

1- 动态web

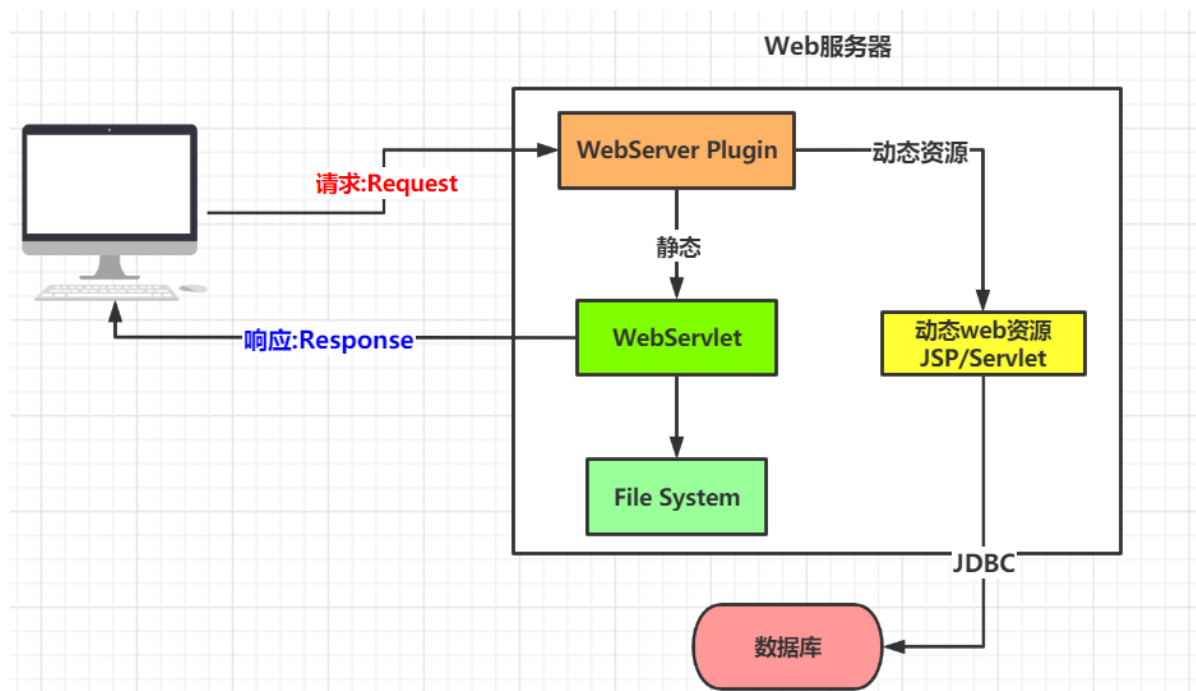
1.1 web服务器

本身是一个程序，运行在服务器上。也称为Web容器，写的程序是运行在Web容器中。

容器作用：

- 共享资源(图片，网页)，将服务器上资源分享给浏览器。解析写的Java程序
- 处理用户发送的请求，并且对请求做出响应。把生成的结果以网页的方式显示在浏览器上。

基本结构



服务器缺点： 服务器的动态web资源出现错误，需要重新编写我们的后台程序,重新发布。


服务器优点： Web页面可以动态更新，所有用户看到都不是同一个页面，它可以与数据库交互。

2- Tomcat

2.1 安装tomcat

tomcat官网：<https://tomcat.apache.org/>

← → ↻ 🔍 tomcat.apache.org/download-90.cgi



Apache Tomcat®

Search...

APACHECON
September 21-23
2021
[Save the date!](#)

Apache Tomcat
Home
Taglibs
Maven Plugin

Download
Which version?
Tomcat 10
Tomcat 9
Tomcat 8
Tomcat 7
Tomcat Migration Tool for Jakarta EE
Tomcat Connectors
Tomcat Native
Taglibs
Archives

Documentation
Tomcat 10.0
Tomcat 9.0
Tomcat 8.5
Tomcat 7.0
Tomcat Connectors
Tomcat Native
Wiki
Migration Guide
Presentations
Specifications

Problems?

