

javaweb-jsp文档

第一章 Tomcat 基础

一、web 概念

1. 软件架构

1. C/S: 客户端/服务器端 -----> QQ, 360 ... client server
2. B/S: 浏览器/服务器端 -----> 京东, 网易, 淘宝 browser/server

2. 资源分类

1. **静态资源**: 所有用户访问后, 得到的结果都是一样的, 称为静态资源。静态资源可以直接被浏览器解析。
如图片、视频、
2. **动态资源**: 每个用户访问相同资源后, 得到的结果可能不一样, 称为动态资源。动态资源被访问后, 需要先转换为静态资源, 再返回给浏览器, 通过浏览器进行解析。
 - 如: servlet,jsp,php,asp....

二、常见的web服务器

1、概念

1. 服务器: 安装了服务器软件的计算机
2. 服务器软件: 接收用户的请求, 处理请求, 做出响应
3. web服务器软件: 接收用户的请求, 处理请求, 做出响应。

在web服务器软件中, 可以部署web项目, 让用户通过浏览器来访问这些项目

2、常见服务器软件

动态服务器

- webLogic: oracle公司, 大型的JavaEE服务器, 支持所有的JavaEE规范, 收费的。
- webSphere: IBM公司, 大型的JavaEE服务器, 支持所有的JavaEE规范, 收费的。
- JBOSS: JBOSS公司的, 大型的JavaEE服务器, 支持所有的JavaEE规范, 收费的。
- Tomcat: Apache基金组织, 中小型的JavaEE服务器, 仅仅支持少量的JavaEE规范servlet/jsp。开源的, 免费的。(300左右的并发)

静态的服务器

- nginx (代理, 反向代理等) 极高的并发

- apache

3、Tomcat 历史


- 1) Tomcat 最初由Sun公司的软件架构师 James Duncan Davidson 开发，名称为“JavaWebServer”。
- 2) 1999年，在 Davidson 的帮助下，该项目于1999年于apache 软件基金会旗下的 JServ 项目合并，并发布第一个版本（3.x），即是现在的Tomcat，该版本实现了Servlet2.2 和 JSP 1.1 规范。
- 3) 2001年，Tomcat 发布了4.0版本，作为里程碑式的版本，Tomcat 完全重新设计了其架构，并实现了 Servlet 2.3 和 JSP1.2规范。

目前 Tomcat 已经更新到 10.0.x版本，但是目前企业中的Tomcat服务器，主流版本还是 7.x 和 8.x。

4、Tomcat 安装

下载

<https://tomcat.apache.org/download-80.cgi>

 apache-tomcat-8.5.42-windows-x64.zip

安装

将下载的 .zip 压缩包，解压到系统的目录（建议是没有中文不带空格的目录）下即可。

5、Tomcat 目录结构

Tomcat 的主要目录文件如下：

目录	目录下文件	说明
bin	/	存放Tomcat的启动、停止等批处理脚本文件
	startup.bat , startup.sh	用于在windows和linux下的启动脚本
	shutdown.bat , shutdown.sh	用于在windows和linux下的停止脚本
conf	/	用于存放Tomcat的相关配置文件
	Catalina	用于存储针对每个虚拟机的Context配置
	context.xml	用于定义所有web应用均需加载的Context配置，如果web应用指定了自己的context.xml，该文件将被覆盖
	catalina.properties	Tomcat 的环境变量配置
	catalina.policy	Tomcat 运行的安全策略配置
	logging.properties	Tomcat 的日志配置文件，可以通过该文件修改Tomcat 的日志级别及日志路径等
	server.xml	Tomcat 服务器的核心配置文件
	tomcat-users.xml	定义Tomcat默认的用户及角色映射信息配置
	web.xml	Tomcat 中所有应用默认的部署描述文件，主要定义了基础Servlet和MIME映射。
lib	/	Tomcat 服务器的依赖包
logs	/	Tomcat 默认的日志存放目录
webapps	/	Tomcat 默认的Web应用部署目录
work	/	Web 应用JSP代码生成和编译的临时目录

6、Tomcat 启动停止

启动

双击 bin/startup.bat 文件；

停止

双击 bin/shutdown.bat 文件；

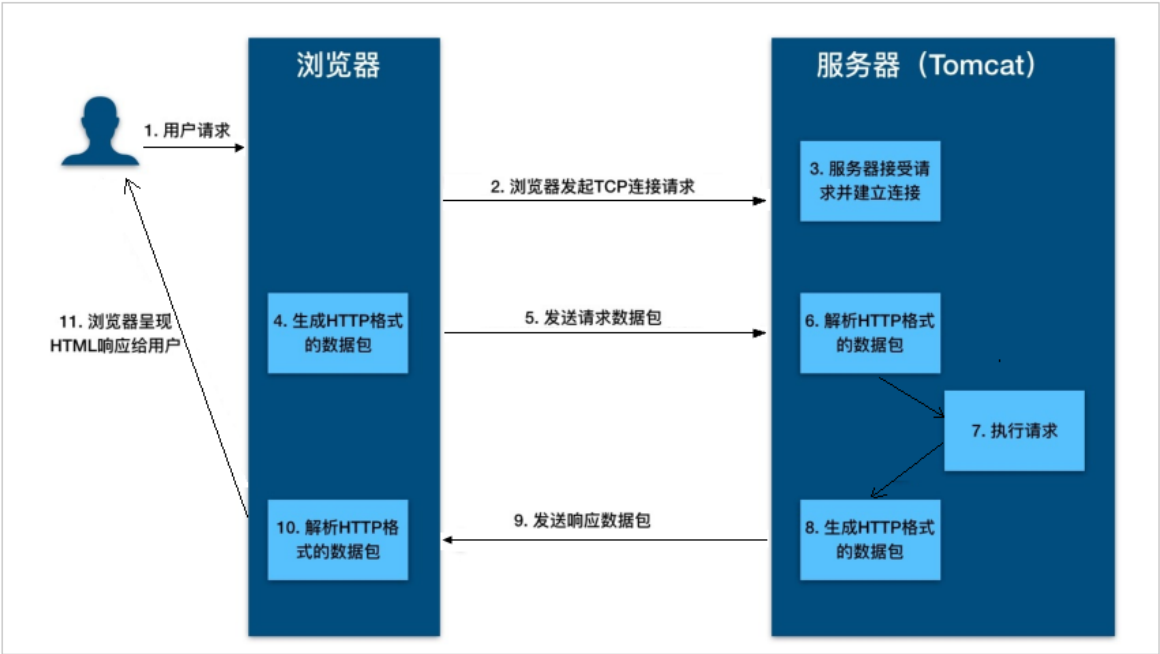
访问

http://localhost:8080

三、Tomcat 架构

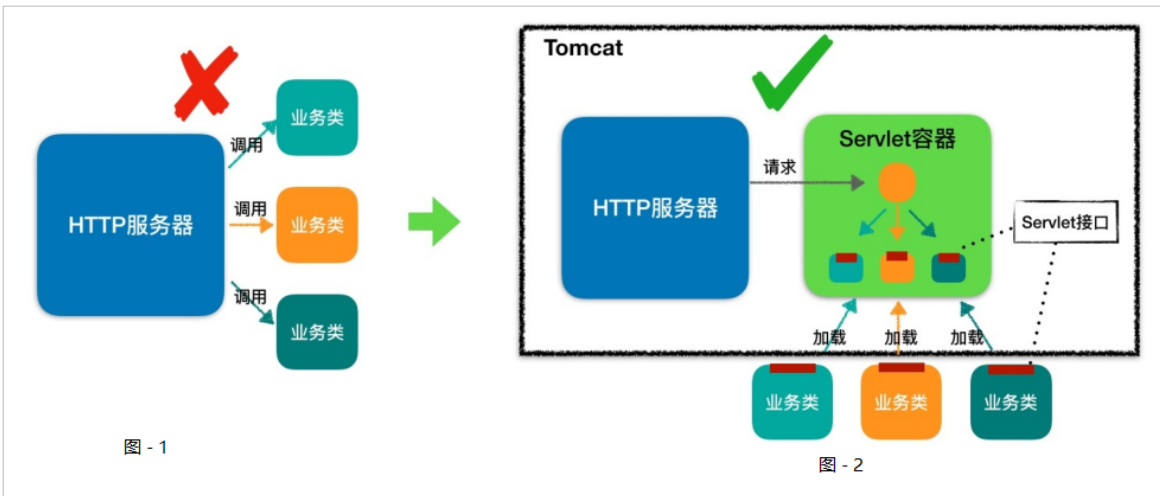
1、Http工作原理

HTTP协议是浏览器与服务器之间的数据传送协议。作为应用层协议，HTTP是基于TCP/IP协议来传递数据的（HTML文件、图片、查询结果等），HTTP协议不涉及数据包（Packet）传输，主要规定了客户端和服务端之间的通信格式。



2、Http服务器请求处理

浏览器发给服务端的是一个HTTP格式的请求，HTTP服务器收到这个请求后，需要调用服务端程序来处理，所谓的服务端程序就是你写的Java类，一般来说不同的请求需要由不同的Java类来处理。



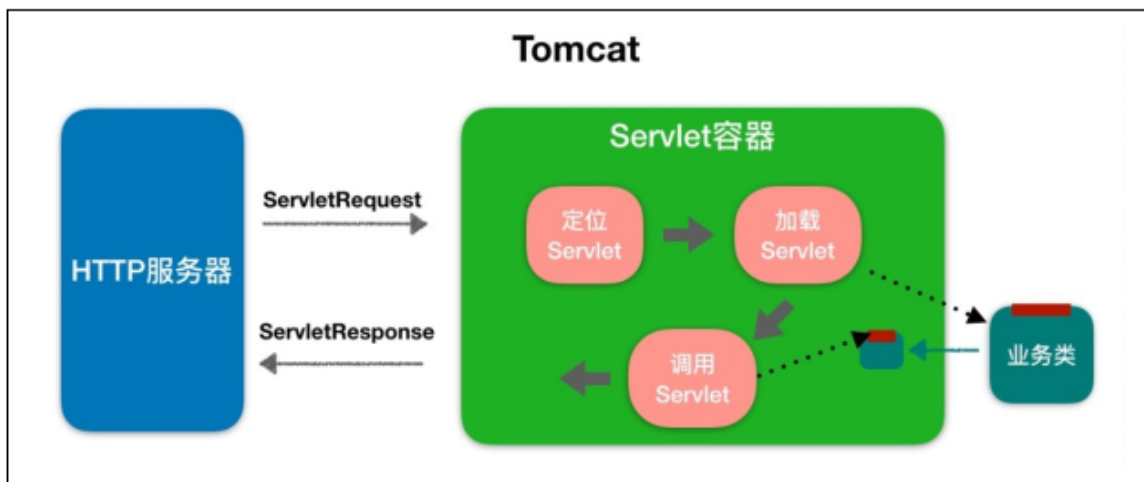
- 1) 图1，表示HTTP服务器直接调用具体业务类，它们是紧耦合的。
- 2) 图2，HTTP服务器不直接调用业务类，而是把请求交给容器来处理，容器通过Servlet接口调用业务类。因此Servlet接口和Servlet容器的出现，达到了HTTP服务器与业务类解耦的目的。而Servlet接口和Servlet容器这一整套规范叫作Servlet规范。Tomcat按照Servlet规范的要求实现了Servlet容器，同时它们也具有HTTP服务器的功能。作为Java程序员，如果我们要实现新的业务功能，只需要实现一个

