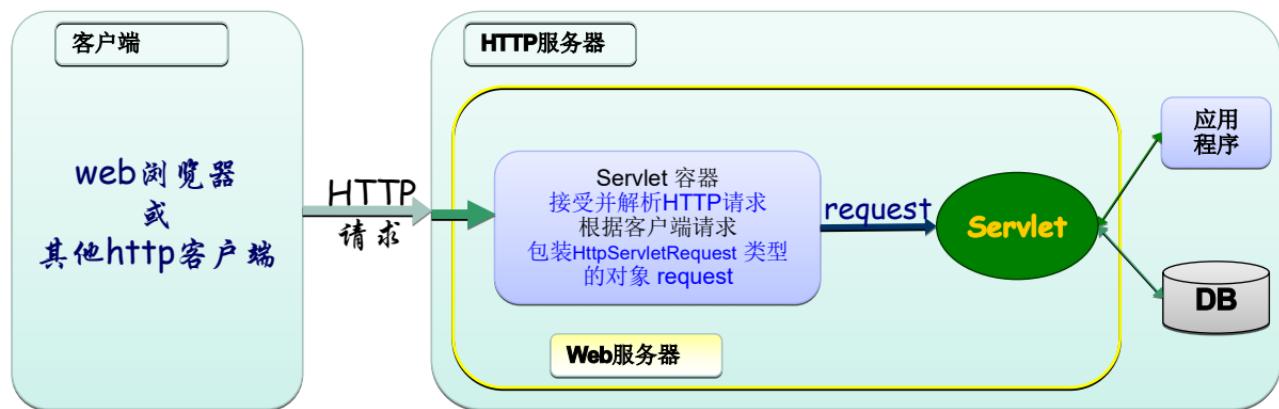
Servlet API 中关于请求的层次结构



ServletRequest

ServletRequest 对象是怎么来的

- 客户端发送 HTTP 请求给服务器
- 服务器接受到 HTTP 请求后解析这个请求
- 然后根据 Java EE 规范封装 HttpServletRequest 对象



ServletRequest

■ 获取表单内容的方法

- `String getParameter(String parameterName)`
 - ◆ 返回表单中参数名(区分大小写)对应的值
 - 如果parameterName指定的参数不存在, 返回 null
 - 如果parameterName指定的参数没有任何值, 返回空字符串
 - 如果parameterName指定的参数有多个, 那么只取其中一个
- `Enumeration getParameterNames()`
 - ◆ 返回一个存放String对象的枚举(Enumeration)
 - ◆ 通过该枚举中存放当前请求发来的from表单中所有的参数
 - 关于 from 表单请参见 HTML 部分的 PPT, 此处不再赘述
 - ◆ 接口 Enumeration 中提供了如下方法
 - `boolean hasMoreElements()` : 判断是否还有更多元素
 - `E nextElement()` : 获取下一个元素

ServletRequest

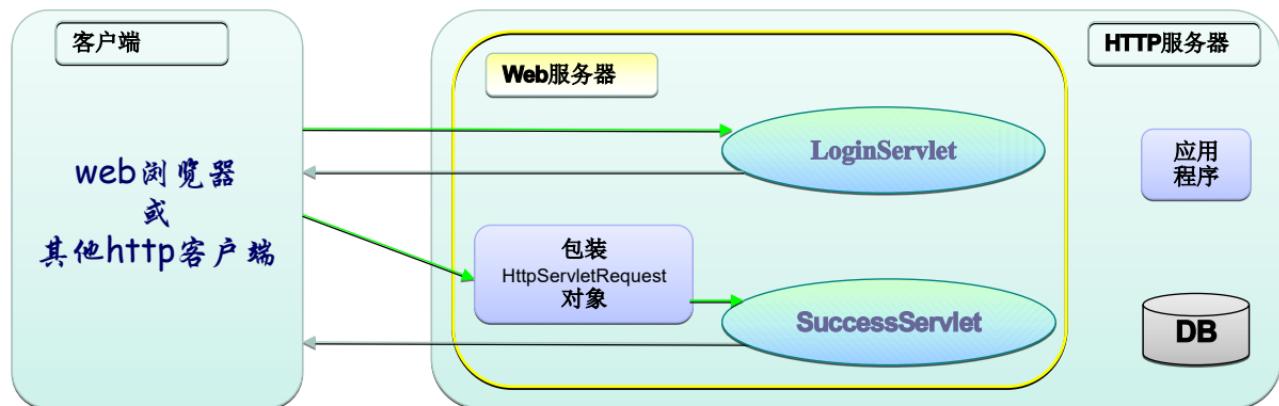
■ 获取表单内容的方法

- `String[] getParameterValues(String parameterName)`
 - ◆ 返回表单中参数名(区分大小写)对应的字符串数组
 - `parameterName` 指定的参数不存在, 返回null;
 - `parameterName` 指定的参数只有一个值, 返回仅有一个元素的数组
 - `parameterName` 指定的参数有多个, 返回有多个元素的数组
- `Map getParameterMap()`
 - ◆ 返回当前 `request` 中所有的参数名和参数值组成的 `map` 集合
 - 参数名构成 `Map` 的键
 - 参数的值(是个数组)构成 `Map` 的值
 - 这里的值是个数组, 跟 `getParameterNames()` 返回的值相同
 - ◆ 该方法被认为是 `getParameterNames()` 的替代方案

ServletRequest

■ 获取表单数据举例

- LoginServlet : 构建登陆页面, 负责填充数据
- SuccessServlet : 登陆后的页面, 负责显式收集来的数据



ServletRequest

■ 获取表单数据举例

■ LoginServlet 核心代码

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.print("<html><body><center>");
out.print("<form action='/FirstServlet/success?urlparam=nihao' ");
out.print("method='post'>");
out.print("<table width='300' border='1' cellspacing='1' cellpadding='1'>");
out.print("<tr><td width='80'>UserName</td>");
out.print("<td width='220'><input type='text' name='username' />");
out.print("</td></tr><tr><td>PassWord</td>");
out.print("<td><input type='text' name='password' /></td></tr>");
out.print("<tr><td colspan='2' align='center' >");
out.print("<input type='submit' name='Submit' value='Submit' />");
out.print("</td></tr></table></form>");
out.print("</center></body></html>");
out.flush();    out.close();
```

ServletRequest

■ 获取表单数据举例

■ SuccessServlet 核心代码

```
resp.setContentType("text/html");
PrintWriter out = resp.getWriter();
out.print("<html><body><center>getParameter() : <br>");
out.print( req.getParameter("urlparam") + "<br>" );
out.print( req.getParameter("username") + "<br>" );
out.print( req.getParameter("password") + "<br>" );
out.print("getParameterNames() & getParameterValues() : <br>");
for(Enumeration e = req.getParameterNames(); e.hasMoreElements(); ){
    String strName = (String) e.nextElement();
    String[] values = (String[]) req.getParameterValues(strName);
    for( String value : values ){
        out.print( strName + " : " +value );
    }
    out.print("<br>");
}
out.print("</center></body></html>");
out.flush();      out.close();
```

ServletRequest

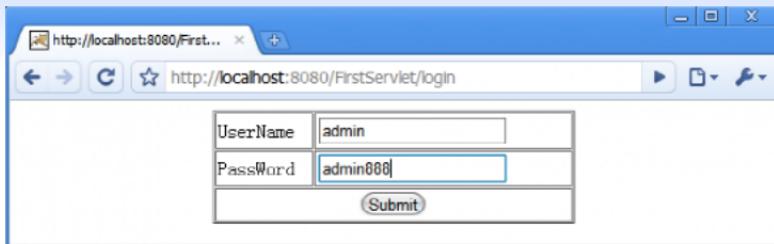
■ 获取表单数据举例

■ web.xml 中添加