Tomcat 9 Software Downloads

Welcome to the Apache Tomcat® 9.x software download page. This page provides download links for obtaining the latest version of Tomcat 9.0.x software. Unsure which version you need? Specification versions implemented, minimum Java version required and lots more useful information may be found on the [Tomcat 9.0.x page](#).

Quick Navigation

[KEYS](#) | [9.0.44](#) | [Browse](#) | [Archives](#)

Release Integrity

You **must** [verify](#) the integrity of the downloaded files. We provide OpenPGP signatures for every release file. This signature should be matched against the Managers. We also provide [SHA-512](#) checksums for every release file. After you download the file, you should calculate a checksum for your download, and compare it to the checksums provided here.

Mirrors

You are currently using <https://apachemirror.sg.wuchna.com/>. If you encounter a problem with this mirror, please select another mirror. If all mirrors are unavailable, please try again later.

Other mirrors:

9.0.44

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

Binary Distributions

- Core:
 - [zip \(pgp, sha512\)](#)
 - [tar.gz \(pgp, sha512\)](#)
 - [32-bit Windows zip \(pgp, sha512\)](#)
 - [64-bit Windows zip \(pgp, sha512\)](#)
 - [32-bit/64-bit Windows Service Installer \(pgp, sha512\)](#)
- Full documentation:
 - [tar.gz \(pgp, sha512\)](#)

下载此安装绿色版

2.2 Tomcat配置

2.2.1 环境变量的配置

```
CATALINA_HOME  
%CATALINA_HOME%\bin
```

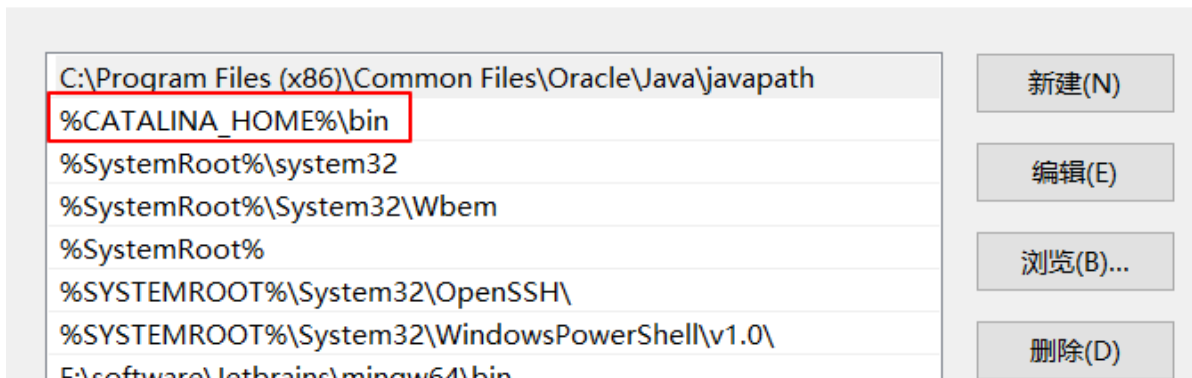
添加Tomcat的安装目录

新建系统变量

变量名(N):

变量值(V):