Servlet，并把它注册到Tomcat（Servlet容器）中，剩下的事情就由Tomcat帮我们处理了。

3、Servlet容器工作流程

为了解耦，HTTP服务器不直接调用Servlet，而是把请求交给Servlet容器来处理，那Servlet容器又是如何工作的呢？

当客户请求某个资源时，HTTP服务器会用一个ServletRequest对象把客户的请求信息封装起来，然后调用Servlet容器的service方法，Servlet容器拿到请求后，根据请求的URL和Servlet的映射关系，找到相应的Servlet，如果Servlet还没有被加载，就用反射机制创建这个Servlet，并调用Servlet的init方法来完成初始化，接着调用Servlet的service方法来处理请求，把ServletResponse对象返回给HTTP服务器，HTTP服务器会把响应发送给客户端。

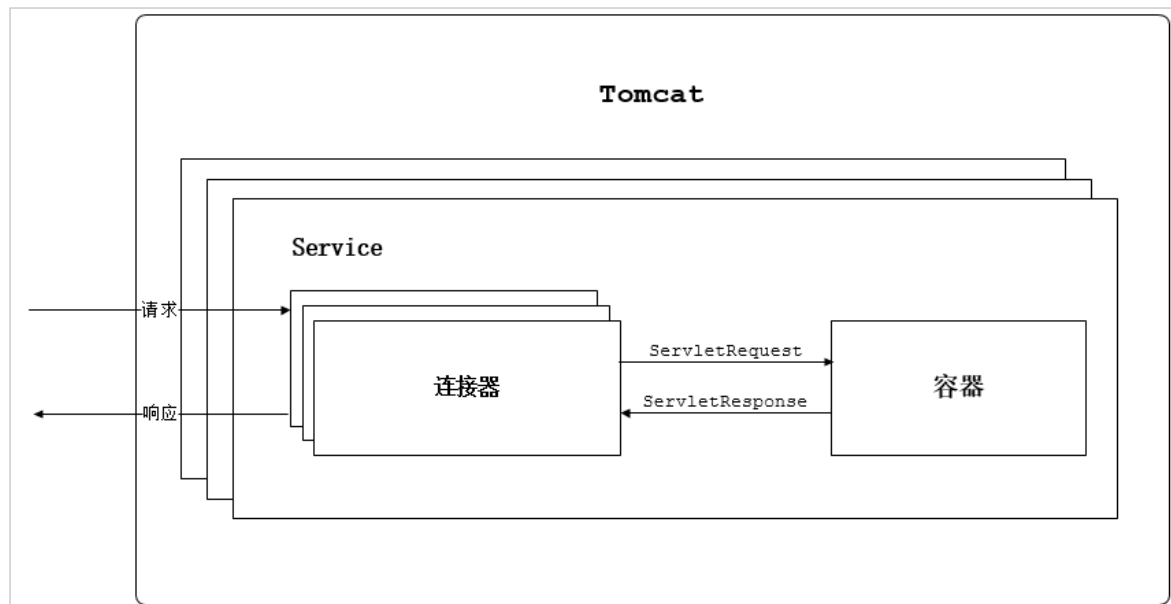


4、Tomcat整体架构

我们知道如果要设计一个系统，首先是要了解需求，我们已经了解了Tomcat要实现两个核心功能：

- 1) 处理Socket连接，负责网络字节流与Request和Response对象的转化。
- 2) 加载和管理Servlet，以及具体处理Request请求。

因此Tomcat设计了两个核心组件连接器（Connector）和容器（Container）来分别做这两件事情。连接器负责对外交流，容器负责内部处理。



四、Tomcat 服务器配置

Tomcat 服务器的配置主要集中于 tomcat/conf 下的 catalina.policy、catalina.properties、context.xml、server.xml、tomcat-users.xml、web.xml 文件。

1、server.xml

server.xml 是tomcat 服务器的核心配置文件，包含了Tomcat的 Servlet 容器（Catalina）的所有配置。由于配置的属性特别多，我们在这里主要讲解其中的一部分重要配置。

Server

Server是server.xml的根元素，用于创建一个Server实例，默认使用的实现类是 org.apache.catalina.core.StandardServer。

```
<Server port="8005" shutdown="SHUTDOWN">
    ...
</Server>
```

port : Tomcat 监听的关闭服务器的端口。

shutdown： 关闭服务器的指令字符串。

Connector

Connector 用于创建链接器实例。默认情况下，server.xml 配置了两个链接器，一个支持HTTP协议，一个支持AJP协议。因此大多数情况下，我们并不需要新增链接器配置，只是根据需要对已有链接器进行优化。

```
<Connector port="8080" protocol="HTTP/1.1" connectionTimeout="20000"
    redirectPort="8443" />
```

属性说明：

1) port： 端口号，Connector 用于创建服务端Socket 并进行监听， 以等待客户端请求链接。如果该属性设置为

0，Tomcat将会随机选择一个可用的端口号给当前Connector 使用。

2) protocol： 当前Connector 支持的访问协议。默认为 HTTP/1.1 。

3) connectionTimeOut： Connector 接收链接后的等待超时时间， 单位为 毫秒。 -1 表示不超时。

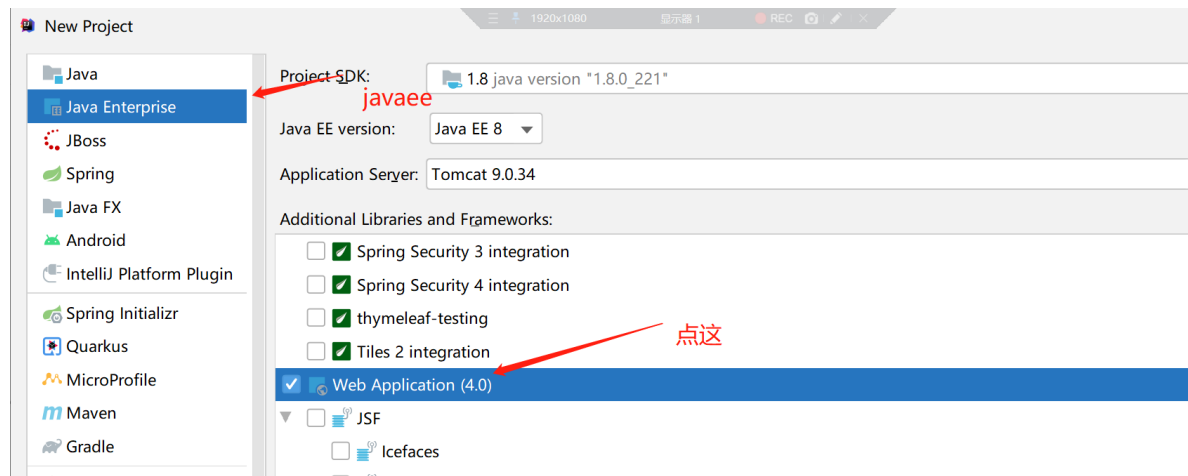
4) URLEncoding： 用于指定编码URI的字符编码， Tomcat8.x版本默认的编码为 UTF-8 。

完整的配置如下：

```
<Connector port="8080"
    protocol="HTTP/1.1"
    executor="tomcatThreadPool"
    maxThreads="1000"
    minSpareThreads="100"
    acceptCount="1000"
    maxConnections="1000"
    connectionTimeout="20000"
    compression="on"
    compressionMinSize="2048"
    disableUploadTimeout="true"
    redirectPort="8443"
    URIEncoding="UTF-8" />
```

五、创建javaweb项目

1、创建项目

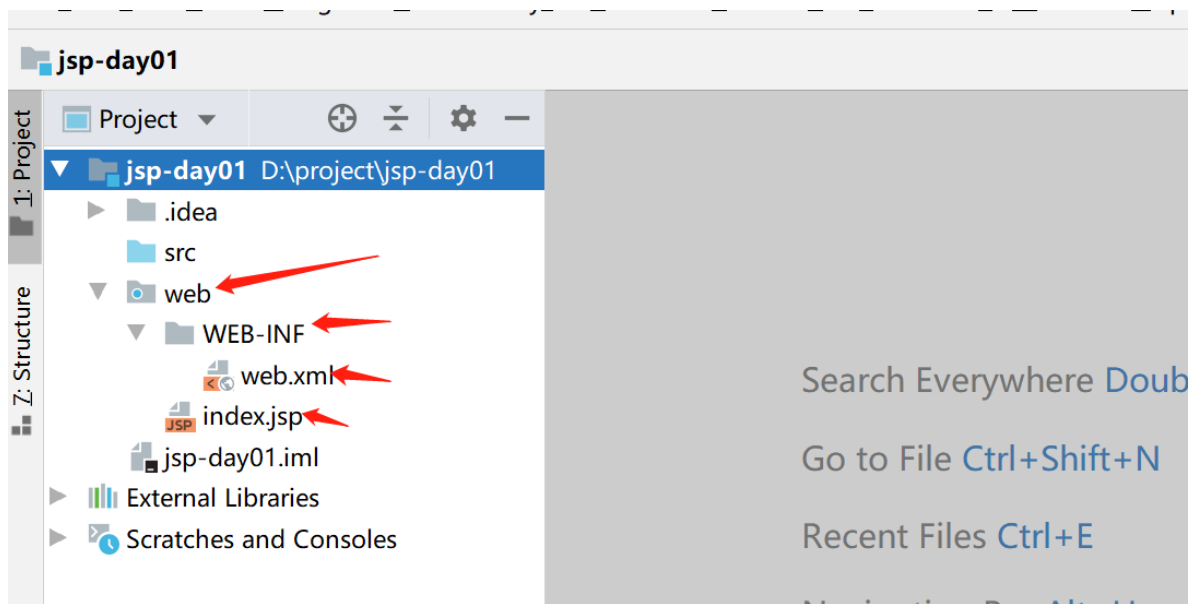


2、创建名字

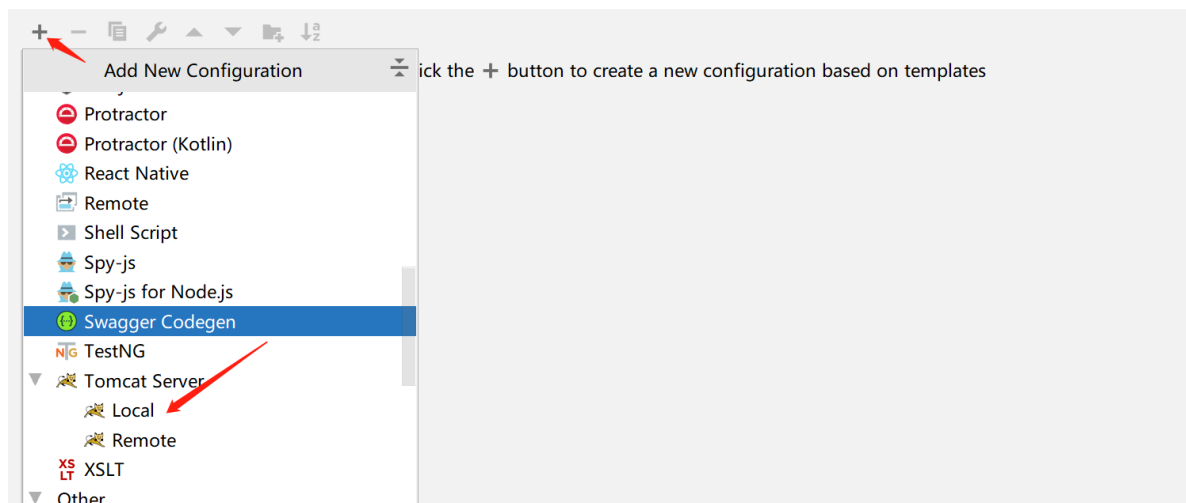
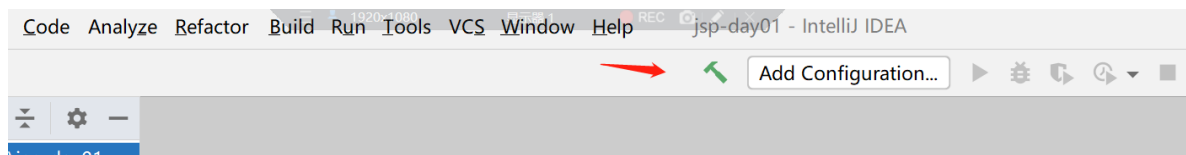
Project name:

Project location:

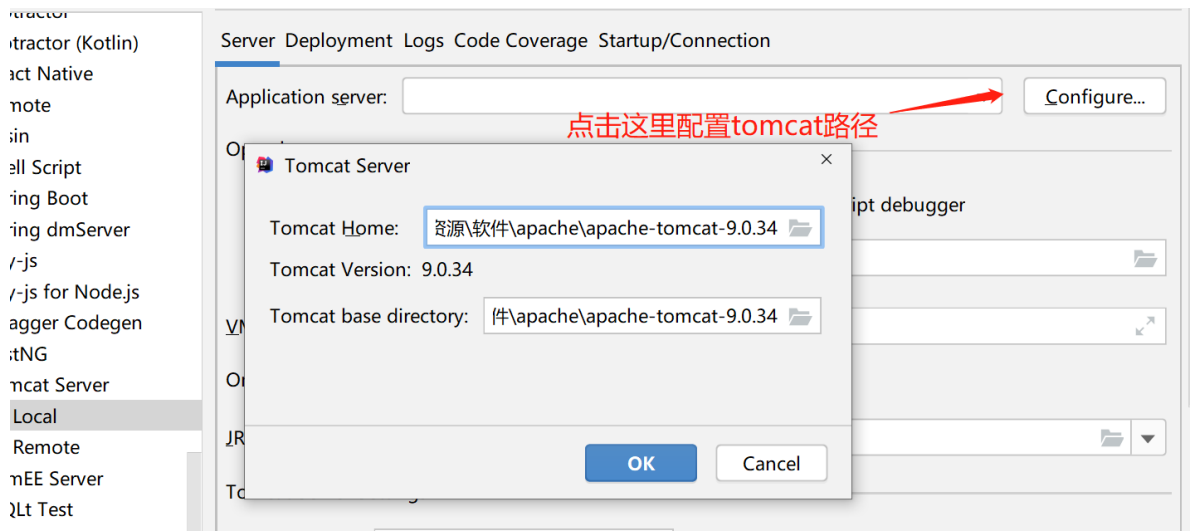
3、项目结构



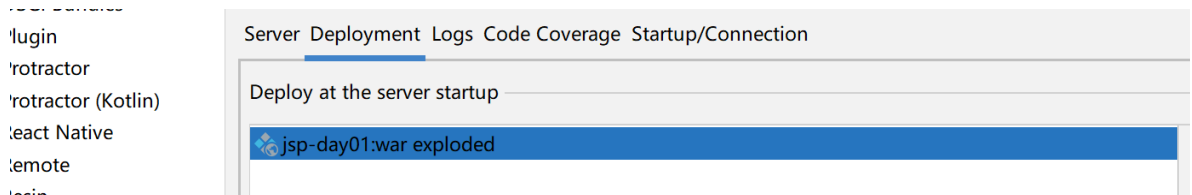
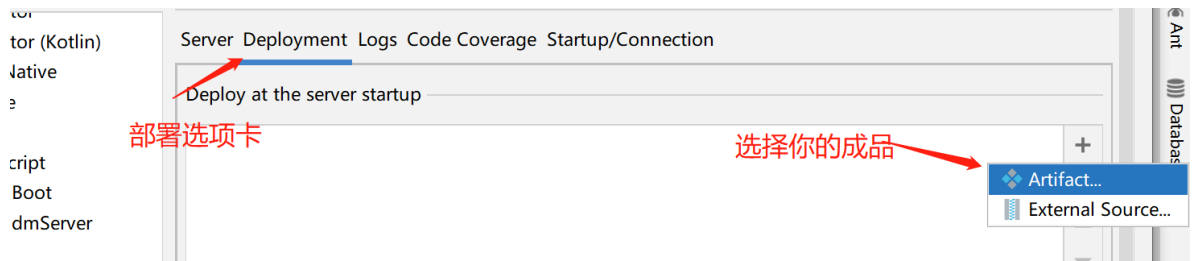
4、配置服务器



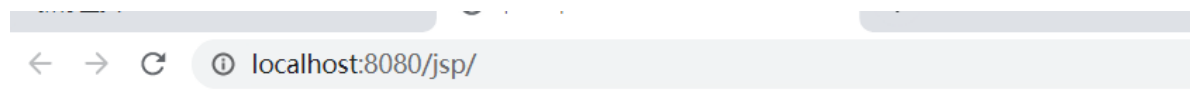
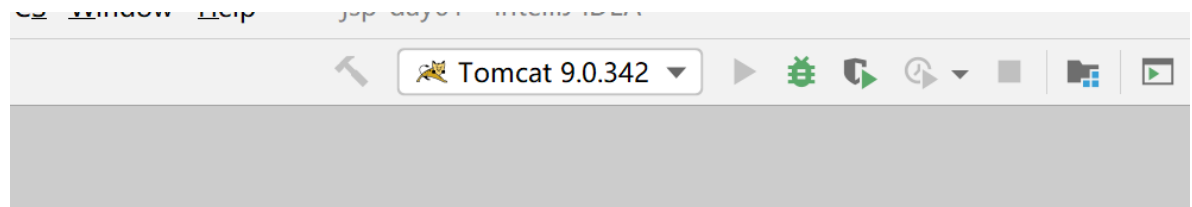
5、配置web服务器 (tomcat)



6、部署文件



8、启动



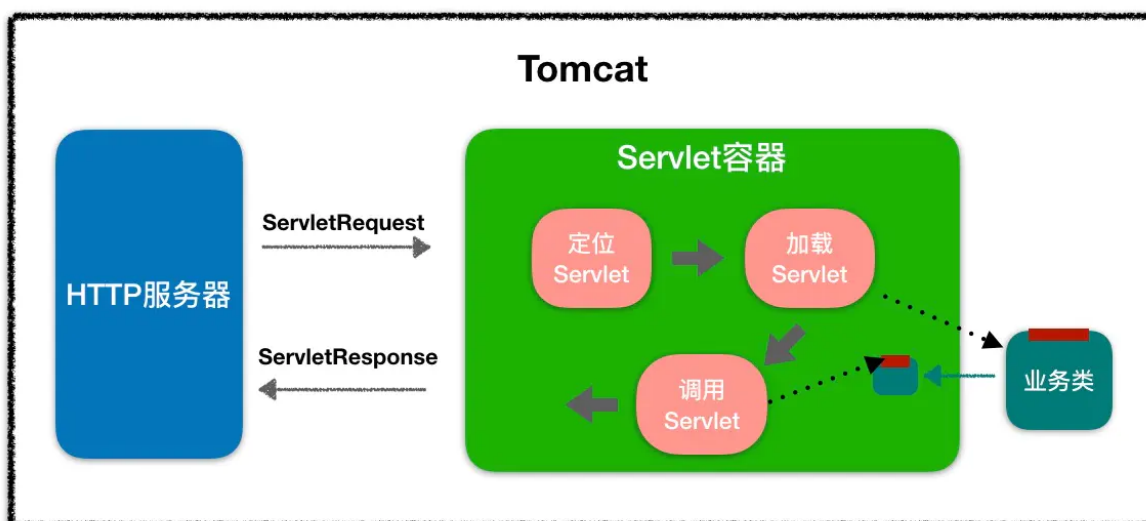
\$END\$

第二章 Servlet入门

Servlet就是一个java程序，用来处理请求和响应。

一、Servlet架构

下图展示了Servlet在Web应用程序中的位置：



二、Servlet任务

- 处理请求request，生成响应response

三、Servlet相关知识

1、Servlet加载时机

在默认情况下，当Web客户**第一次请求访问某个Servlet时**，Web容器会创建这个Servlet的实例。

当设置了web.xml中的子元素后，Servlet容器在启动Web应用时，将按照指定顺序创建并初始化这个Servlet。设置的数值**大于0**即可。例如：

```
<servlet>
  <servlet-name>HelloServlet</servlet-name>
  <servlet-class>com.longsin.servlet.HelloServlet</servlet-class>
  <load-on-startup>2</load-on-startup>
</servlet>
```

2、Servlet的生命周期

先看与Servlet生命周期有关的三个方法：init(), service(), destroy(). Servlet生命周期可被定义为从创建直到毁灭的整个过程。以下是三个方法分别对应的Servlet过程：

- init(): Servlet进行初始化;
- service(): Servlet处理客户端的请求;
- destroy(): Servlet结束, 释放资源;

在调用destroy()方法后, Servlet由JVM的垃圾回收器进行垃圾回收。

现在我们来详细讨论Servlet生命周期的方法：

init()方法:

Servlet被装载后, Servlet容器创建一个Servlet实例并且调用Servlet的init()方法进行初始化在Servlet生命周期中init()方法**只被调用一次**。

当用户调用一个Servlet时, Servlet容器就会创建一个Servlet实例, **每一个用户请求都会产生一个新的线程**, init()方法简单的创建或加载一些数据, 这些数据将会被用在Servlet的整个生命周期。

init()方法的定义如下:

```
public void init() throws ServletException {
    // 初始化代码...
}
```

service()方法:

service()方法是执行实际任务的主要方法。Servlet 容器 (即 Web 服务器) 调用 service()方法来处理来自客户端 (浏览器) 的请求, 并把格式化的响应写回给客户端。

每次服务器接收到一个 Servlet 请求时, 服务器会产生一个新的线程并调用服务。service()方法检查 HTTP 请求类型 (GET、POST、PUT、DELETE 等), 并在适当的时候调用doGet()、doPost()等方法。

service()的定义如下:

```
public void service(ServletRequest request, ServletResponse response)
    throws ServletException, IOException{
    // service()代码...
}
```

destroy()方法:

destroy()方法只会被调用一次，在Servlet生命周期结束时被调用。destroy()方法可以让Servlet关闭数据库连接、停止后台、把cookie列表或点击计数器写入到磁盘，并执行其他类似的清理活动。在调用destroy()方法之后，Servlet对象被标记为垃圾回收。

destroy()方法的定义如下所示：

```
public void destroy() {  
    // 终止化代码...  
}
```

总结：

- 在首次访问某个Servlet时，init()方法会被执行，而且也会执行service()方法。
- 再次访问时，只会执行service()方法，不再执行init()方法。
- 在关闭Web容器时会调用destroy()方法。

3、实现一个servlet

当服务器接收到一个请求，就要有一个servlet去处理这个请求，所以完成一个servlet通常需要两步走。一方面要写一个java程序定义一个servlet，另一方面要配置一下servlet确定这个servlet要处理哪一个请求。

1、创建Servlet的三种方式

- (1) 实现javax.servlet.Servlet接口。
- (2) 继承javax.servlet.GenericServlet类
- (3) 继承javax.servlet.http.HttpServlet类