```
<!-- LoginServlet -->
<servlet>
    <servlet-name>cn.royallin.servlet.LoginServlet</servlet-name>
    <servlet-class>cn.royallin.servlet.LoginServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>cn.royallin.servlet.LoginServlet</servlet-name>
    <url-pattern>/login</url-pattern>
</servlet-mapping>
<!-- SuccessServlet -->
<servlet>
    <servlet-name>cn.royallin.servlet.SuccessServlet</servlet-name>
    <servlet-class>cn.royallin.servlet.SuccessServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>cn.royallin.servlet.SuccessServlet</servlet-name>
    <url-pattern>/success</url-pattern>
</servlet-mapping>
```

ServletRequest

访问 LoginServlet



提交后的结果页面(SuccessServlet)



ServletRequest

■ 获取报头内容的一般方法

- HttpServletRequest.getHeader(String name)
 - ◆ 返回指定报头名称 name 对应的报头内容
 - ◆ 报头名称不区分大小写
 - req.getHeader("Content-Type")等同于req.getHeader("content-type")
- HttpServletRequest 中定义的获取指定报头的方法
 - ◆ String getAuthType()
 - ◆ Cookie[] getCookies()
 - ◆ long getDateHeader(String name)
 - ◆ String getHeader(String name)
 - ◆ Enumeration getHeaderNames()
 - ◆ Enumeration getHeaders(String name)
 - ◆ int getIntHeader(String name)
 - ◆ String getRemoteUser()

ServletRequest

常见请求报头一览

请求报头名称	说明
Accept	浏览器可以接受的MIME类型
Accept-Charset	浏览器可以接受的字符集
Accept-Encoding	浏览器能够进行解码的数据编码方式
Accept-Language	浏览器所希望的语言种类 当服务器能够提供一种以上语言版本时，要用到此请求信息 特别是在有国际化要求的应用中
Authorizatin	授权信息。 通常出现在对服务器发送的 WWW-authenticate 的头应答中
Content-Length	表示请求消息正文的长度

ServletRequest

常见请求报头一览

请求报头名称	说明
cookie	向服务器返回服务器之前设置的 cookie 信息
host	初始 URL 中的主机和端口 用于获得提出请求的主机的主机名和端口
referer	包含一个 URL , 用户从该 URL 代表的页面出发访问当前请求 即：即从哪个 URL 来到当前的 Servlet 的
User-Agent	浏览器相关信息
Connection	表示是否需要持久连接 如果取值是 Keep-Alive 或该请求使用的是 HTTP1.1, 则可以利用持久连接的优点, 当页面包含多个元素时, 可以显著地减少下载所耗用的时间。 (HTTP 1.1 默认进行持久连接)

ServletRequest

■ 获取请求报头举例

```
//获取用户的浏览器等信息  
String userAgent = request.getHeader( "User-Agent" );
```

```
//获取客户端的 Cookie  
Cookie[ ] cookies = request.getCookies();
```

ServletRequest

■ 获取和设置请求的字符编码

- 获取请求的编码
 - ◆ `ServletRequest#getCharacterEncoding()`
- 设置请求的编码
 - ◆ `ServletRequest#setCharacterEncoding(String env)`

ServletRequest

■ 读取客户端上传的文件

■ Servlet 3.0 以前上传文件

- ◆ 可以使用如下方法获得输入流
 - `ServletRequest#getInputStream()` : 获得字节流
 - `ServletRequest#getReader()` : 获得字符流
- ◆ 以上方式获取的流中包含了
 - 所上传文件的内容
 - 所上传文件的类型
 - 所上传文件的名称
 -
- ◆ 直接使用 Servlet 3.0 以前的 API 实现上传是不太容易的
 - 一般可以借助第三方组件，比较著名的如:
 - » `jspSmartUpload`
 - » `commons-fileupload`
 - 有兴趣可以执行研究这两个组件，此处不做研究

ServletRequest

■ 读取客户端上传的文件

■ Servlet 3.0 新增加了 javax.servlet.http.Part 接口

- ◆ 它表示一个发送multipart/form-data格式的POST请求中所包含的各个部分
- ◆ 其中提供了以下方法
 - InputStream getInputStream() : 获取当前 Part 中所包含的文件的内容
 - String getName() : 获取当前 Part 的名称 (input 标签的 name 属性指定的值)
 - long getSize() : 当前 Part 的内容的大小
 - void write(String fileName) : 把当前 Part 的内容写出到 fileName 对应的路径
 - String getContentType() : 获取当前 Part 所包含的内容的类型
 - void delete() : 删除当前 Part 所对应的内容
- ◆ Servlet 3.0 中获取 Part 对象的方法
 - HttpServletRequest#getPart(String name) : 获取指定名称对应的 Part
 - HttpServletRequest#getParts() : 获取当前表单中所包含的所有的 Part

ServletRequest

■ 读取客户端上传的文件

- 使用 Part , 必须对 Servlet 开启 multipart-config 支持
- 在类中使用 @MultipartConfig 注解
 - ◆ @MultipartConfig 的属性
 - String location : 设置上传后, 文件的存储位置
 - long maxFileSize : 设置允许上传文件的最大值
 - » 默认是 -1L, 即不限制
 - long maxRequestSize : 设置 multipart/form-data 请求的最大值
 - » 默认是 -1L, 即不限制
 - int fileSizeThreshold : 文件写出到磁盘的阀值(即够多大以后写出到磁盘)
- 在 web.xml 中相应的 servlet 标签内使用 <multipart-config />
 - ◆ <multipart-config> 标签也有子标签, 如 location 等
 - ◆ 子标签的作用于上述 @MultipartConfig 的属性作用相同

ServletRequest

■ 读取客户端上传的文件

■ 客户端表单的写法