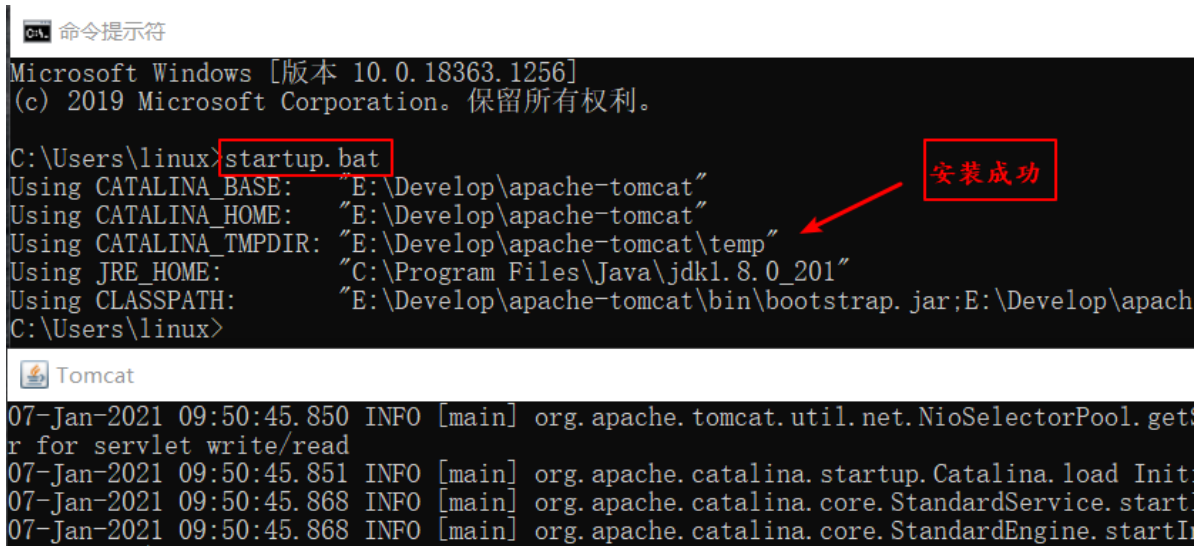
配置Path



Tomcat的启动与关闭

常用命令

启动的命令: startup.bat
关闭的命令: shutdown.bat



2.3 Tomcat的目录结构



2.4 Tomcat项目发布

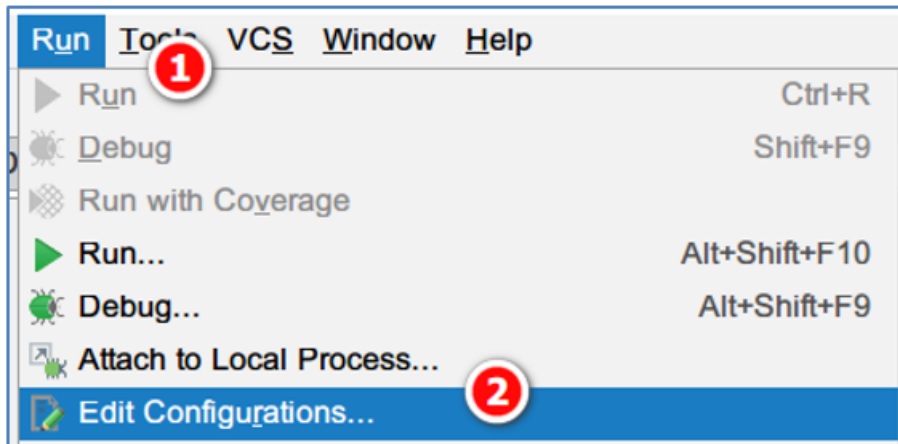