我们在日常开发中一般会使用第三种方法来进行Servlet的创建，前两种方法理解即可。

注意：创建Servlet文件后，需要在web.xml文件中完成Servlet配置，才可以使用。

2、配置Servlet的两种方式

(1) 使用web.xml文件配置Servlet。例如，我有一个名为ServletDemo的Servlet，主要将它配置到服务器进行运行，可以按照下面的代码进行配置web.xml文件：

```
<servlet>  
    <servlet-name>user</servlet-name>  
    <servlet-class>com.xinzhi.controller.UserServlet</servlet-class>  
</servlet>  
<servlet-mapping>  
    <servlet-name>user</servlet-name>  
    <url-pattern>/user.do</url-pattern>  
</servlet-mapping>
```

(2) 使用注解进行Servlet配置（高版本后默认使用此方法）：（学完位置再学注解）
当我们去创建一个Servlet时会默认继承HttpServlet类，会使用注解方式进行配置Servlet：

```
// 这种方式配置的效果与第一种一致
@WebServlet("/HelloServlet")
```

注意：两种配置方式不能同时使用，即配置了web.xml就不能使用注解，使用了注解也就不能使用web.xml配置了。

通过实现Servlet接口，这个接口定义了servlet的生命周期，所有的方法需要我们实现。

```
public class UserServlet implements Servlet {
    @Override
    public void init(ServletConfig servletConfig) throws ServletException {

    }

    @Override
    public ServletConfig getServletConfig() {
        return null;
    }

    @Override
    public void service(ServletRequest servletRequest, ServletResponse
servletResponse) throws ServletException, IOException {
        servletResponse.getWriter().print("<h1>hello servlet</h1>");
    }

    @Override
    public String getServletInfo() {
        return null;
    }

    @Override
    public void destroy() {

    }
}
```

GenericServlet

```
public abstract class GenericServlet implements Servlet, ServletConfig,
Serializable {
    private static final long serialVersionUID = 1L;
    private transient ServletConfig config;

    public GenericServlet() {
    }

    public void destroy() {
    }

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

```

    public void init() throws ServletException {
    }

    public abstract void service(ServletRequest var1, ServletResponse var2)
    throws ServletException, IOException;

    public String getServletName() {
        return this.config.getServletName();
    }

    ....
}

```

```

public class UserServicelet extends GenericServlet {
    @Override
    public void service(ServletRequest servletRequest, ServletResponse
    servletResponse) throws ServletException, IOException {
        servletResponse.getWriter().print("<h1>hello servlet</h1>");
    }
}

```

Http只是会根据请求的类型进行特殊的调用

```

//
// Source code recreated from a .class file by IntelliJ IDEA
// (powered by Fernflower decompiler)
//

package javax.servlet.http;

import java.io.IOException;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.text.MessageFormat;
import java.util.Enumeration;
import java.util.ResourceBundle;
import javax.servlet.DispatcherType;
import javax.servlet.GenericServlet;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public abstract class HttpServlet extends GenericServlet {

    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
        String protocol = req.getProtocol();
        String msg = IStrings.getString("http.method_get_not_supported");
        if (protocol.endsWith("1.1")) {
            resp.sendError(405, msg);
        } else {

```

```

        resp.sendError(400, msg);
    }

}

protected long getLastModified(HttpServletRequest req) {
    return -1L;
}

protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
    String protocol = req.getProtocol();
    String msg = IStrings.getString("http.method_post_not_supported");
    if (protocol.endsWith("1.1")) {
        resp.sendError(405, msg);
    } else {
        resp.sendError(400, msg);
    }
}

// 还是会调用它，只是会根据请求的类型进行特殊的调用
protected void service(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
    String method = req.getMethod();
    long lastModified;
    if (method.equals("GET")) {
        lastModified = this.getLastModified(req);
        if (lastModified == -1L) {
            this.doGet(req, resp);
        } else {
            long ifModifiedSince;
            try {
                ifModifiedSince = req.getDateHeader("If-Modified-Since");
            } catch (IllegalArgumentException var9) {
                ifModifiedSince = -1L;
            }

            if (ifModifiedSince < lastModified / 1000L * 1000L) {
                this.maybeSetLastModified(resp, lastModified);
                this.doGet(req, resp);
            } else {
                resp.setStatus(304);
            }
        }
    } else if (method.equals("HEAD")) {
        lastModified = this.getLastModified(req);
        this.maybeSetLastModified(resp, lastModified);
        this.doHead(req, resp);
    } else if (method.equals("POST")) {
        this.doPost(req, resp);
    } else if (method.equals("PUT")) {
        this.doPut(req, resp);
    } else if (method.equals("DELETE")) {
        this.doDelete(req, resp);
    } else if (method.equals("OPTIONS")) {

```

```

        this.doOptions(req, resp);
    } else if (method.equals("TRACE")) {
        this.doTrace(req, resp);
    } else {
        String errMsg = lStrings.getString("http.method_not_implemented");
        Object[] errArgs = new Object[]{method};
        errMsg = MessageFormat.format(errMsg, errArgs);
        resp.sendError(501, errMsg);
    }

}

...

}

```

HttpServletRequest和ServletRequest都是接口

HttpServletRequest继承自ServletRequest

HttpServletRequest比ServletRequest多了一些针对于Http协议的方法。例如：

getHeader(), getMethod(), getSession()

四、servlet的匹配规则

1、四种匹配规则

(1) 精确匹配

<url-pattern>中配置的项必须与url完全精确匹配。

```

<servlet-mapping>
    <servlet-name>MyServlet</servlet-name>
    <url-pattern>/user/users.html</url-pattern>
    <url-pattern>/index.html</url-pattern>
    <url-pattern>/user/addUser.action</url-pattern>
</servlet-mapping>

```

当在浏览器中输入如下几种url时，都会被匹配到该servlet

<http://localhost:8080/appDemo/user/users.html>

<http://localhost:8080/appDemo/index.html>

<http://localhost:8080/appDemo/user/addUser.action>

注意：

<http://localhost:8080/appDemo/user/addUser/> 是非法的url, 不会被当作<http://localhost:8080/appDemo/user/addUser>识别

另外上述url后面可以跟任意的查询条件, 都会被匹配, 如

<http://localhost:8080/appDemo/user/addUser?username=Tom&age=23> 会被匹配到MyServlet。

(2) 路径匹配

以“/”字符开头, 并以“/*”结尾的字符串用于路径匹配

```
<servlet-mapping>
  <servlet-name>MyServlet</servlet-name>
  <url-pattern>/user/*</url-pattern>
</servlet-mapping>
```

路径以/user/开始, 后面的路径可以任意。比如下面的url都会被匹配。

- <http://localhost:8080/appDemo/user/users.html>
- <http://localhost:8080/appDemo/user/addUser.action>
- <http://localhost:8080/appDemo/user/updateUser.action>

(3) 扩展名匹配**

以“*.”开头的字符串被用于扩展名匹配

```
<servlet-mapping>
  <servlet-name>MyServlet</servlet-name>
  <url-pattern>*.jsp</url-pattern>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

则任何扩展名为jsp或action的url请求都会匹配, 比如下面的url都会被匹配

- <http://localhost:8080/appDemo/user/users.jsp>
- <http://localhost:8080/appDemo/toHome.action>

(4) 缺省匹配

```
<servlet-mapping>
  <servlet-name>MyServlet</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

2、匹配顺序

1. 精确匹配。
2. 路径匹配, 先最长路径匹配, 再最短路径匹配。
3. 扩展名匹配。

注意：使用扩展名匹配，前面就不能有任何的路径。

4. 缺省匹配，以上都找不到servlet，就用默认的servlet，配置为<url-pattern>/

3、需要注意的问题

路径匹配和扩展名匹配无法同时设置

匹配方法只有三种，要么是路径匹配（以“/”字符开头，并以“/*”结尾），要么是扩展名匹配（以“*.”开头），要么是精确匹配，三种匹配方法不能进行组合，不要想当然使用通配符或正则规则。

- 如<url-pattern>/user/*.action</url-pattern>是非法的
- 另外注意：<url-pattern>/aa/*/bb</url-pattern>是精确匹配，合法，这里的*不是通配的含义

“/*”和“/”含义并不相同

- “/*”属于路径匹配，并且可以匹配所有request，由于路径匹配的优先级仅次于精确匹配，所以“/*”会覆盖所有的扩展名匹配，很多404错误均由此引起，所以这是一种特别恶劣的匹配模式。
- “/”是servlet中特殊的匹配模式，该模式有且仅有一个实例，优先级最低，不会覆盖其他任何url-pattern，只是会替换servlet容器的内建default servlet，该模式同样会匹配所有request。

Tomcat在%CATALINA_HOME%\conf\web.xml文件中配置了默认的Servlet，配置代码如下

```
<servlet>
    <servlet-name>default</servlet-name>
    <servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-
class>
    <init-param>
        <param-name>debug</param-name>
        <param-value>0</param-value>
    </init-param>
    <init-param>
        <param-name>listings</param-name>
        <param-value>>false</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet>
    <servlet-name>jsp</servlet-name>
    <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
    <init-param>
        <param-name>fork</param-name>
        <param-value>>false</param-value>
    </init-param>
    <init-param>
        <param-name>xpoweredBy</param-name>
        <param-value>>false</param-value>
    </init-param>
    <load-on-startup>3</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>/</url-pattern>
```

```

</servlet-mapping>

    <!-- The mappings for the JSP servlet -->
    <servlet-mapping>
        <servlet-name>jsp</servlet-name>
        <url-pattern>*.jsp</url-pattern>
        <url-pattern>*.jspx</url-pattern>
    </servlet-mapping>

```

- “/*”和“/”均会拦截静态资源的加载，需要特别注意

4、举例

映射的URL	对应的Servlet
/hello	servlet1
/bbs/admin/*	servlet2
/bbs/*	servlet3
*.jsp	servlet4
/	servlet5

实际请求映射的结果

去掉上下文路径的剩余路径	处理请求的Servlet
/hello	servlet1
/bbs/admin/login	servlet2
/bbs/admin/index.jsp	servlet2
/bbs/display	servlet3
/bbs/index.jsp	servlet3
/bbs	servlet3
/index.jsp	servlet4
/hello/index.jsp	servlet4
/hello/index.html	servlet5
/news	servlet5

五：请求和响应

1、请求-request

(1) request概述

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。（以后学习）

(2) 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。

(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

案例：request.getRemoteAddr(): 封IP

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();
if(ip.equals("127.0.0.1")) {
    response.getWriter().print("您的IP已被禁止！");
} else {
    response.getWriter().print("Hello!");
}
```

(4) 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，这取决于<form>的method属性值；

GET请求和POST请求的区别：

GET请求：

- Ø 请求参数会在浏览器的地址栏中显示，所以不安全；
- Ø 请求参数长度限制长度在1K之内；
- Ø 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：

- Ø 请求参数不会显示浏览器的地址栏，相对安全；
- Ø 请求参数长度没有限制；

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；

```

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);
}

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;

```

public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    String[] names = request.getParameterValues("name");
    System.out.println(Arrays.toString(names));
}

```

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

```

public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    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。

```

Map<String,String[]> paramMap = request.getParameterMap();
for(String name : paramMap.keySet()) {
    String[] values = paramMap.get(name);
    System.out.println(name + ": " + Arrays.toString(values));
}