```
<html>
  <head>
    <title>上传文件</title>
    <meta http-equiv="Content-Type"
          content="text/html; charset=UTF-8">
  </head>
  <body>
    <form action="/yourAppName/servlet/upload"
          method="post" enctype="multipart/form-data">
      FileName <input type="text" name="FileID"> <br>
      FileData <input type="file" name="FileData"> <br>
      <input type="submit" name="uploadfile" value="Upload">
    </form>
  </body>
</html>
```

ServletRequest

■ 在 ServletRequest 中的数据操作

- 将数据内容 o 放入 ServletRequest 中
 - ◆ `setAttribute(String name , Object o)`
- 获取指定 name 的数据
 - ◆ `getAttribute(String name)`
- 删除指定 name 的数据
 - ◆ `removeAttribute(String name)`
- 获取所有数据的 name
 - ◆ `Enumeration getAttributeNames()`

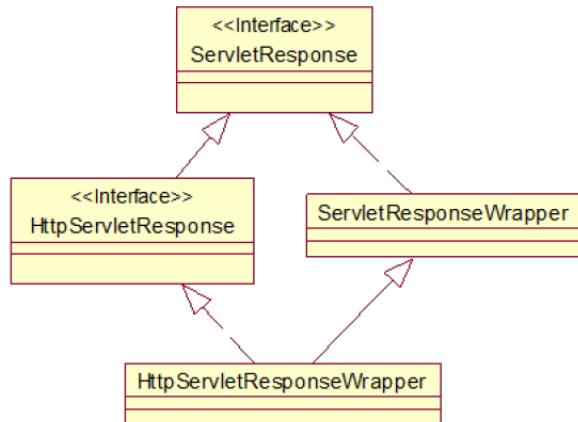
ServletRequest

■ 获取路径

- 获取当前应用的路径
 - ◆ String getContextPath()
- 获取 Servlet 路径
 - ◆ String getServletPath()
- 获取请求的 URL
 - ◆ StringBuffer getRequestURL()
- 获取请求的 URI
 - ◆ String getRequestURI()

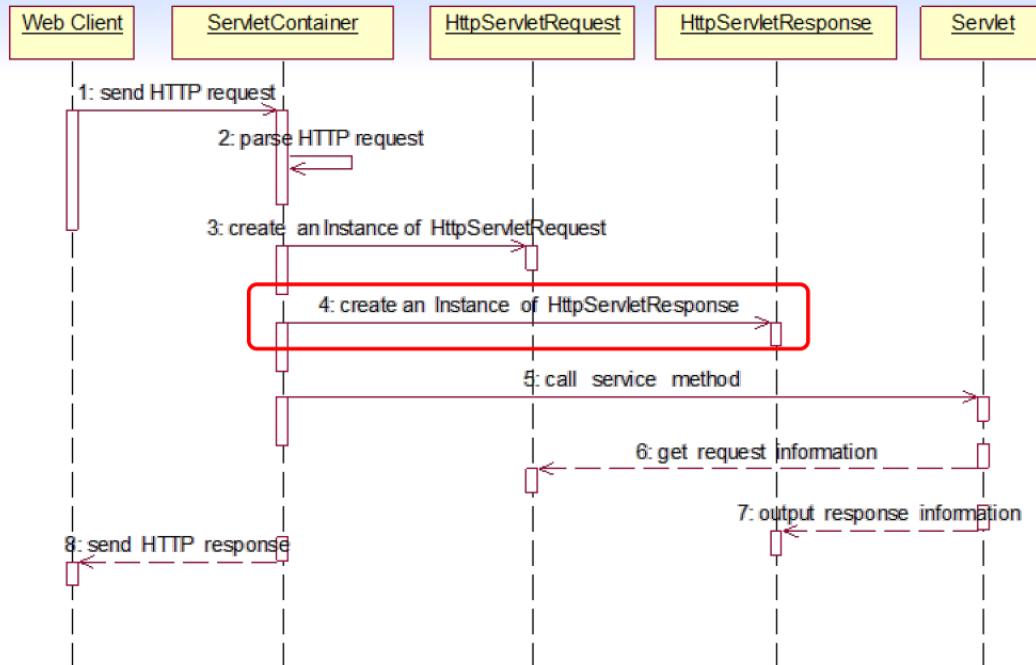
ServletResponse

Servlet API 中 response 层次结构



ServletResponse

ServletReponse 是如何形成的



ServletResponse

服务器响应代码

■ HTTP 1.1 中定义的状态代码

- ◆ 100-199 是信息性代码，标示客户应该采取的其它动作
- ◆ 200-299 表示请求成功
- ◆ 300-399 表示那些已经移走的文件，常包括 Location 报头，指出新的地址
- ◆ 400-499 表示由客户引发的错误
- ◆ 500-599 表示由服务器引发的错误

■ 设置状态代码的通用方法

- ◆ `response.setStatus(int state)`

■ 设置 302 和 404 状态代码

- ◆ 302 命令浏览器链接到新的位置：`sendRedirect(String url)`
 - 该方法会生成 302 响应和 Location 报头；url 可以是相对，也可以是绝对
- ◆ 发送状态代码和简短信息：`setError(int code , String msg)`

ServletResponse

■ 响应报头

■ 设置报头的通用方法

- ◆ `HttpServletResponse.setHeader(String name, String value)`

■ HttpServletResponse 常用的设置报头的方法

- ◆ `setContentType(String mimeType)` 方法设置 Content-type 报头
- ◆ `setContentLength(int length)` 设置Content-length报头
- ◆ `addCookie(Cookie c)` 方法：向 Set-Cookie 报头插入一个 cookie
- ◆ `sendRedirect(String url)` 方法，设置设置状态码和Location报头

■ 判断指定名称的报头是否存在

- ◆ `boolean containsHeader(String name)`

ServletResponse

常用的响应报头

- Allow:
 - ◆ 指定服务器支持的请求方法
- cache-control:
 - ◆ 告诉用户什么环境下可以使用缓存，主要有一系列的值
- connection:
 - ◆ 指示浏览器是否要使用持续性http链接，值有close和open两个
- content-encoding:
 - ◆ 传输过程中应该使用的编码方式
- content-language:
 - ◆ 报头表示文档使用的语言

ServletResponse

常用的响应报头

- content-length:
 - ◆ 报头响应中的字节数
- content-type:
 - ◆ 报头给出的响应文档具体是什么
- refresh:
 - ◆ 表明浏览器应该多长时间（秒）之后请求最新的页面
 - ◆ response.setHeader("refresh", "5, URL=http://www.baidu.com"), 每隔5秒载入页面url
- set-cookie:
 - ◆ 指定一个与页面相关联的cookie
 - ◆ 每个cookie都要求一个单独的set-cookie报头

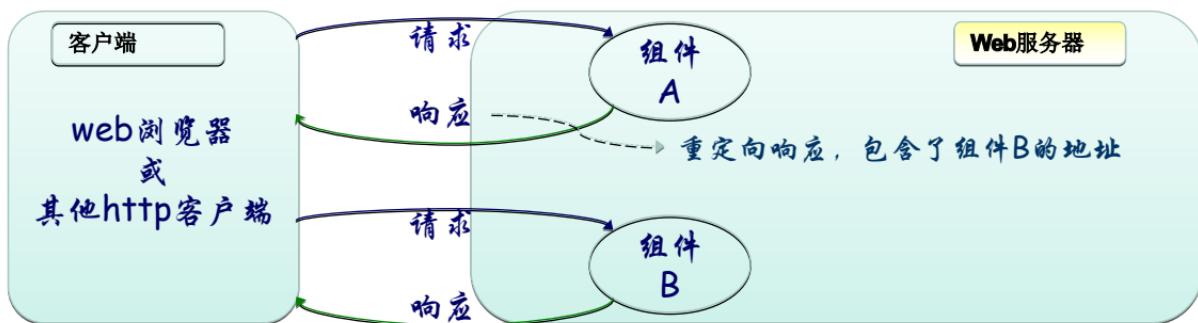
ServletResponse

web组件的调用关系之 重定向 – redirect

使用方法

```
response.sendRedirect("/应用名/访问资源名");
```

重定向原理



重定向中，组件A 的 request 和 response 不会被组件B 所共享或继续使用
使用重定向，浏览器地址栏显式的地址会发生改变！

ServletResponse

■ 重定向使用需注意

- `response.sendRedirect("/应用名/访问资源名");`
 - ◆ 这里，应用名指的是你的工程的名字
 - 也就是在 webapps 目录下的，你要访问的那个目录(文件夹)的名字
 - ◆ 访问资源名，指的是你要访问该站点下的那个地方
 - 可能是一个 servlet，如 login
 - 也可能是一个 jsp 页面或者 html 页面
 - ◆ 其中，两个 / 是必不可少的！
 - /redirect/login
- 重定向中指定的路径
 - ◆ 会显示在浏览器的地址栏上
 - `http://localhost:8080/应用名/访问资源名`
 - ◆ 地址栏中的地址不再是刚才的地址，而是后来的

■ 重定向实例

RequestDispatcher

■ RequestDispatcher

- 它是 javax.servlet 中的一个接口
 - ◆ 使用时的实现类，由Servlet容器提供
- 该接口中所包含的方法(仅此两个)
 - ◆ forward() : 转发
 - ◆ include() : 包含

■ 获取 RequestDispatcher 的途径

- 本部分学习一个方法
 - ◆ ServletRequest#getRequestDispatcher(String path)
- 在 Part 8 中学习以下两个方法
 - ◆ ServletContext#getRequestDispatcher(String path)
 - ◆ ServletContext#getNamedDispatcher(String name)

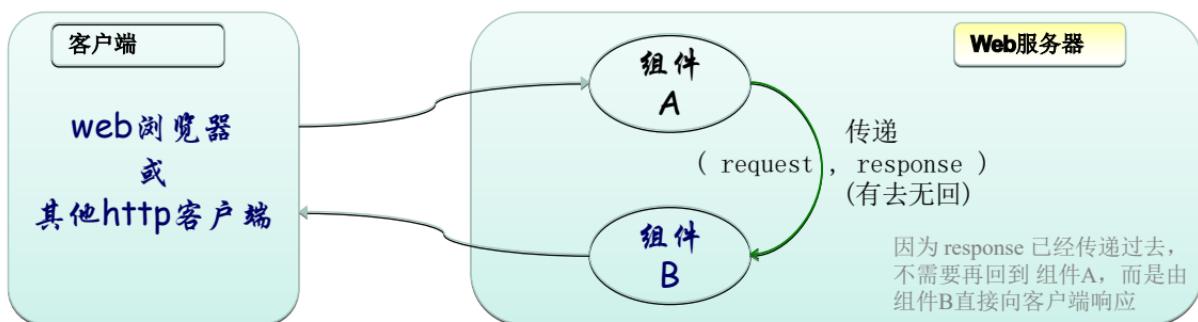
forward

web组件的调用关系之 转发 - forward() 方法

使用方法