方式1: 直接将项目复制到webapps目录下。

方式2: 采用压缩文件.war的方式。

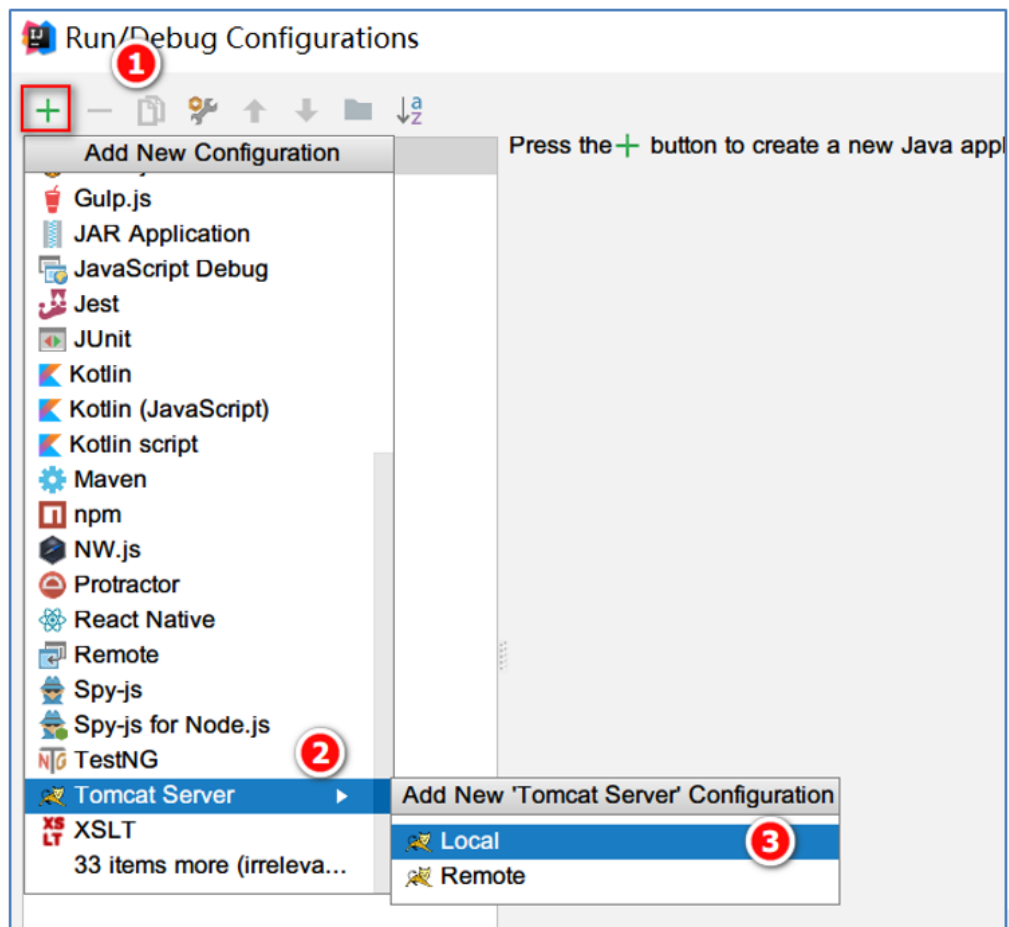
将整个项目使用压缩工具打包成一个zip文件,将zip的扩展名为war。
复制到webapps目录下, tomcat会自动解压成一个同名的目录。