```

(5) 请求转发

请求转发表示由 **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。

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");
        RequestDispatcher rd = request.getRequestDispatcher("/BServlet");
        rd.forward(request, response);
    }
}
```

```
public class BServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        System.out.println("BServlet");
    }
}
```

(6) request域方法

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。

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，
- 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；

域方法通常在进行重定向时使用，多个servlet共享数据。

2、响应-response

(1) response概述

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;

(2) response响应正文

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

- `PrintWriter out = response.getWriter();` 获取字符流，处理字符;
- `ServletOutputStream out = response.getOutputStream();` 获取字节流，处理文件;

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`response.getWriter()`，要么使用`response.getOutputStream()`，但不能同时使用这两个流。不然会抛出`IllegalStateException`异常。

字符响应流

(1) 字符编码

重要：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.setCharacterEncoding("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.setCharacterEncoding("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：

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

(3) 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.setCharacterEncoding("utf-8");`

- `response.setHeader("Refresh","5; URL=http://www.baidu.cn");` 5秒后自动跳转到百度主页。

(4) 设置状态码及其他方法

- `response.setContentType("text/html;charset=utf-8")`: 等同与调用 `response.setHeader("content-type", "text/html;charset=utf-8")`; 用它就行了。
- `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。

(5) 重定向, 重要

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。

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 {
        response.setStatus(302);
        response.setHeader("Location", "http://www.baidu.com");
    }
}
```

上面代码的作用是: 当访问AServlet后, 会通知浏览器重定向到百度主页。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, 然后发出第二个请求。

便捷的重定向方

```
public class AServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.sendRedirect("http://www.baidu.com");
    }
}
```

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>。

5.4 重定向小结

- 重定向是两次请求，请求转发是一次
- 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；

3、重定向和转发的区别

- 重定向是两次请求，转发是一个请求
- **重定向是浏览器的行为，请求转发是服务器行为**
- **重定向浏览器的地址会发生变化，转发不会**
- 重定向可以重定向到任何地址，转发只能在项目内转发

4、session和cookie

http是无状态的，他不保存状态，意思就是一个浏览器发的请求，随后就断开了，下一次发送请求就和上一次无关了。

比如一个用户购买一个商品，第一次需要登录，如果再买一个时向服务器发送请求，服务器如果不知道是谁发的，那么他就得再登录一次，这显然是不合理的，于是就提出了cookie和session的概念。

cookie是记录在浏览器端的一个字符串，session是保存在服务器端的一个对象。他们两互相配合让服务器有了能识别客户端一些状态的能力，意思就是服务就能知道这个客户端有没有登录等。cookie就相当于通行证，session就是门房，进去时需要从门房识别一个身份。

创建时机

1. 当浏览器向客户端发送请求时，服务器会为他创建一个session，同时相应会加一个头（Set-Cookie: jsessionid=ewrwerwer123）
2. 浏览器得到相应就会在自己这保存下这个字符串，以后访问这个网站的时候就会一直带着。
3. 当下一个请求发起时，会带着这个cookie的信息，服务器通过查询id找的session，通过session内保存的信息，就能获得这个客户端的状态。

第三章 jsp入门学习

一、JSP基础语法

1、JSP模板元素

JSP页面中的HTML内容称之为JSP模版元素。

JSP模版元素定义了网页的基本骨架，即定义了页面的结构和外观。

2、JSP脚本片段

JSP脚本片段用于在JSP页面中编写多行Java代码（在<%>不能定义方法）。语法：<%多行java代码 %>

例如：

```
<%  
    int num = 0;  
    num = ++num;  
    out.println("num:" + num);  
%>
```

注意：

- 1、JSP脚本片段中只能出现java代码，不能出现其它模板元素，JSP引擎在翻译JSP页面中，会将JSP脚本片段中的Java代码将被原封不动地放到Servlet的_jspService方法中。
- 2、JSP脚本片段中的Java代码必须严格遵循Java语法，例如，每执行语句后面必须用分号（;）结束。
- 3、在一个JSP页面中可以有多脚本片段，在两个或多个脚本片段之间可以嵌入文本、HTML标记和其他JSP元素。
- 4、多个脚本片段中的代码可以相互访问

3、JSP表达式

JSP脚本表达式（expression）用于将程序数据输出到客户端，语法：<%=变量或表达式 %>

例如：

```
<%= "123" %>
```

4、JSP声明

JSP页面中编写的所有代码，默认会翻译到servlet的service方法中，而JSP声明中的Java代码被翻译到_jspService方法的外面。语法：<%! java代码 %>

JSP声明可用于定义JSP页面转换成的Servlet程序的静态代码块、成员变量和方法。

例如：

```
<%!  
static {  
    System.out.println("静态代码块");  
}  
  
private String name = "XinZhi";  
  
public void TestFun(){  
    System.out.println("成员方法！");  
}  
%>  
<%  
    TestFun();  
    out.println("name:" + name);  
%>
```

5、JSP注释

在JSP中，注释有显式注释，隐式注释，JSP自己的注释：

显式注释	直接使用HTML风格的注释：<!-- 注释内容 -->
隐式注释	直接使用JAVA的注释：//、/...../
JSP自己的注释	<%- - 注释内容 - -%>

区别：

HTML的注释在浏览器中查看源文件的时候是可以看得到的，而JAVA注释和JSP注释在浏览器中查看源文件时是看不到注释的内容的。

二、JSP原理

1、jsp本质上是是什么

浏览器向服务器发请求，不管访问的是什么资源，其实都是在访问Servlet，所以当访问一个jsp页面时，其实也是在访问一个Servlet，服务器在执行jsp的时候，首先把jsp编译成一个Servlet，所以我们访问jsp时，其实不是在访问jsp，而是在访问jsp翻译过后的那个Servlet。

所以jsp的本质其实就是个html模板，编译器会根据模板生成对应的servlet。

例如下面的代码：

```
<%@ page language="java" contentType="text/html; charset=UTF-8"  
    pageEncoding="UTF-8"%>  
<%  
    String path = request.getContextPath();  
    String basePath =  
request.getScheme()+"://"+request.getServerName()+":"+request.getServerPort()+pa  
th+"/";  
%>
```

```

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<base href="<%=basePath%>">
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
    <%!
        static {
            System.out.println("静态代码块");
        }

        private String name = "XinZhi";

        public void TestFun(){
            System.out.println("成员方法! ");
        }
    %>
    <%
        TestFun();
        out.println("name:" + name);
    %>
</body>
</html>

```

当我们通过浏览器访问index.jsp时，服务器首先将index.jsp翻译成一个index_jsp.class，在Tomcat服务器的work\Catalina\localhost\项目名\org\apache\jsp目录下可以看到index_jsp.class的源代码文件index_jsp.java

```

package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class index_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {

    static {
        System.out.println("静态代码块");
    }

    private String name = "XinZhi";

    public void TestFun(){
        System.out.println("成员方法! ");
    }

    private static final javax.servlet.jsp.JspFactory _jspxFactory =
        javax.servlet.jsp.JspFactory.getDefaultFactory();

```

```

    private static java.util.Map<java.lang.String,java.lang.Long>
    _jspx_dependants;

    private volatile javax.el.ExpressionFactory _el_expressionfactory;
    private volatile org.apache.tomcat.InstanceManager _jsp_instancemanager;

    public java.util.Map<java.lang.String,java.lang.Long> getDependants() {
        return _jspx_dependants;
    }

    public javax.el.ExpressionFactory _jsp_getExpressionFactory() {
        if (_el_expressionfactory == null) {
            synchronized (this) {
                if (_el_expressionfactory == null) {
                    _el_expressionfactory =
_jspFactory.getJspApplicationContext(getServletConfig().getServletContext()).ge
tExpressionFactory();
                }
            }
        }
        return _el_expressionfactory;
    }

    public org.apache.tomcat.InstanceManager _jsp_getInstanceManager() {
        if (_jsp_instancemanager == null) {
            synchronized (this) {
                if (_jsp_instancemanager == null) {
                    _jsp_instancemanager =
org.apache.jasper.runtime.InstanceManagerFactory.getInstanceManager(getServletCo
nfig());
                }
            }
        }
        return _jsp_instancemanager;
    }

    public void _jspInit() {
    }

    public void _jspDestroy() {
    }

    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;

out.write("\r\n");
out.write("<!DOCTYPE html PUBLIC \"-//W3C//DTD HTML 4.01
Transitional//EN\" \"http://www.w3.org/TR/html4/loose.dtd\">\r\n");
out.write("<html>\r\n");
out.write("<head>\r\n");
out.write("<meta http-equiv=\"Content-Type\" content=\"text/html;
charset=UTF-8\">\r\n");
out.write("<title>Insert title here</title>\r\n");
out.write("</head>\r\n");
out.write("<body>\r\n");
out.write("\t");
out.write('\r');
out.write('\n');
out.write(' ');

TestFun();
out.println("name:" + name);

out.write("\r\n");
out.write("</body>\r\n");
out.write("</html>");
} catch (java.lang.Throwable t) {
    if (!(t instanceof javax.servlet.jsp.SkipPageException)){
        out = _jspx_out;
        if (out != null && out.getBufferSize() != 0)
            try {
                if (response.isCommitted()) {
                    out.flush();
                } else {
                    out.clearBuffer();
                }
            } catch (java.io.IOException e) {}
        if (_jspx_page_context != null)
            _jspx_page_context.handlePageException(t);
        else throw new ServletException(t);
    }
} finally {
    _jspxFactory.releasePageContext(_jspx_page_context);
}
}
}

```

index_jsp这个类是继承org.apache.jasper.runtime.HttpJspBase这个类的，通过查看HttpJspBase源代码，可以知道HttpJspBase类是继承HttpServlet的，所以HttpJspBase类是一个Servlet，而index_jsp又是继承HttpJspBase类的，所以index_jsp类也是一个Servlet，所以当浏览器访问服务器上的index.jsp页面时，其实就是在访问index_jsp这个Servlet，index_jsp这个Servlet使用_jspService这个方法处理请求。

HttpJspBase源码如下:

```
import java.io.IOException;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.jsp.HttpJspPage;
import javax.servlet.jsp.JspFactory;

import org.apache.jasper.compiler.Localizer;

public abstract class HttpJspBase extends HttpServlet implements HttpJspPage{

    protected HttpJspBase() {
    }

    public final void init(ServletConfig config)
    throws ServletException
    {
        super.init(config);
        jspInit();
        _jspInit();
    }

    public String getServletInfo() {
        return Localizer.getMessage("jsp.engine.info");
    }

    public final void destroy() {
        jspDestroy();
        _jspDestroy();
    }

    /**
     * Entry point into service.
     */
    public final void service(HttpServletRequest request, HttpServletResponse
response)
    throws ServletException, IOException
    {
        _jspService(request, response);
    }

    public void jspInit() {
    }

    public void _jspInit() {
    }

    public void jspDestroy() {
    }
}
```



```
protected void _jspDestroy() {  
}  
  
public abstract void _jspService(HttpServletRequest request,  
                                HttpServletResponse response)  
    throws ServletException, IOException;  
}
```

2、_jspService方法

问题1: Jsp页面中的html排版标签是如何被发送到客户端的?

浏览器接收到的这些数据, 都是在_jspService方法中使用如下的代码输出给浏览器的。

问题2: Jsp页面中的java代码服务器是如何执行的?