```
request.getRequestDispatcher("/相对路径<不包含应用名>")
    .forward(request, response);
```

转发的原理



转发中，组件A的request和response **会**被组件B继续使用(被传递过来)
使用转发，浏览器地址栏显式的地址**不会**发生改变！

include

■ web组件的调用关系之 包含 - include() 方法

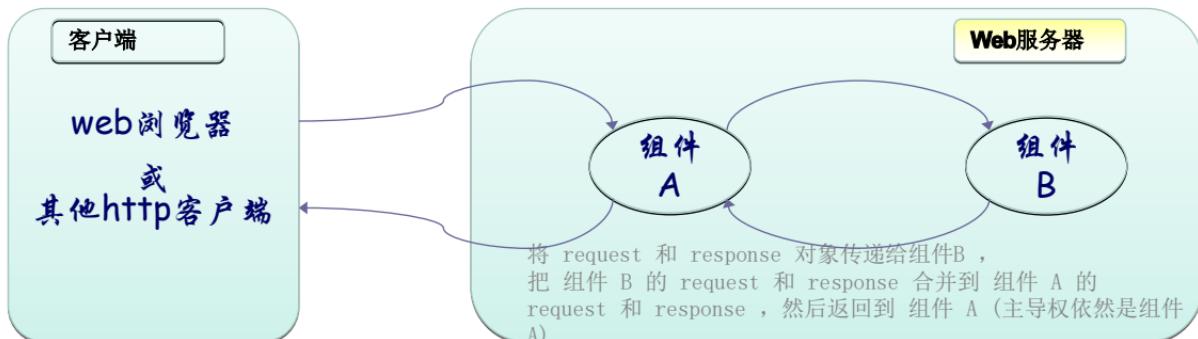
■ 用法

```
request.getRequestDispatcher("/相对路径<不包含应用名>")
    .include(request, response);
```

■ 作用

- 原有的组件和后来的组件将共享 request 对象和 response 对象

■ 包含的原理



forward 与 include 的区别

■ forward与include的区别

■ forward()

- ◆ 在 http 回应被“确认”(committed) 以前才能调用forward()方法
 - 否则将抛出IllegalStateException异常。
 - 这里的“确认”是指将 http 回应的内容主体送回用户端
- ◆ 调用forward()方法后，原先存放在HttpServletResponse对象中的内容将会自动被清除.

■ include()

- ◆ 利用 include() 方法将 http 请求转送给其他 Servlet 后
 - 被调用的 Servlet 虽然可以处理这个 http 请求
 - 但是最后的主导权仍然是在原来的 Servlet
- ◆ 换言之，被调用的 Servlet 如果产生任何 http 回应，将会并入原来的 HttpServletResponse对象。

根据前面的内容，比较 froward 和 sendRedirect 的区别 [这是一道经典面试/笔试题]

什么是会话

一般意义上的会话

- 指两人以上的对话(多用于学习别种语言或方言时)

计算机中的会话

- 在计算机科学领域来说，尤其是在网络领域，会话(session)是一种持久网络协议，在用户(或用户代理)端和服务器端之间建立关联，从而起到交换数据包的作用机制，会话(session)在网络协议(例如telnet或FTP)中是非常重要的部分
- 简而言之，会话是一种面向连接的可靠通信方式

什么是会话跟踪

会话状态

- 每次向Web服务器发送请求，服务器需要记录的相关信息，以便把这些信息传递给你的下一个请求
- 服务器记录这些信息是为了记录请求者的某种状态
- 记录并管理这些信息，一般称之为会话状态管理
- 在 Web 服务器端编程中，会话状态管理是一个经常必须考虑的重要问题

会话跟踪

- 在某个会话中对会话状态进行管理和传递

什么是会话跟踪

会话跟踪的需求

- 源头：http是”无状态”协议
 - ◆ 客户程序每次访问服务器，都打开到服务器的单独连接
 - ◆ 服务器也不自动维护客户的上下文信息
- 上下文的缺失引起许多困难
 - ◆ 无法保持用户是否登录
 - ◆ 无法保存用户已添加到购物车的商品……
- 解决上述不足和缺陷，就是会话跟踪的需要

什么是会话跟踪

会话跟踪的实现方式

■ 隐藏表单域

- ◆ <input type="hidden" name="paramName" value="paramValue">

■ URL 重写

- ◆ http://localhost:8080/loginServlet
- ◆ http://localhost:8080/loginServlet;jsessionid=1234

■ 基于客户端的跟踪 : Cookie

- ◆ javax.servlet.http.Cookie

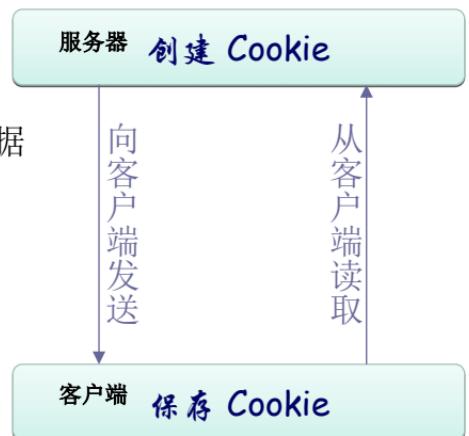
■ 基于服务器端的跟踪 : Session

- ◆ javax.servlet.http.HttpSession

Cookie - 理解Cookie

什么是 Cookie

- Cookie 是由服务器创建，保存在客户端的小文本数据
 - ◆ Cookie 由服务器端创建
 - 在服务器端是 Cookie 对象
 - 服务器端可以创建和封装一个 Cookie 对象
 - ◆ 在客户端保存，可以传回服务器
 - 在客户端是小段的文本信息
 - 一个 Cookie 存储在客户机上并包含状态信息
 - ◆ Cookie 本质上是 key-value 形式的小文本数据
 - 一般形式是 cookieName=cookieValue
 - ◆ 作用是记录客户状态
- Cookie 中含有的信息
 - ◆ Cookie的名字、Cookie的值
 - ◆ Cookie的失效日期
 - ◆ Cookie的域名和路径



Cookie - 理解Cookie

Cookie可以做什么

- 在电子商务会话中标识用户
 - ◆ 见下一部分(会话跟踪)
- 记录用户名和密码
 - ◆ 登陆论坛或信箱时，可以选择记住用户名、记住密码等
- 定制站点
 - ◆ 根据客户爱好，定制针对该客户的页面
- 定向广告
 - ◆ 根据客户的爱好和取向，定制他感兴趣的广告

Cookie - 理解Cookie

Cookie的不足及使用时需注意

■ 不足

- ◆ 比如公用计算机上，你的浏览记录被他人一览无余
- ◆ 隐私信息被恶意记录

■ 使用时需注意

- ◆ 由于现实的和可预见的隐私问题，某些用户选择关闭cookie
- ◆ 作为开发人员，应避免用cookie保存比较敏感的信息
- ◆ 如果需要保存敏感信息到 cookie 可以对这些信息加密处理

Cookie - 使用Cookie

■ 创建 Cookie 对象

- Cookie 类的构造

```
public Cookie ( String name , String value )
```

- Cookie 的 name 和 value 的命名要求

- ◆ 不论是 name 和 value 都不能含有空格和以下任何字符
 - ▣ [] () = , " / ? @ : ;
- ◆ 对于 name , 其命名还必须满足以下要求
 - ▣ 不能是以下单词或单词组合(大小写不一样都不行)
 - » Comment、Discard、Domain、Expires、Max-Age、Path、Secure、Version
 - ▣ 不能以 \$ 为开头
 - ▣ Cookie 类里对 name 的命名做了检查, 所以必须满足以上两个条件

- 一个简单的 Cookie 可以通过构造完成

- ◆ 其他的诸如域名、最长有效期一般会由容器完成

Cookie - 使用Cookie

显式设置 Cookie 最大时效

- 通过当前 Cookie 对象的 setMaxAge() 方法设置

- public void setMaxAge(int expiry)

- 本方法中的 expiry 参数

- 默认的时间单位是 秒 (seconds)
 - 比如 Cookie 的最大时效是一周就是 setMaxAge(60*60*24*7)
 - 将最大时效设置为 0 , 即 expiry = 0
 - 意思是命令浏览器删除该 Cookie
 - 如果没有显式设置最大时效, 默认为 -1
 - 1 的意思是当浏览器关闭时, 删除相应的 Cookie

```
Cookie cookie = new Cookie("cookie-name","cookie-value");
```

```
cookie.setMaxAge( 60 * 60 * 24 * 7 );
```

Cookie - 使用Cookie

■ 将 Cookie 添加到 http 响应报头

- 调用的方法