2.5 idea中配置Tomcat

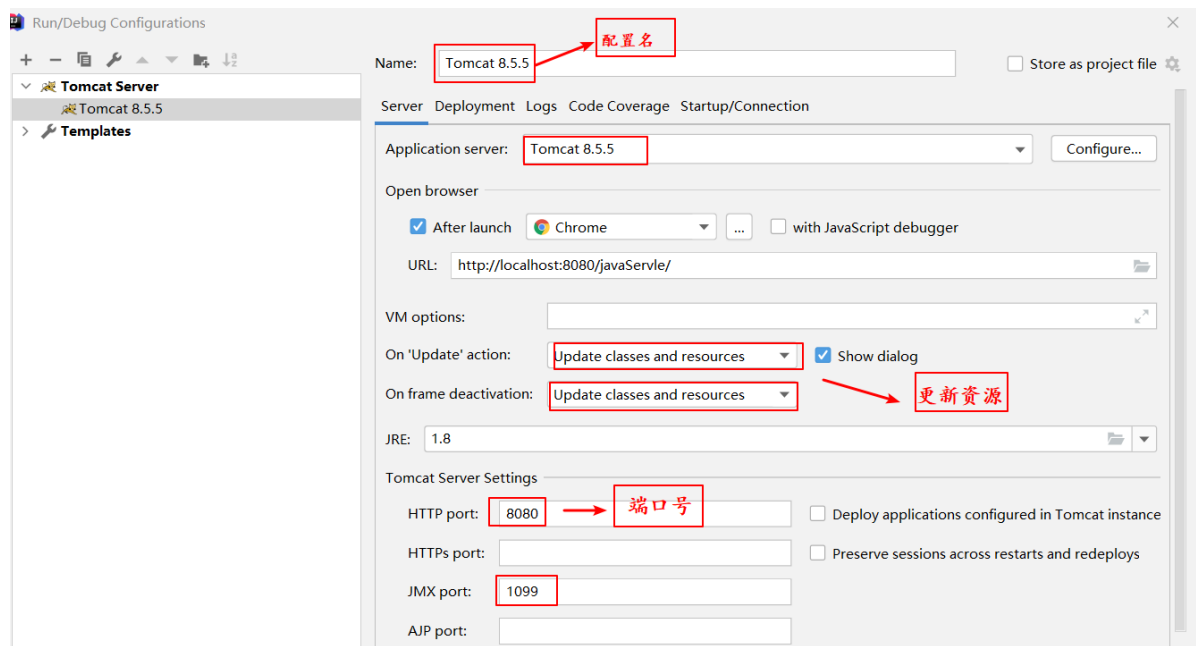
编辑运行配置



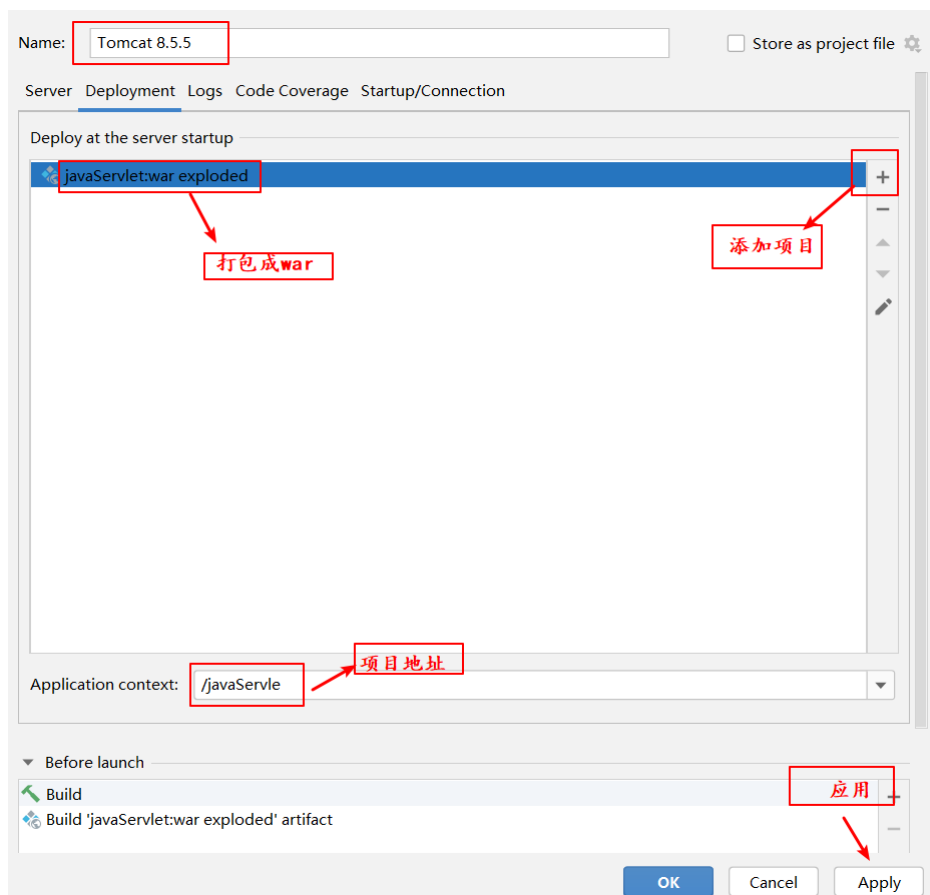
添加Tomcat的配置服务器信息



配置服务器的详细信息



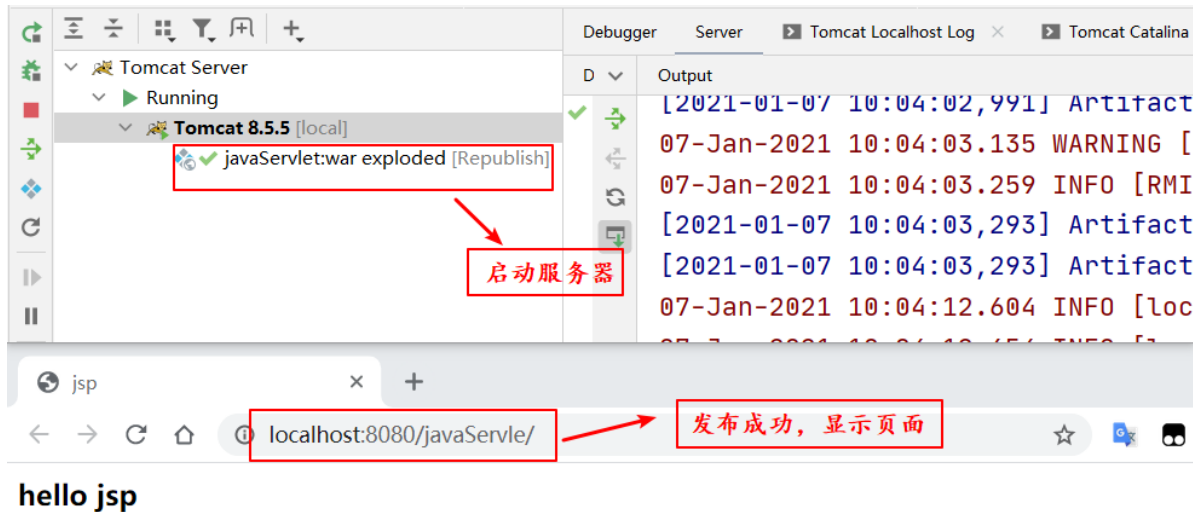
修改项目发布的访问地址



点右上角的启动图标，启动Tomcat服务器



服务器启动成功的信息



日志乱码现象

启动Tomcat的时候会出 乱码 [main] org.apache.catalina.startup.VersionLoggerListener.log
Server. 增加一个链接:

解决这种乱码的问题, 解决问题的方法是到tomcat/conf/目录下,修改logging.properties

将java.util.logging.ConsoleHandler.encoding = utf-8
更改为 java.util.logging.ConsoleHandler.encoding = GBK


3- Maven

3.1 下载安装Maven

官网 <https://maven.apache.org/>

Maven - Download Apache Maven x +

← → ↺ 🔒 https://maven.apache.org/download.cgi

 **Apache Maven Project**
http://maven.apache.org/

Apache / Maven / Download Apache Maven 📄

Welcome
License

ABOUT MAVEN
What is Maven?
Features
Download
Use
Release Notes

DOCUMENTATION
Maven Plugins
Index (category)
User Centre
Plugin Developer Centre
Maven Central Repository
Maven Developer Centre
Books and Resources
Security

COMMUNITY
Community Overview
Project Roles
How to Contribute
Getting Help
Issue Management
Getting Maven Source
The Maven Team

PROJECT DOCUMENTATION
Project Information

Downloading Apache Maven 3.6.3

Apache Maven 3.6.3 is the latest release and recommended version for all users.