在jsp中编写的java代码会被翻译到_jspService方法中去, 当执行_jspService方法处理请求时, 就会执行在jsp编写的java代码了, 所以Jsp页面中的java代码服务器是通过调用_jspService方法处理请求时执行的。

3、jsp在服务器的执行流程

第一次执行:

1. 客户端通过电脑连接服务器, 因为是请求是动态的, 所以所有的请求交给WEB容器来处理
2. 在容器中找到需要执行的*.jsp文件
3. 之后*.jsp文件通过转换变为*.java文件
4. *.java文件经过编译后, 形成*.class文件
5. 最终服务器要执行形成的*.class文件

第二次执行:

1. 因为已经存在了*.class文件, 所以不在需要转换和编译的过程

修改后执行:

1. 源文件已经被修改过了, 所以需要重新转换, 重新编译。

三、JSP指令

1、JSP指令标识的语法格式

```
<%@ 指令名 属性1 = "属性1的值" 属性2 = "属性2的值" ....%>
```

- 指令名:用于指定指令名称 在JSP中包含page include taglib 这3种指令
- 属性: 用于指定指令属性名称 不同的指令包含不同的属性 在同一个指令中可以设置多个属性 各个属性之间用逗号或者空格隔开
- 属性值:用于指定属性的值

注意点:

指令标识`<%@`是一个完整的指令,不能够添加空格,但是便签中定义的属性与指令名之间是有空格的

2、Page指令

page指令是JSP页面中最常见的指令,用于定义整个JSP页面的相关属性

语法格式

```
<%@ page 属性1 = "属性1的值" 属性2 = "属性2的值" ....%>
```

page指令的相关属

language属性

用于设置整个JSP页面的使用的语言,目前只支持JAVA语言,改属性默认值是JAVA

```
<%@ page language="java" %>
```

improt属性

设置JSP导入的类包

```
<%@ page import="java.util.*" %>
```

pageEccoding属性

这种JSP页面的编码格式,也就是指定文件编码

设置JSP页面的MIME类型和字符编码

```
<%@ page contentType ="text/html; charset=UTF-8" %>
```

Sessions属性

设置页面是否使用HTTP的session会话对象.Boolean类型,默认值是true

```
<%@ page session ="false" %>
```

- session是JSP的内置对象之一

autoFlush属性

设置JSP页面缓存满时,是否自动刷新缓存,默认值是:true, 如果这种为false,则当页面缓存满是就会抛出异常

```
<%@ page autoFlush ="false" %>
```

isErrorPage属性

可以把当前页面设置成错误处理页面来处理另外jsp页面的错误

```
<%@ page isErrorPage ="true" %>
```

errorPage属性

指定当前jsp页面异常错误的另一个JSP页面,指定的JSP页面的isErrorPage属性必须为true,属性值是一个url字符串

```
<%@ page errorPage = "errorPage.jsp" %>
```

3、include指令

include指令用于引入其它JSP页面，如果使用include指令引入了其它JSP页面，那么JSP引擎将把这两个JSP翻译成一个servlet。所以include指令引入通常也称之为静态引入。

语法：<%@ include file="relativeURL"%>

file属性用于指定被引入文件的路径。路径以"/"开头，表示代表当前web应用。

注意细节：

1. 被引入的文件必须遵循JSP语法。
2. 被引入的文件可以使用任意的扩展名，即使其扩展名是html，JSP引擎也会按照处理jsp页面的方式处理它里面的内容，为了见明知意，JSP规范建议使用.jspf (JSP fragments(片段)) 作为静态引入文件的扩展名。
3. 由于使用include指令将会涉及到2个JSP页面，**并会把2个JSP翻译成一个servlet**，所以这2个JSP页面的指令不能冲突（除了pageEncoding和导包除外）。

四、JSP标签

一、Jsp标签分类

- 1) 内置标签（动作标签）：不需要在jsp页面导入标签
- 2) jstl标签：需要在jsp页面中导入标签
- 3) 自定义标签：开发者自行定义，需要在jsp页面导入标签

JSP标签也称之为Jsp Action(JSP动作)元素，它用于在Jsp页面中提供业务逻辑功能，避免在JSP页面中直接编写java代码，造成jsp页面难以维护。

常用内置标签有以下三个：

1、标签一<jsp:include>

<jsp:include>标签用于把另外一个资源的输出内容插入进当前JSP页面的输出内容之中，这种在JSP页面执行时的引入方式称之为动态引入。

语法：``

page	用于指定被引入资源的相对路径，它也可以通过执行一个表达式来获得。
flush	指定在插入其他资源的输出内容时，是否先将当前JSP页面的已输出的内容刷新到客户端。

标签与include指令的区别：

<jsp:include>标签是动态引入，<jsp:include>标签涉及到的2个JSP页面会被翻译成2个servlet，这2个servlet的内容在执行时进行合并。而include指令是静态引入，涉及到的2个JSP页面会被翻译成一个servlet，其内容是在源文件级别进行合并。

2、标签<jsp:forward>和<jsp:param>

<jsp:forward>标签用于把给另外一个资源（服务器跳转，地址不变）。

```
<%--使用jsp:forward标签进行请求转发--%>
<jsp:forward page="/index2.jsp" >
    <jsp:param value="10086" name="num"/>
    <jsp:param value="10010" name="num2"/>
</jsp:forward>
```

五、九大内置对象

1. request 对象

代表的是来自客户端的请求，客户端发送的请求封装在 request 对象中，通过它才能了解到用户的请求信息，然后作出响应，它是 HttpServletRequest 的实例，作用域为 request（响应生成之前）

常用方法：

```
Object getAttribute(String name); // 返回指定属性的属性值
void setAttribute(String key, Object value); // 设置属性的属性值
Enumeration getAttributeNames(); // 返回所有可以用属性名的枚举
String getParameter(String name); // 返回指定name的参数值
Enumeration getParameterNames(); // 返回可用参数名的枚举
String[] getParameterValues(String name); // 返回包含参数name的所有值的数组
ServletInputStream getInputStream(); // 得到请求体中一行的二进制流
BufferedReader getReader(); // 返回解码过了的请求体

String getServerName(); // 返回接收请求的服务器主机名
int getServerPort(); // 返回服务器接收此请求所用的端口号
String getRemoteAddr(); // 返回发送请求的客户端的IP地址
String getRemoteHost(); // 返回发送请求的客户端主机名
String getRealPath(); // 返回一个虚拟路径的真实路径
String getCharacterEncoding(); // 返回字符编码方式
int getContentLength(); // 返回请求体的长度（字节数）
String getContentType(); // 返回请求体的MIME类型
String getProtocol(); // 返回请求用的协议类型以及版本号
String getScheme(); // 返回请求用的协议名称（例如：http https ftp）
```

2. response 对象

对象代表的是对客户端的响应，也就是说可以通过 response 对象来组织发送到客户端的数据；但是由于组织方式比较底层，所以不建议初学者使用，需要向客户端发送文字时直接使用；它是 HttpServletResponse 的实例；作用域为 page（页面执行期）

常用方法：

```
String getCharacterEncoding();// 返回响应应用的是哪种字符编码
ServletOutputStream getOutputStream();// 返回响应的一个二进制输出流
PrintWriter getWriter();// 返回可以向客户端输出字符的一个对象
void setContentLength(int len);// 设置响应头长度
void setContentType(String type);// 设置响应的MIME类型
void sendRedirect(String location);// 重新定向客户端的请求
```

3. session 对象

指的是客户端与服务器的一次会话,从客户连接到服务器的一个 WebApplication 开始,直到客户端与服务器断开连接为止;它是 HttpSession 类的实例,作用域为 session (会话期)

常用方法:

```
long getCreationTime();// 返回SESSION创建时间
public String getId();// 返回SESSION创建时JSP引擎为它设的惟一ID号
long getLastAccessedTime();// 返回此SESSION里客户端最近一次请求时间
int getMaxInactiveInterval();// 返回两次请求间隔多长时间此SESSION被取消(ms)
String[] getValueNames();// 返回一个包含此SESSION中所有可用属性的数组
void invalidate();// 取消SESSION,使SESSION不可用
boolean isNew();// 返回服务器创建的一个SESSION,客户端是否已经加入
void removeValue(String name);// 删除SESSION中指定的属性
void setMaxInactiveInterval();// 设置两次请求间隔多长时间此SESSION被取消(ms)
```

4. out 对象

out 对象是 JspWriter 类的实例,是向客户端输出内容常用的对象;作用域为 page (页面执行期)

常用方法:

```
void clear();// 清除缓冲区的内容
void clearBuffer();// 清除缓冲区的当前内容
void flush();// 清空流
int getBufferSize();// 返回缓冲区以字节数的大小,如不设缓冲区则为0
int getRemaining();// 返回缓冲区还剩余多少可用
boolean isAutoFlush();// 返回缓冲区满时,是自动清空还是抛出异常
void close();// 关闭输出流
```

5. page 对象

page 对象就是指向当前 JSP 页面本身,有点象类中的 this 指针,它是 Object 类的实例;page 对象代表了正在运行的由 JSP 文件产生的类对象,不建议初学者使用;作用域为 page (页面执行期)

常用方法:

```

class getClass();// 返回此Object的类
int hashCode();// 返回此Object的hash码
boolean equals(Object obj);// 判断此Object是否与指定的Object对象相等
void copy(Object obj);// 把此Object拷贝到指定的Object对象中
Object clone();// 克隆此Object对象
String toString();// 把此Object对象转换成String类的对象
void notify();// 唤醒一个等待的线程
void notifyAll();// 唤醒所有等待的线程
void wait(int timeout);// 使一个线程处于等待直到timeout结束或被唤醒
void wait();// 使一个线程处于等待直到被唤醒
void enterMonitor();// 对Object加锁
void exitMonitor();// 对Object开锁

```

6. application 对象

实现了用户间数据的共享，可存放全局变量；它始于服务器的启动，直到服务器的关闭，在此期间，此对象将一直存在；这样在用户的前后连接或不同用户之间的连接中，可以对此对象的同一属性进行操作；在任何地方对此对象属性的操作，都将影响到其他用户对此的访问；服务器的启动和关闭决定了 application 对象的生命；它是 ServletContext 类的实例；作用域为 application

常用方法：

```

Object getAttribute(String name);// 返回给定名的属性值
Enumeration getAttributeNames();// 返回所有可用属性名的枚举
void setAttribute(String name,Object obj);// 设定属性的属性值
void removeAttribute(String name);// 删除一属性及其属性值
String getServerInfo();// 返回JSP(SERVLET)引擎名及版本号
String getRealPath(String path);// 返回一虚拟路径的真实路径
ServletContext getContext(String uripath);// 返回指定webApplication的application对象
int getMajorVersion();// 返回服务器支持的Servlet API的最大版本号
int getMinorVersion();// 返回服务器支持的Servlet API的最大版本号
String getMimeType(String file);// 返回指定文件的MIME类型
URL getResource(String path);// 返回指定资源(文件及目录)的URL路径
InputStream getResourceAsStream(String path);// 返回指定资源的输入流
RequestDispatcher getRequestDispatcher(String uripath);// 返回指定资源的RequestDispatcher对象
Servlet getServlet(String name);// 返回指定名的Servlet
Enumeration getServlets();// 返回所有Servlet的枚举
Enumeration getServletNames();// 返回所有Servlet名的枚举
void log(String msg);// 把指定消息写入Servlet的日志文件
void log(Exception exception,String msg);// 把指定异常的栈轨迹及错误消息写入Servlet的日志文件
void log(String msg,Throwable throwable);// 把栈轨迹及给出的Throwable异常的说明信息写入Servlet的日志文件