```
response.addCookie( cookie );
```

- 该方法不修改之前的 set-cookie 报头

- ◆ 该方法的作用是在报头之后添加新的内容
- ◆ 如果已经有同名的 cookie 存在，那么覆盖原有的内容
 - 事实上，JavaScript 中也是这么处理的

- 该方法是创建新的报头

- ◆ 因此方法名曰 addCookie 而非 setCookie

- 该方法会把服务器上的操作"同步"到客户端

- ◆ 只有调用了 addCookie ，客户端才会生效
- ◆ 否则操作的都是服务器端的数据，跟客户端没有任何关系

Cookie - 使用Cookie

■ 将 Cookie 添加到 http 响应报头

- 创建Cookie对象以及调用setMaxAge是在服务器中完成的
 - ◆ 操作的只不过是服务器内存中的数据
 - ◆ 并没有向浏览器(客户端)发送任何内容
- 不将 Cookie 发送到客户端没有任何意义
 - ◆ 这是初级开发人员常犯的低级错误
 - 想要删除某个 cookie , 所以调用了 cookie.setMaxAge(0)
 - 重新访问后, 发现相应的 cookie 并没有删除
 - 这时只需把你对相应 cookie 的操作发送给客户端

```
cookie.setMaxAge(0); //命令浏览器删除 cookie
```

```
//如果没有下一行, 浏览器是不会删除 cookie 的  
response.addCookie(cookie);
```

Cookie - 使用Cookie

从客户端读取 Cookie

- 对于没有禁用 cookie 的浏览器
 - ◆ 一般会发送 cookie 给相应的网站
 - 我们不需要关注浏览器是怎么发送 cookie 的，这是浏览器的事
 - 除非你想研究浏览器
- 从客户端读取 Cookie 需要两个步骤
 - ◆ 通过 request 获取当前所关联的所有 cookie 数组

```
Cookie[] cookies = request.getCookies();
```

- ◆ 遍历 cookie 数组，根据名称，可以找到相应的值

```
for( Cookie cookie : cookies){  
    System.out.println( cookie.getName() + " - " + cookie.getValue() );  
}
```

Cookie - Cookie实例

完整的 Cookie 使用实例

- 前面四个部分的代码片段连起来，就是个完整的实例
 - ◆ 完成一个记录用户登录信息，实现下次自动登录的实例
- 纸上谈来终觉浅，绝知此事须躬行
 - ◆ 代码是敲出来的，不是看出来的
 - ◆ 诸公还是多加练习！



Session - 理解 Session

■ Session也是一种记录客户状态的机制

- 服务器把客户信息以某种方式记录在服务器
- Cookie 把客户状态记录在客户端
- Session 把客户状态记录在服务器上

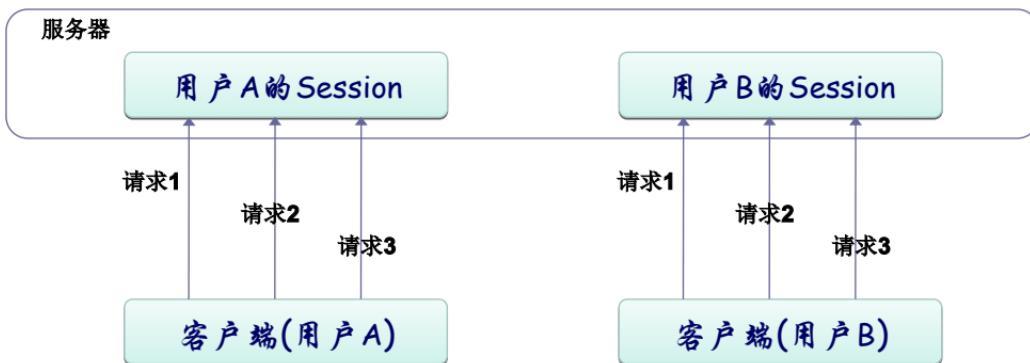
■ Session中数据的特点

- 由服务器创建，存储在服务器上
- 类似于 Cookie，也是以 key-value 形式存放数据
 - ◆ 不同的是 value 可以是任何形式，比如可以是 Java 对象等
- 通常 Session 中也包含了失效时间等信息
- 每个"用户"都会有且仅有一个 Session
 - ◆ 访问动态资源比如 JSP、Servlet 等会导致创建 Session
 - ◆ 访问静态资源，比如 html 、 image 等不会导致创建 Session
 - ◆ 也可以通过 `request.getSession(true)` 强制开启 Session

Session - 理解 Session

使用 Session 可以实现跨请求的数据共享

- 同一个"用户"的多次请求所共用的数据可以存放在 Session 中
 - ◆ 比如用户登录后的信息，可以记录在 Session 中，以后的请求可以这些数据
- 必须是同一个"用户"的不同请求才能共享 Session 中的数据
 - ◆ 不同"用户"使用不同的 Session
 - ◆ 不同"用户"的请求不能共享 Session 中的数据



Session - 理解 Session

Session的有效期

- 服务器会把长时间没有活跃的 Session 从内存中删除
 - ◆ 为了防止服务器端内存溢出，减轻服务器端负荷
- 通常 Session 在超过一定时间后，Session 就自动失效
 - ◆ 这个时间就是 Session 的有效期
 - ◆ 超时时间可以通过设置 maxInactiveInterval 属性来设置
- 可以通过修改 web.xml 设置 Session 的有效期

```
<session-config>
    <session-timeout>60</session-timeout> <!-- 单位是分钟 -->
</session-config>
```

- 通过 Session 的 invalidate() 可以使 Session 失效
 - ◆ 比如，用户注销时，可以使用 该方法

Session - HttpSession中的常用方法

- Object getAttribute(String name)
- Enumeration getAttributeNames()
- long getCreationTime()
- String getId()
- long getLastAccessedTime()
- int getMaxInactiveInterval()
- ServletContext getServletContext()
- void invalidate()
- boolean isNew()
- void removeAttribute(String name)
- void setAttribute(String name, Object value)
- void setMaxInactiveInterval(int interval)

Session - Session 对浏览器的要求

■ Session 使用 Cookie 作识别标志

- 因为 HTTP 是无状态协议，因此不能根据 HTTP 链接来标识客户
- 服务器向客户端发送一个 JSESSIONID 的 Cookie 来标识客户
 - ◆ 通过该 jsessionid 可以唯一地标识一个客户
 - ◆ 该 jsessionid 的值就是当前 session 对象的 id，在服务器是唯一的
- 服务器会自动生成 JSESSIONID 的 Cookie
 - ◆ 它的 maxAge 默认为 -1，表示当前浏览器有效
 - ◆ 一旦用户关闭浏览器，立即失效，同时各个窗口之间互不共享
 - ◆ 但，当前浏览器窗口的子窗口可以共享父窗口的Cookie（亦即共享Session）

■ Session 对浏览器的要求是支持并开启 Cookie 功能

- 一般而言，对于 Linux / Windows 都开启了 Cookie 功能
- 早期的 wap 浏览器都不支持 Cookie 功能
 - ◆ 如果你用的是早期的手机，其中的 wap 浏览器是不支持 Cookie 的

Session - URL 重写

■ 必要性

- 对于浏览器不支持 Cookie 的情况，可以使用 URL 重写
 - ◆ 本身就是对客户端不支持 Cookie 的解决方案
- 实际上，对于绝大多数的 wap 应用，都采用了 URL 重写

■ 原理

- 将该用户 Session 的 id 信息重写到 URL 地址中
- 服务器可以解析重写后的 URL 从而获取 Session 的 id
- 即便客户端不支持 Cookie 也可以使用 Session 来记录客户状态

■ 使用方法

- `response.encodeURL("index?xxx=123") ;`
- `response.encodeRedirectURL("index?xxx=123");`

Session - 使用步骤

■ 获得与当前请求相关联的会话对象

- `HttpSession session = request.getSession();`

■ 查找与会话相关联的信息

- 通过 `session.getAttribute(attrName);` 获取

■ 存储会话中的信息

- 通过 `session.setAttribute(key , value);` 存入

■ 废弃会话

- 通过 `session.removeAttribute(attrName);` 废弃指定的值
- 调用 `session.invalidate()` 废弃整个会话
- 会话有效期超出，会话被服务器干掉

Session - 使用Session

■ 使用 Session 实现的计数器

- 记录某个用户的信息
- 记录该用户的某一时间段的访问次数
 - ◆ 所谓某一时间段是指 Session 有效期范围内

■ 使用 Cookie 实现的计数器

- 记录某个用户的信息
- 统计该用户的访问次数
 - ◆ 客户端必须支持 Cookie , 并且不删除 Cookie

Session - 使用Session

Session使用实例

- 完成一个 `cn.eshop.entity.Product` 类，用于模拟商品
 - ◆ 其中包括商品名称、商品单价、商品描述等属性
- 完成一个 `cn.eshop.entity.ShoppingCar` 类，用于模拟购物车
 - ◆ 使用 Map 作为底层存放容器
 - ◆ 实现如下功能
 - 添加商品、删除商品、清空购物车、修改商品数量
- 在 `cn.eshop.servlet` 包中完成相关的 Servlet
 - ◆ 增、删、改 功能对应 3 个 Servlet
 - ◆ 查看购物车使用 一个 Servlet
 - ◆ 完成一个商品列表的 Servlet 输出到浏览器
- 测试你完成的购物车

Session pk Cookie

- 存取方式
- 隐私安全
- 有效期
- 服务器负载
- 浏览器支持
- 跨域名支持

ServletConfig

■ 用于描述一个 Servlet 的配置信息

- 获得当前 Servlet 实例的名称
 - ◆ `String getServletName()`
- 获得当前 Servlet 实例的初始化参数
 - ◆ `String getInitParameter(String name)`
 - 根据 name 获取指定的初始化参数的值
 - 如果 name 对应的 初始化参数不存在，返回 null
 - ◆ `Enumeration getInitParameterNames()`
 - 返回当前 Servlet 实例的所有的初始化参数的名称
 - 注意返回类型是 `java.util.Enumeration`
- 获得当前 Servlet 实例所依赖的应用
 - ◆ `ServletContext getServletContext()`

ServletConfig

在配置文件中配置 Servlet

配置 Servlet 的初始化参数

- ◆ 在 servlet 标签中使用 init-param 子标签配置
 - 其中 <param-name> 子标签用于设置初始化参数的名称
 - <param-value> 子标签用于设置指定参数的值
- ◆ 同一个 servlet 中可以使用多个 init-param 标签配置多个初始化参数
 - 使用 config.getInitParameter(name) 可以获取指定名称的参数对应的值
 - 使用 config.getInitParameterNames() 可以获取所有的参数的名称