The currently selected download mirror is <https://mirror.jframeworks.com/apache/>. If you encounter a problem with this mirror, please select another mirror may also consult the [complete list of mirrors](#).

Other mirrors:

System Requirements

Java Development Kit (JDK)	Maven 3.3+ require JDK 1.7 or above to execute - they still allow you to build against 1.3 and other JDK versions by Using
Memory	No minimum requirement
Disk	Approximately 10MB is required for the Maven installation itself. In addition to that, additional disk space will be used for y 500MB.
Operating System	No minimum requirement. Start up scripts are included as shell scripts and Windows batch files.

Files

Maven is distributed in several formats for your convenience. Simply pick a ready-made binary distribution archive and follow the [installation instructions](#). U In order to guard against corrupted downloads/installations, it is highly recommended to [verify the signature](#) of the release bundles against the public [KEYS](#)

	Link	Checksums
Binary tar.gz archive	apache-maven-3.6.3-bin.tar.gz	apache-maven-3.6.3-bin.tar.gz.sha512
Binary zip archive	apache-maven-3.6.3-bin.zip	apache-maven-3.6.3-bin.zip.sha512
Source tar.gz archive	apache-maven-3.6.3-src.tar.gz	apache-maven-3.6.3-src.tar.gz.sha512
Source zip archive	apache-maven-3.6.3-src.zip	apache-maven-3.6.3-src.zip.sha512