```

7. pageContext 对象

提供了对 JSP 页面内所有的对象及名字空间的访问，也就是说他可以访问到本页所在的 session，也可以取本页面所在的 application 的某一属性值，他相当于页面中所有功能的集大成者，它的本类名也叫 pageContext；作用域为 Pageconfig 对象

常用方法:

```
JspWriter getOut();// 返回当前客户端响应被使用的JspWriter流(out)
HttpSession getSession();// 返回当前页中的HttpSession对象(session)
Object getPage();// 返回当前页的Object对象(page)
ServletRequest getRequest();// 返回当前页的ServletRequest对象(request)
ServletResponse getResponse();// 返回当前页的ServletResponse对象(response)
Exception getException();// 返回当前页的Exception对象(exception)
ServletConfig getServletConfig();// 返回当前页的ServletConfig对象(config)
ServletContext getServletContext();// 返回当前页的ServletContext对象(application)
void setAttribute(String name,Object attribute);// 设置属性及属性值
void setAttribute(String name,Object obj,int scope);// 在指定范围内设置属性及属性值
public Object getAttribute(String name);// 取属性的值
Object getAttribute(String name,int scope);// 在指定范围内取属性的值
public Object findAttribute(String name);// 寻找一属性,返回起属性值或NULL
void removeAttribute(String name);// 删除某属性
void removeAttribute(String name,int scope);// 在指定范围删除某属性
int getAttributeScope(String name);// 返回某属性的作用范围
Enumeration getAttributeNamesInScope(int scope);// 返回指定范围内可用的属性名枚举
void release();// 释放pageContext所占用的资源
void forward(String relativeUrlPath);// 使当前页面重导到另一页面
void include(String relativeUrlPath);// 在当前位置包含另一文件
```

8. config 对象

config 对象是在一个 Servlet 初始化时, JSP 引擎向它传递信息用的, 此信息包括 Servlet 初始化时所要用的参数 (通过属性名和属性值构成) 以及服务器的有关信息 (通过传递一个 ServletContext 对象); 作用域为 page

常用方法:

```
ServletContext getServletContext();// 返回含有服务器相关信息的ServletContext对象
String getInitParameter(String name);// 返回初始化参数的值
Enumeration getInitParameterNames();// 返回Servlet初始化所需所有参数的枚举
```

9. exception 对象

这是一个例外对象, 当一个页面在运行过程中发生了例外, 就产生这个对象; 如果一个JSP页面要应用此对象, 就必须把 isErrorPage 设为true, 否则无法编译; 他实际上是 Throwable 的对象; 作用域为 page

常用方法:

```
String getMessage();// 返回描述异常的消息
String toString();// 返回关于异常的简短描述消息
void printStackTrace();// 显示异常及其栈轨迹
Throwable fillInStackTrace();// 重写异常的执行栈轨迹
```

10、总结

对象名	描述	类型	作用域
request	请求对象	javax.servlet.HttpServletRequest	Request
response	响应对象	javax.servlet.SrvletResponse	Page
pageContext	页面上下文对象	javax.servlet.jsp.PageContext	Page
session	会话对象	javax.servlet.http.HttpSession	Session
application	应用程序对象	javax.servlet.ServletContext	Application
out	输出对象	javax.servlet.jsp.JspWriter	Page
config	配置对象	javax.servlet.ServletConfig	Page
page	页面对象	javax.lang.Object	Page
exception	例外对象	javax.lang.Throwable	Page

六、JSP属性作用域

JSP中提供了四种属性范围（四大域对象），如下：

1. **当前页（pageContext）**：一个属性只能在一个页面中取得，跳转到其他页面无法取得
2. **一次服务器请求（request）**：一个页面中设置的属性，只要经过了请求重定向之后的页面可以继续取得。
3. **一次会话（session）**：一个用户设置的内容，只要是与此用户相关的页面都可以访问（一个会话表示一个人，这个人设置的东西只要这个人不走，就依然有效），关了浏览器就不见了。
4. **上下文中（application）**：在整个服务器上设置的属性，所有人都可以访问

七、静态资源的路径问题

在jsp中调用图片、JS脚本等，针对取得的路径有两种调用方式：

- 1、放入Body中生成绝对路径（不建议）

```
<%  
    String path = request.getContextPath();  
    String basePath =  
request.getScheme()+"://"+request.getServerName()+":"+request.getServerPort()+pa  
th+"/";  
    %>
```



```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <title>image调用</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
  <h1>图片访问</h1>
  <div>
    
  </div>
</body>
</html>
```

2、在Head中指定，Body中使用相对路径（建议）

```
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>
<%
    String path = request.getContextPath();
    String basePath =
request.getScheme()+"://"+request.getServerName()+":"+request.getServerPort()+pa
th+"/";
%>
<html>
<head>
  <title>image调用</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <base href="<%=basePath%>">
</head>
<body>
  <h1>图片访问</h1>
  <div>
    
  </div>
</body>
</html>
```

八、错误页面和404页面

```
<error-page>
  <error-code>404</error-code>
  <location>/pages/404.jsp</location>
</error-page>

<error-page>
  <exception-type>java.lang.Exception</exception-type>
  <location>/pages/err.jsp</location>
</error-page>
<welcome-file-list>
  <welcome-file>/pages/home.jsp</welcome-file>
</welcome-file-list>
```

图片居中的方式：

可以绝对定位、浮动等手段都可以，这里使用一下弹性容器flex

```
<div style="display: flex;justify-content: center;align-content: center;width: 100%;height: 100%">

</div>
```

第四章 EL表达式和JSTL标签库

一、EL表达式

1、特点

- (1) 是一个由java开发的工具包
- (2) 用于从特定域对象中读取并写入到响应体开发任务，不能向域对象中写入。
- (3) EL工具包自动存在Tomcat的lib中（el-api.jar），开发是可以直接使用，无需其他额外的包。
- (4) 标准格式：\${域对象别名.关键字} 到指定的域中获取相应关键字的内容，并将其写入到响应体。

2、域对象

jsp	el	描述
application	applicationScope	全局作用域对象
session	sessionScope	会话作用域
request	requestScope	请求作用域对象
pageContext	pageScope	当前页作用域对象

注：使用时可以省略域对象别名

默认查找顺序：pageScope -> requestScope -> sessionScope -> applicationScope

最好只在pageScope中省略

注：对应案例

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
```

```

<head>
    <title>jsp</title>
</head>
<body>
    <%
        application.setAttribute("name","application");
        session.setAttribute("name","session");
        request.setAttribute("name","request");
        pageContext.setAttribute("name","pageContext");
    %>
    <br>-----使用java语言-----<br>
    application中的值: <%= application.getAttribute("name") %> <br>
    session中的值: <%= session.getAttribute("name") %> <br>
    request中的值: <%= request.getAttribute("name") %> <br>
    pageContext中的值: <%= pageContext.getAttribute("name") %> <br>

    <br>-----使用EL表达式-----<br>
    application中的值: ${applicationScope.name} <br>
    session中的值: ${sessionScope.name} <br>
    request中的值: ${requestScope.name} <br>
    pageContext中的值: ${pageScope.name} <br>

    <br>-----使用EL表达式,省略域对象-----<br>
    application中的值: ${name} <br>

</body>
</html>

```

3、支持的运算

- (1) 数学运算
- (2) 比较运算 > gt < lt >= ge <= le == eq != !=
- (3) 逻辑运算 && || !

注: 对应案例

```

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>EL运算</title>
</head>
<body>
    <%
        request.setAttribute("num1","12");
        request.setAttribute("num2","14");

        application.setAttribute("flag1",true);
        application.setAttribute("flag2",false);
    %>
    <br>-----使用java语言-----<br>
    <%
        String num1 = (String)request.getAttribute("num1");
        String num2 = (String)request.getAttribute("num2");
        int num3 = Integer.parseInt(num1) + Integer.parseInt(num2);
    %>

```

```

boolean flag1 = (Boolean) application.getAttribute("flag1");
boolean flag2 = (Boolean) application.getAttribute("flag2");
boolean flag3 = flag1 && flag2;
//输出方式一
out.write(Boolean.toString(flag3));

%>
<!-- 输出方式二 -->
<h1><%=num3%></h1>

<br>-----使用EL表达式-----<br>
<h1>${ requestScope.num1 + requestScope.num2 }</h1>
<h1>${ requestScope.num1 > requestScope.num2 }</h1>
<h1>${ applicationScope.flag1 && applicationScope.flag2 }</h1>

</body>
</html>

```

4、其他的内置对象

- (1) param 使用 \${param.请求参数名} 从请求参数中获取参数内容
- (2) paramValues 使用 \${ paramValues.请求参数名 } 从请求参数中获取多个值，以数组的形式
- (3) initParam 使用 \${ initParam.参数名 } 获取初始化参数

注：对应案例

```

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>其他内置对象</title>
</head>
<body>
    url: ...?username=zhangsan&password=admin <br>
    url中的参数: 用户名: ${param.username} ---- 密码: ${param.password} <br>
    -----
    <br>
    url: ...?username=zhangsan&username=lisi <br>
    url中的参数: 用户1: ${paramValues.username[0]} -- 用户2:
    ${paramValues.username[1]} <br>
    -----
    <br>
    <!--
    在web.xml中添加启动参数
    <context-param>
        <param-name>driver</param-name>
        <param-value>com.mysql.jdbc.Driver</param-value>
    </context-param>
    -->
    获取启动参数: ${initParam.driver}
</body>
</html>

```

5、EL表达式的缺陷

- (1) 只能读取域对象中的值，不能写入
- (2) 不支持if判断和控制语句

二、JSTL标签工具类

1、基本介绍

- (1) JSP Standard Tag Lib jsp标准标签库
- (2) 是sun公司提供
- (3) 组成

核心标签	对java在jsp上基本功能进行封装，如 if while等	主要学习
sql标签	JDBC在jsp上的使用	
xml标签	Dom4j在jsp上的使用	
format标签	jsp文件格式转换	

- (4) 使用原因：使用简单，且在JSP编程当中要求尽量不出现java代码。

2、使用方式

- (1) 导入依赖的jar包 jstl.jar standard.jar
- (2) 在jsp中引入JSTL的core包依赖约束

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

3、重要标签的使用

(1) <c:set>

在JSP文件上设置域对象中的共享数据

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<html>
<head>
    <title> c:set </title>
</head>
<body>
    <!--
    相当于
    <%-- <% --%>
    <%-- request.setAttribute("name","zhangsang");--%>
    <%-- %> --%>
    -->
    <c:set scope="request" var="name" value="zhangsang" />
    通过JSTL标签添加的作用域中的值: ${requestScope.name} <br>
    <c:set scope="application" var="name" value="lisi" />
```

```

        通过JSTL标签添加的作用域中的值: ${applicationScope.name}    <br>
        <c:set scope="request" var="name" value="wangwu" />
        通过JSTL标签添加的作用域中的值: ${requestScope.name}    <br>
        <c:set scope="page" var="name" value="zhaoliu" />
        通过JSTL标签添加的作用域中的值: ${pageScope.name}    <br>
    </body>
</html>

```

(2) <c:if>

控制哪些内容能够输出到响应体

```

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
    <title> c:if </title>
</head>
<body>
    <c:set scope="page" var="age" value="20"/>
    <br>-----使用java语言-----
    -----<br>
    <%
        if( Integer.parseInt((String)pageContext.getAttribute("age")) >= 18 ){
    %>
    输入: 欢迎光临!
    <% } else { %>
    输入: 未满十八, 不准入内!
    <% } %>
    <br>-----使用JSTL标签-----
    -----<br>

    <c:if test="${ age ge 18 }">
        输入: 欢迎光临!
    </c:if>
    <c:if test="${ age lt 18 }">
        输入: 未满十八, 不准入内!
    </c:if>
</body>
</html>