```
<servlet>
    ...
    <init-param>
        <param-name>parameterName</param-name>
        <param-value>parameterValue</param-value>
    </init-param>
    ...
</servlet>
```

ServletConfig

在配置文件中配置Servlet

- 指定当 web 应用启动时，装载 Servlet 的次序
 - ◆ 在 servlet 标签中使用 <load-on-startup> 子标签设置
 - ◆ 其中的数字含义如下：
 - 如果这个数字是正数或0，则容器先加载数值较小的 servlet，再加载数值稍大的
 - 如果是负数或者没有设置，容器将在 web 客户端首次访问该 servlet 时才加载

```
<servlet>
    ...
    <init-param>
        <param-name>parameterName</param-name>
        <param-value>parameterValue</param-value>
    </init-param>
    ...
    <load-on-startup>2</load-on-startup>
</servlet>
```

ServletContext

Servlet容器的上下文环境对象

- 其类型是 javax.servlet.ServletContext
 - ◆ Servlet 容器启动时会加载每个 web 应用
 - ◆ 并为每个 web 应用创建一个惟一的 ServletContext 对象
- 可以理解成是一个 web 应用的服务器端组件的共享内存
 - ◆ 服务器端的所有组件都可以从该对象中获取或设置数据
 - 用于操纵数据的方法包括:
 - » Object getAttribute(String name)
 - » Enumeration getAttributeNames()
 - » void removeAttribute(String name)
 - » void setAttribute(String name, Object object)
 - ◆ 使用该对象可以实现跨 Session 的数据共享
- 获得当前的 ServletContext 对象
 - ◆ GenericServlet#getServletContext() (HttpServlet继承了GenericServlet)
 - ◆ HttpSession#getServletContext()
 - ◆ ServletConfig#getServletContext()

ServletContext

被 ServletContext 管理的数据

- 可以被整个 web 应用所使用
- 可以实现跨会话的数据共享
- 对应 JSP 中的 application 内置对象



ServletContext

配置ServletContext

配置 ServletContext 初始化参数

- 在 web-app 标签中使用 <context-param> 子标签来设置
 - <param-name> 指定初始化参数的名称
 - <param-value> 指定初始化参数的值
- 可以使用设置多个初始化参数

获得初始化参数

- String getInitParameter(String name)
- Enumeration getInitParameterNames()

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ... ... >
    ...
    ...
    <context-param>
        <param-name>parameterName</param-name>
        <param-value>parameterValue</param-value>
    </context-param>
</web-app>
```

ServletContext

ServletContext 中的常用方法

- 获得指定 uripath 对应的 ServletContext 对象
 - ◆ `ServletContext getContext(String uripath)`
- 与路径有关的
 - ◆ `String getContextPath()`
 - ◆ `String getRealPath(String path)`
- 获得 RequestDispatcher 对象
 - ◆ `RequestDispatcher getNamedDispatcher(String name)`
 - ◆ `RequestDispatcher getRequestDispatcher(String path)`
- 获得 Servlet 的相关信息
 - ◆ `String getServerInfo()`
 - ◆ `String getServletContextName()`
- 获得输入流
 - ◆ `InputStream getResourceAsStream(String path)`
 - Class 类的 `getResourceAsStream` 需要用到类加载器，而该方法不需要

Filter Overview

Java Servlet 2.3 中新增加的功能

- 主要作用是对 Servlet 容器的请求和响应进行检查和修改

Filter 本身并不生成请求和响应对象

■ 它只提供过滤作用

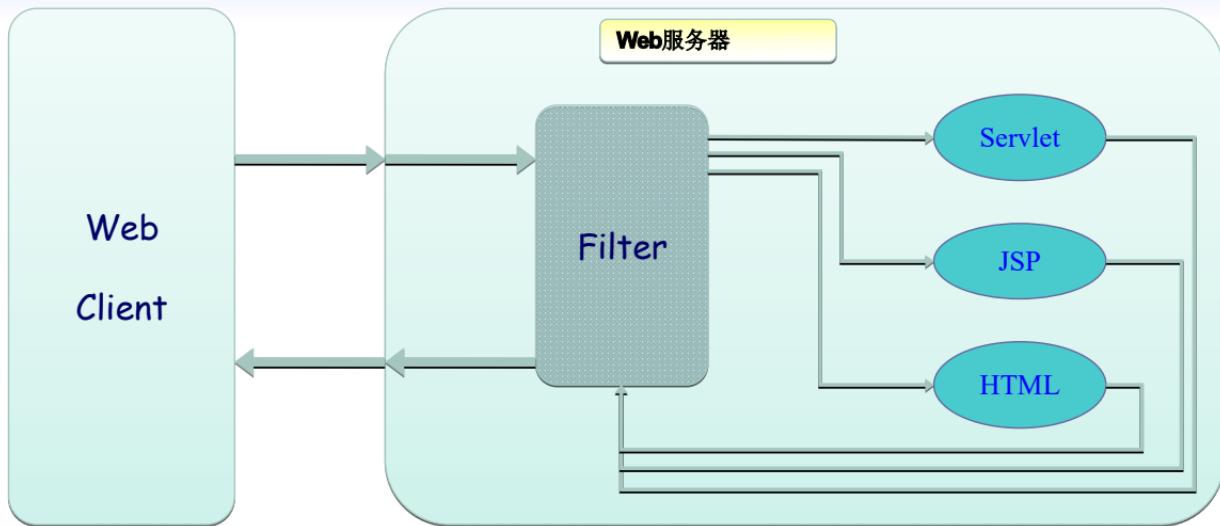
- ◆ 在 Servlet 被调用之前，检查 Request 对象
 - 可以对其 Request Header 和 Request 内容进行审查和修改
- ◆ 在 Servlet 调用结束之后，检查 Response 对象
 - 可以对其 Response Header 和 Response 内容进行审查和修改