```

(3) <c:choose>

在jsp中进行多分支判断, 决定哪个内容写入响应体

```

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
    <title> c:choose </title>
</head>
<body>
    <c:set scope="page" var="age" value="6"/>

```

```

<br>-----使用java语言-----
-----<br>
<%
    if( Integer.parseInt((String)pageContext.getAttribute("age")) == 18 ){
    %>
    输入：您今年成年了
    <% } else if( Integer.parseInt((String)pageContext.getAttribute("age")) >
18 ){ %>
    输入：您已经成年了
    <% } else { %>
    输出：您还是个孩子
    <% } %>
<br>-----使用JSTL标签-----
-----<br>

<c:choose>
    <c:when test="${age eq 18}">
        输入：您今年成年了
    </c:when>
    <c:when test="${age gt 18}">
        输入：您已经成年了
    </c:when>
    <c:otherwise>
        输入：您还是个孩子
    </c:otherwise>
</c:choose>
</body>
</html>

```

(3) <c:forEach>

循环遍历

使用方式

```

<c:forEach var="申明循环变量的名称" begin="初始化循环变量"
            end="循环变量可以接受的最大值" step="循环变量的递增或递减值">
    *** step属性可以不写，默认递增1
    *** 循环变量默认保存在pageContext中
</c:forEach>

```

例子

```

<%@ page import="com.zn.Student" %>
<%@ page import="java.util.List" %>
<%@ page import="java.util.ArrayList" %>
<%@ page import="java.util.Map" %>
<%@ page import="java.util.HashMap" %>
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
    <title> c:forEach </title>

```

```

</head>
<body>
    <%
        pageContext.setAttribute("students",new ArrayList(){
            add(new Student("s1","zhangsan",16));
            add(new Student("s2","lisi",19));
            add(new Student("s3","wangwu",15));
        });
        pageContext.setAttribute("stuMap", new HashMap(){
            put("m1",new Student("s1","zhangsan",16));
            put("m2",new Student("s2","lisi",18));
            put("m3",new Student("s3","wangwu",15));
        });
    %>
    <br>-----使用java语言-----<br>
    <table>
        <tr><td>学号</td><td>姓名</td><td>年龄</td></tr>
        <%
            List<Student> stus =
(ArrayList<Student>)pageContext.getAttribute("students");
            for (int i = 0; i < stus.size(); i++) {
                <tr><td><%=stus.get(i).getSid()%></td>
                    <td><%=stus.get(i).getName()%></td>
                    <td><%=stus.get(i).getAge()%></td>
                </tr>
            <% } %>
        </table>

    <br>-----使用JSTL标签读取list-----<br>
    <table>
        <tr><td>学号</td><td>姓名</td><td>年龄</td></tr>
        <c:forEach var="student" items="${students}">
            <tr><td>${student.sid}</td>
                <td>${student.name}</td>
                <td>${student.age}</td>
            </tr>
        </c:forEach>
    </table>

    <br>-----使用JSTL标签读取map-----<br>
    <table>
        <tr><td>学号</td><td>姓名</td><td>年龄</td></tr>
        <c:forEach var="student" items="${stuMap}">
            <tr>
                <td>${student.key}</td>
                <td>${student.value.sid}</td>
                <td>${student.value.name}</td>
                <td>${student.value.age}</td>
            </tr>
        </c:forEach>
    </table>

    <br>-----使用JSTL标签读取指定for循环-----<br>
    <select>
        <c:forEach var="item" begin="1" end="10" step="1">
            <option> ${item} </option>
        </c:forEach>
    </select>

```



```
</select>

</body>
</html>
```

其中使用的javaBean

```
/**
 * @author zn
 * @date 2020/1/24
 */

public class Student {

    private String sid;
    private String name;
    private int age;

    public String getsid() {
        return sid;
    }

    public void setSid(String sid) {
        this.sid = sid;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public Student(String sid, String name, int age) {
        this.sid = sid;
        this.name = name;
        this.age = age;
    }
}
```

第五章 Listener、Filter

一、概念

- **servlet**

servlet是一种运行服务器端的java应用程序，他可以用来处理请求和响应。

- **filter**

过滤器，不像Servlet，它不能产生一个请求或者响应，它是一个中间者，能修改处理经过他的请求和响应，并不能直接给客户端响应。

- **listener**

监听器，他用来监听容器内的一些变化，如session的创建，销毁等。当变化产生时，监听器就要完成一些工作。

二、生命周期

1、servlet：servlet的生命周期始于它被装入web服务器的内存时，并在web服务器终止或重新装入servlet时结束。servlet一旦被装入web服务器，一般不会从web服务器内存中删除，直至web服务器关闭或重新结束。

1. 装入：启动服务器时加载Servlet的实例；
2. 初始化：web服务器启动时或web服务器接收到请求时，或者两者之间的某个时刻启动。初始化工作有init () 方法负责执行完成；
3. 调用：从第一次到以后的多次访问，都是只调用doGet()或doPost()方法；
4. 销毁：停止服务器时调用destroy()方法，销毁实例。

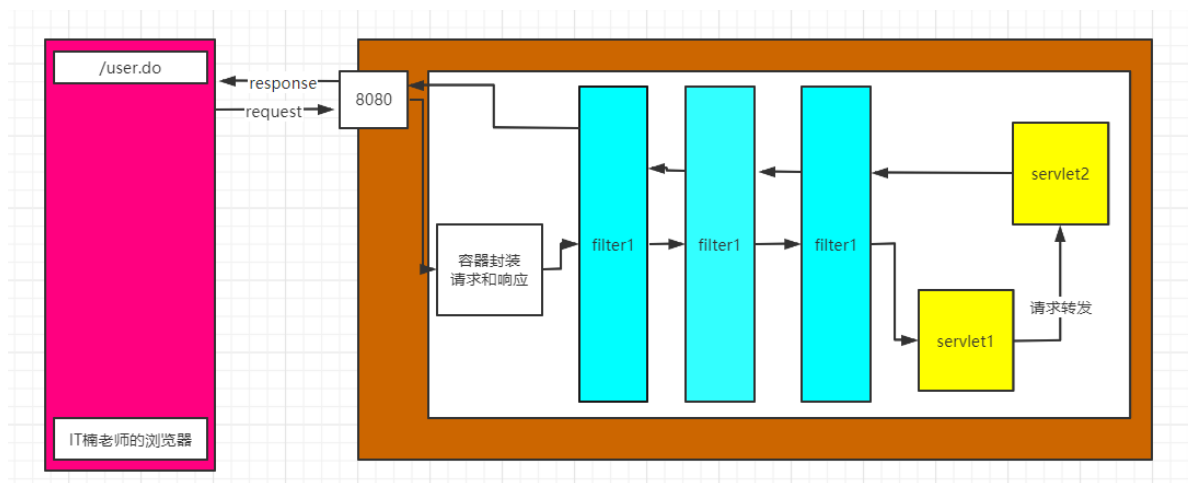
2、filter：一定要实现javax.servlet包的Filter接口的三个方法init()、doFilter()、destroy()，空实现也行

1. 启动服务器时加载过滤器的实例，并调用init()方法来初始化实例；
2. 每一次请求时都只调用方法doFilter()进行处理；
3. 停止服务器时调用destroy()方法，销毁实例。

3、listener：类似于servlet和filter

servlet2.4规范中提供了8个listener接口，可以将其分为三类，分别如下：

- 第一类：与servletContext有关的listener接口。包括：ServletContextListener、ServletContextAttributeListener
- 第二类：与HttpSession有关的Listener接口。包括：HttpSessionListener、HttpSessionAttributeListener、HttpSessionBindingListener、HttpSessionActivationListener；
- 第三类：与ServletRequest有关的Listener接口，包括：ServletRequestListener、ServletRequestAttributeListener



web.xml 的加载顺序是：context- param -> listener -> filter -> servlet

三、使用方式

listener:

```
public class TestListener implements
HttpSessionListener,ServletRequestListener,ServletRequestAttributeListener {
    private Logger logger = LoggerFactory.getLogger(TestListener.class);

    //sessionListener start!
    public void sessionCreated(HttpSessionEvent arg0) {
        logger.info(".....TestListener sessionCreated().....");
    }

    public void sessionDestroyed(HttpSessionEvent arg0) {
        logger.info(".....TestListener sessionDestroyed().....");
    }
    //sessionListener end!

    //requestListener start!
    public void requestInitialized(ServletRequestEvent arg0) {
        logger.info(".....TestListener requestInitialized().....");
    }

    public void requestDestroyed(ServletRequestEvent arg0) {
        logger.info(".....TestListener requestDestroyed().....");
    }
    //requestListener end!

    //attributeListener start!
    public void attributeAdded(ServletRequestAttributeEvent srae) {
        logger.info(".....TestListener attributeAdded().....");
    }

    public void attributeRemoved(ServletRequestAttributeEvent srae) {
        logger.info(".....TestListener attributeRemoved().....");
    }

    public void attributeReplaced(ServletRequestAttributeEvent srae) {
        logger.info(".....TestListener attributeReplaced().....");
    }
}
```

```

    }
    //attributeListener end!
}

```

Filter:

```

public class TestFilter implements Filter {
    private Logger logger = LoggerFactory.getLogger(TestFilter.class);

    public void destroy() {
        logger.info(".....execute TestFilter destroy().....");
    }

    public void doFilter(ServletRequest arg0, ServletResponse arg1,
        FilterChain arg2) throws IOException, ServletException {
        logger.info(".....execute TestFilter
doFilter().....");
        arg2.doFilter(arg0, arg1);
    }

    public void init(FilterConfig arg0) throws ServletException {
        logger.info(".....execute TestFilter init().....");
    }
}

```

Servlet

```

public class TestServlet extends HttpServlet {

    private Logger logger = LoggerFactory.getLogger(TestServlet.class);
    private static final long serialVersionUID = -4263672728918819141L;

    @Override
    public void init() throws ServletException {
        logger.info("...TestServlet init() init.....");
        super.init();
    }

    @Override
    public void destroy() {
        logger.info("...TestServlet init() destroy.....");
        super.destroy();
    }

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        this.doPost(req, resp);
    }

    @Override
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response)

```

```

        throws ServletException, IOException {
    logger.info("...TestServlet doPost() start.....");
    //操作attribute
    request.setAttribute("a", "a");
    request.setAttribute("a", "b");
    request.getAttribute("a");
    request.removeAttribute("a");
    //操作session
    request.getSession().setAttribute("a", "a");
    request.getSession().getAttribute("a");
    request.getSession().invalidate();
    logger.info("...TestServlet doPost() end.....");
}
}

```

配置XML

```

<!-- 测试filter -->
<filter>
    <filter-name>TestFilter</filter-name>
    <filter-class>com.xy.web.filter.TestFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>TestFilter</filter-name>
    <url-pattern>*.do</url-pattern>
</filter-mapping>
<!-- 测试servlet -->
<servlet>
    <servlet-name>TestServlet</servlet-name>
    <servlet-class>com.xy.web.servlet.TestServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>TestServlet</servlet-name>
    <url-pattern>/*</url-pattern>
</servlet-mapping>
<!-- 测试listener -->
<listener>
    <listener-class>com.xy.web.listener.TestListener</listener-class>
</listener>

```