Filter 可以过滤的 Web 组件包括

- Servlet
- JSP
- HTML

Filter Overview

Filter 过滤过程



Filter Interface

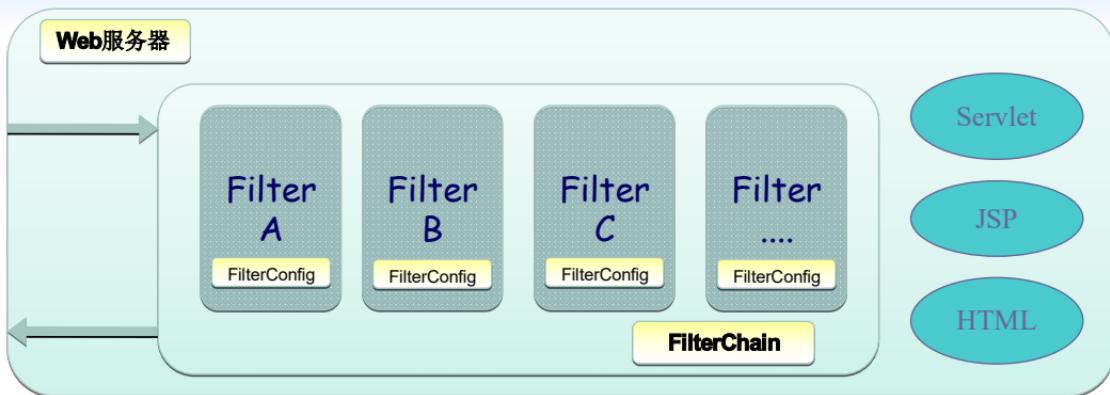
■ javax.servlet.Filter 中定义的方法

- `init(FilterConfig config)`
 - ◆ Filter 的初始化方法
 - ◆ 容器创建 Filter 之后将调用这个方法
 - ◆ 使用这个方法可以读取 web.xml 文件中定义的初始化参数
- `doFilter(ServletRequest req,ServletResponse resp,FilterChain chain)`
 - ◆ 该方法完成实际的过滤操作
 - ◆ 当客户请求访问与 Filter 相关联的 URL 时，将调用该方法
 - ◆ chain 用于访问后续的 Filter 或 Servlet
- `destroy()`
 - ◆ 容器在销毁 Filter 实例前调用该方法
 - ◆ 该方法中可以释放该 Filter 所占用的资源

FilterChain Interface

■ javax.servlet.FilterChain 接口

■ 过滤器链



■ 该接口中定义的方法

- doFilter(ServletRequest req,ServletResponse resp)
 - ◆ 负责把所有的过滤器给串联起来
 - ◆ 使得一个过滤器执行完后，下一个可以继续执行
 - ◆ 被串联的多个过滤器按照配置文件中的映射顺序依次执行

Create Filter

实现 Filter

```
public class ExceptionFilter implements Filter {  
    public void destroy() {  
        System.out.println(" Filter destroy ");  
    }  
    public void doFilter(ServletRequest req, ServletResponse resp,  
        FilterChain chain) throws IOException, ServletException {  
        try{  
            chain.doFilter(req, resp);  
        }catch( Exception e ){  
            resp.setContentType("text/html;charset=utf-8");  
            resp.getWriter().println("出错了:<br>" + e.getClass().getName() +  
                "<br><pre>" + e.getMessage() + "</pre>");  
        }  
    }  
    public void init(FilterConfig config) throws ServletException {  
        System.out.println(" Filter Start ... ");  
    }  
}
```

Deploy Filter

■ 注册 Filter

```
<filter>
    <filter-name>exception</filter-name>
    <filter-class>com.malajava.ExceptionFilter</filter-class>
</filter>
```

■ 发布 Filter

- 同 Servlet 一样，url-pattern 可以写多个

```
<filter-mapping>
    <filter-name>exception</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Test Filter

■ 在任意页面或Servlet中抛出异常

- 创建一个 jsp 页面，内容如下

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>测试 Filter </title>
</head>
<body>
    <%
        throw new Exception("出错了"); //抛出一个异常
    %>
</body>
</html>
```

Test Filter

在任意页面或Servlet中抛出异常

- 使用浏览器访问页面的效果图



Config init-param

- 在 web.xml 中可以指定初始化参数

```
<filter>
    <filter-name>exception</filter-name>
    <filter-class>com.malajava.filter.ExceptionFilter</filter-class>
    <init-param>
        <param-name>initParam</param-name>
        <param-value>initValue</param-value>
    </init-param> <!-- 初始化参数可以有多个，只需要写多个 init-param 即可-->
</filter>
```

- 访问初始化参数

- 在 init 方法中通过 FilterConfig 对象访问

```
config.getInitParameter("initParam");
//注意这里的参数，必须跟 param-name 中的参数名称写一致
```

Annotation Supported

■ Servlet 3.0 允许使用注解来标注 Filter

■ @WebFilter

- ◆ 用以标注一个实现过 Filter 接口的类
- ◆ @WebFilter 的常用属性
 - String filterName 指定当前Filter 的名称，相当于 xml 中的 filter-name
 - String[] value 指定当前 Filter 对应的 url (与 url-pattern 对应)
 - String[] urlPatterns 与 value 作用相同
 - DispatcherType[] dispatcherTypes 指定当前 Filter 关联的 dispatcher 类型
 - 默认值是 DispatcherType.REQUEST
 - javax.servlet.DispatcherType 是 Servlet 3.0 新定义的枚举
 - boolean asyncSupported 指定是否支持异步操作
 - WebInitParam[] initParams 用于设置 Filter 初始化参数
- @WebInitParam 用法见 @WebServlet 部分

使用 **@WebFilter** 标注不如 web.xml 文件中可以通过映射顺序来控制过滤器的执行顺序

Examples of the Filter

完成以下实例

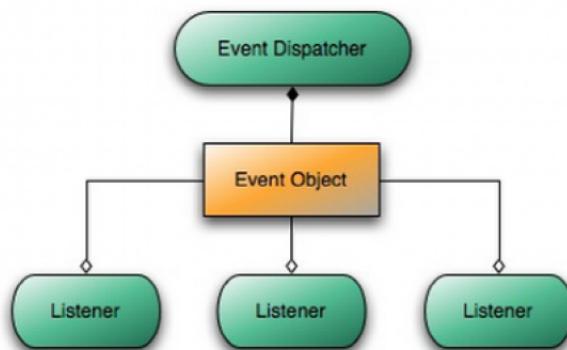
- 使用 Filter 实现把所有的request和response设置为统一编码
 - ◆ 这是解决页面乱码问题的一个常用方法
- 使用过滤器实现记录用户的操作
 - ◆ 比如用户的ip是什么，用户的用户名是什么，访问了那些资源
 - ◆ 访问这些资源用了多长时间等等信息
- 过滤请求中的不良信息，对这些信息作出替换
 - ◆ 思路：把数据正常存入数据库中，但是以后显示在页面时，做替换
 - ◆ 比如把 共匪 处理成 **，把 王小丫 处理成 王**
- 使用 Filter 实现防盗链
 - ◆ 意思是除了本站外，不允许站外使用本站的图片等资源

Listener Overview

Servlet 2.3 中新增加的另一个功能

- 作用是监听 Java Web 程序中的事件
- 对应设计模式中的 Listener 模式
 - ◆ 当事件发生的时候会自动触发该事件对应的 Listener
- 主要用于对 Request、Session、Context 等进行监控

Listener 模型



Listener & Event

■ ServletContextListener

- 接口方法
 - ◆ contextInitialized() 、 contextDestroyed()

- 接受事件 : ServletContextEvent

- 触发场景
 - ◆ Container 加载 Web 应用程序时呼叫 contextInitialized()
 - ◆ Container 移除 Web 应用程序时呼叫 contextDestroyed()

■ ServletContextAttributeListener

- 接口方法
 - ◆ attributeAdded() 、 attributeReplaced() 、 attributeRemoved()

- 接受事件 : ServletContextAttributeEvent

- 触发场景
 - ◆ 有对象新加入到 application (ServletContext) 呼叫 attributeAdded()
 - ◆ 有对象被置换/移除时，呼叫 attributeReplaced() / attributeRemoved()

Listener & Event

■ HttpSessionListener

- 接口方法
 - ◆ sessionCreated() 、 sessionDestroyed()
- 接受事件 : HttpSessionEvent
- 触发场景
 - ◆ 当 session 对象被创建时呼叫 sessionCreated()
 - ◆ 当 session 对象被销毁时呼叫 sessionDestroyed()

■ HttpSessionAttributeListener

- 接口方法
 - ◆ attributeAdded() 、 attributeReplaced() 、 attributeRemoved()
- 接受事件 : HttpSessionBindingEvent
- 当有对象新加入session，或者被置换、删除时呼叫上述方法

Listener & Event

HttpSessionActivationListener

- 接口方法
 - ◆ sessionDidActivate()、sessionWillPassivate()
- 接受事件：HttpSessionEvent
- 触发场景
 - ◆ Activate 操作时呼叫 sessionDidActivate()
 - ◆ Passivate 操作时呼叫 sessionWillPassivate()

Activate 和 Passivate

- 都是用于置换对象的动作
- Passivate
 - ◆ session对象为了资源利用或负载平衡等原因必须暂时存储至硬盘或其它存储设备(透过序列化完成存储)所做的动作称之为 Passivate (钝化)
- Activate
 - ◆ 硬盘或其它存储设备上的session对象重新加载到JVM的动作

Listener & Event

■ HttpSessionBindingListener

- 接口方法
 - ◆ valueBound() 、 valueUnbound()
- 接受事件 : HttpSessionBindingEvent
- 触发场景
 - ◆ 对于实现 HttpSessionBindingListener 的类的实例
 - 如果该实例加入 session 对象的属性，则呼叫 valueBound()
 - 如果该实例从 session 对象中移除，则呼叫 valueUnbound()

Listener & Event

■ ServletRequestListener

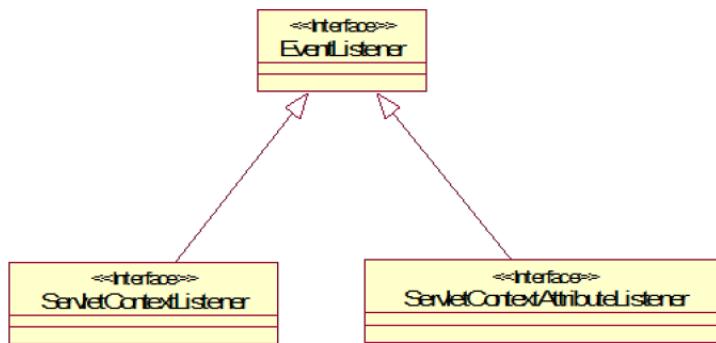
- 接口方法
 - ◆ requestInitialized() 、 requestDestroyed()
- 接受事件 : ServletRequestEvent
- 触发场景
 - ◆ request在被创建或销毁时会分别呼叫这两个方法

■ ServletRequestAttributeListener

- 接口方法
 - ◆ attributeAdded() 、 attributeReplaced() 、 attributeRemoved()
- 接受事件 : ServletRequestAttributeEvent
- 触发场景
 - ◆ 当request有对象加入、置换、移除时分别呼叫上述方法

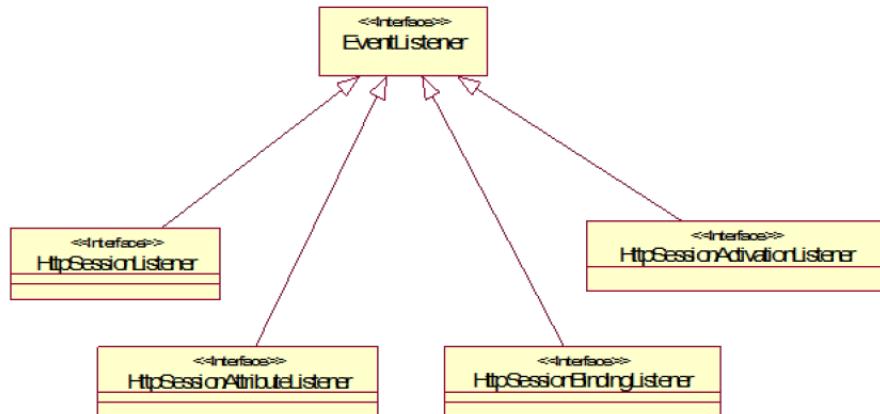
Listener & Event

- Class associated with the ServletContext



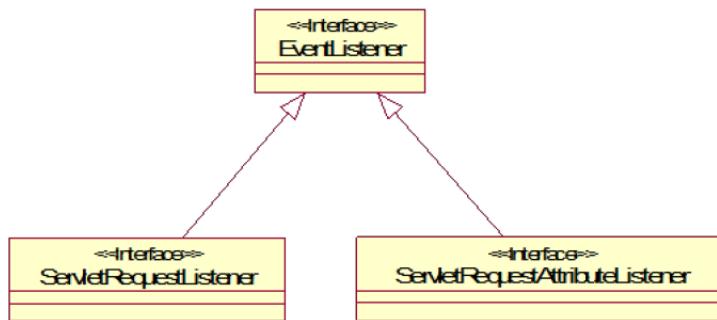
Listener & Event

- Class associated with the HttpSession



Listener & Event

- Class associated with the ServletRequest



Registe Listener

必须注册 Listener，容器才能为相应事件找到处理器

- 除了 HttpSessionBindingListener 外所有的 Listener 都必须注册
- 只有注册过的 Listener 容器才能进行调度
- 注册 Listener
 - ◆ 可以在 web.xml 中添加相应的配置

```
<listener>
    <listener-class>com.malajava.listener.RSAListener</listener-class>
</listener>
```

- 配置文件中，一般把 Listener 的配置放在 servlet 和 filter 之前
- Servlet 3.0 可以在监听器类中使用 @WebListener 注解
 - 这个注解用于标注这个类是一个 监听器，并注册给容器
 - @WebListener 只有一个属性 value，它是 String 类型，指定当前 Listener 的描述

eg: listen to request and session and context

```
public class RSAListener implements  
    ServletRequestListener,  
    HttpSessionListener , ServletContextListener {  
  
    public void requestDestroyed(ServletRequestEvent reqEvent) {  
        HttpServletRequest request =  
            (HttpServletRequest)reqEvent.getServletRequest();  
        String uri = request.getRequestURI();  
        String queryString = request.getQueryString();  
        uri = queryString == null ? uri : ( uri + "?" + queryString );  
        String ip = request.getRemoteAddr();  
        long time = System.currentTimeMillis()  
            - (Long)request.getAttribute("createTime");  
        System.out.println( "IP[ " + ip + " ]请求[ " + uri + " ]结束, 用时[ "  
            + time + " ]毫秒");  
    }  
  
    //接下页
```

eg: listen to request and session and context

//续上页

```
public void requestInitialized(ServletRequestEvent reqEvent){  
    HttpServletRequest request =  
        (HttpServletRequest)reqEvent.getServletRequest();  
    String uri = request.getRequestURI();  
    String queryString = request.getQueryString();  
    uri = queryString == null ? uri : ( uri + "?" + queryString );  
    String ip = request.getRemoteAddr();  
    request.setAttribute("createTime", System.currentTimeMillis() );  
    System.out.println( "IP[ " + ip + " ]请求 " + uri + " 开始" );  
}
```

//接下页

eg: listen to request and session and context

//续上页

```
public void sessionCreated(HttpSessionEvent sessionEvent) {  
    HttpSession session = sessionEvent.getSession();  
    System.out.println("新创建一个 session , ID 为 " + session.getId() );  
}  
  
public void sessionDestroyed(HttpSessionEvent sessionEvent) {  
    HttpSession session = sessionEvent.getSession();  
    System.out.println("销毁一个 session , ID 为 " + session.getId() );  
}
```

//接下页

eg: listen to request and session and context

//续上页

```
public void contextDestroyed(ServletContextEvent contextEvent) {  
    ServletContext context = contextEvent.getServletContext();  
    System.out.println("即将销毁应用: " + context.getContextPath());  
}  
  
public void contextInitialized(ServletContextEvent contextEvent) {  
    ServletContext context = contextEvent.getServletContext();  
    System.out.println("即将启动应用: " + context.getContextPath());  
}
```

}//类体结束

Examples of the Listener

统计登录的人数

■ 原理

- ◆ 每个用户登录都开启 HttpSession , 注销或超时后都销毁 HttpSession

■ 实现步骤

- ◆ 当用户登录时, 创建 HttpSession 对象, 此时计数器增加
- ◆ 当用户注销时, 或用户长时间无操作, 销毁 HttpSession 对象, 计数器减少
- ◆ 因为是统计整个站点的登录人数, 计数器需要保存在 ServletContext 范围内

■ 统计最高访问量

- ◆ 当当前的登录人数比历史最高访问量大时, 替换原来的值

■ 拓展

- ◆ 如何统计当前站点的在线人数?
 - 除了登录的人数外, 还有未登录但是却在线浏览的人数
- ◆ 如何区分登录人数和在线人数?

任他巨力来打我，牵引四两拔千斤

Author: cnhanxj@gmail.com