3.2 配置环境变量

配置如下：

- MAVEN_HOME maven的目录
- path: %MAVEN_HOME%\bin

编辑系统变量

变量名(N): 安装目录

变量值(V):

C:\Program Files\Git\cmd
C:\Program Files\nodejs\
%JAVA_HOME%\bin
%MYSQL_HOME%\bin
%MAVEN_HOME%\bin 配置环境

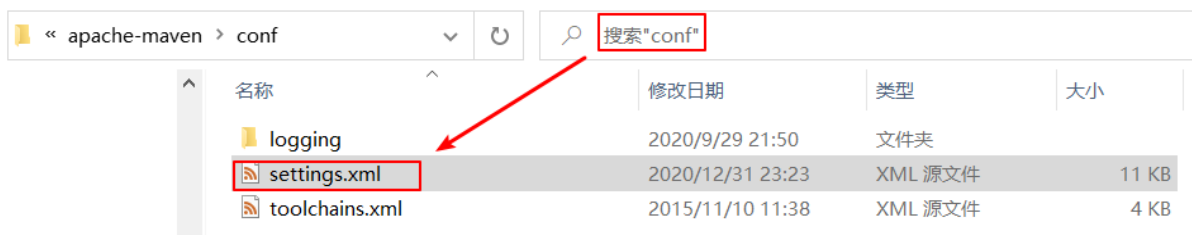
安装成功

```
Windows PowerShell
版权所有 (C) Microsoft Corporation。保留所有权利。

尝试新的跨平台 PowerShell https://aka.ms/pscore6

PS C:\Users\linux> mvn -version
Apache Maven 3.3.9 (bb52d8502b132ec0a5a3f4c09453c07478323dc5; 2015-11-11T00:41:47+08:00)
Maven home: E:\Develop\apache-maven\bin\..
Java version: 1.8.0_201, vendor: Oracle Corporation
Java home: C:\Program Files\Java\jdk1.8.0_201\jre
Default locale: zh_CN, platform encoding: GBK
OS name: "windows 10", version: "10.0", arch: "amd64", family: "dos"
PS C:\Users\linux>
```

3.3 阿里云镜像



国内使用阿里云的镜像

```
<mirror>
  <id>nexus-aliyun</id>
  <mirrorOf>central</mirrorOf>
  <name>Nexus aliyun</name>
  <url>http://maven.aliyun.com/nexus/content/groups/public</url>
</mirror>
```

3.4 本地仓库

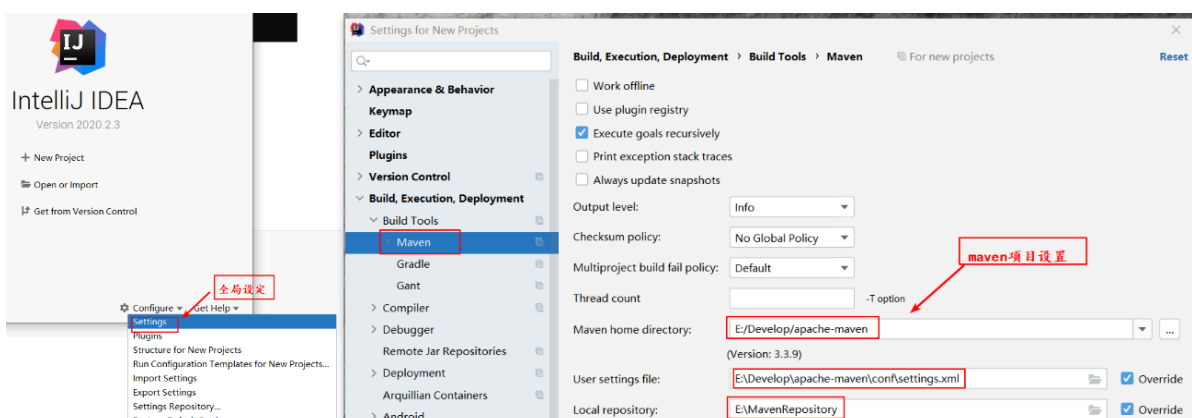
在本地的仓库，远程仓库；

建立一个本地仓库：localRepository

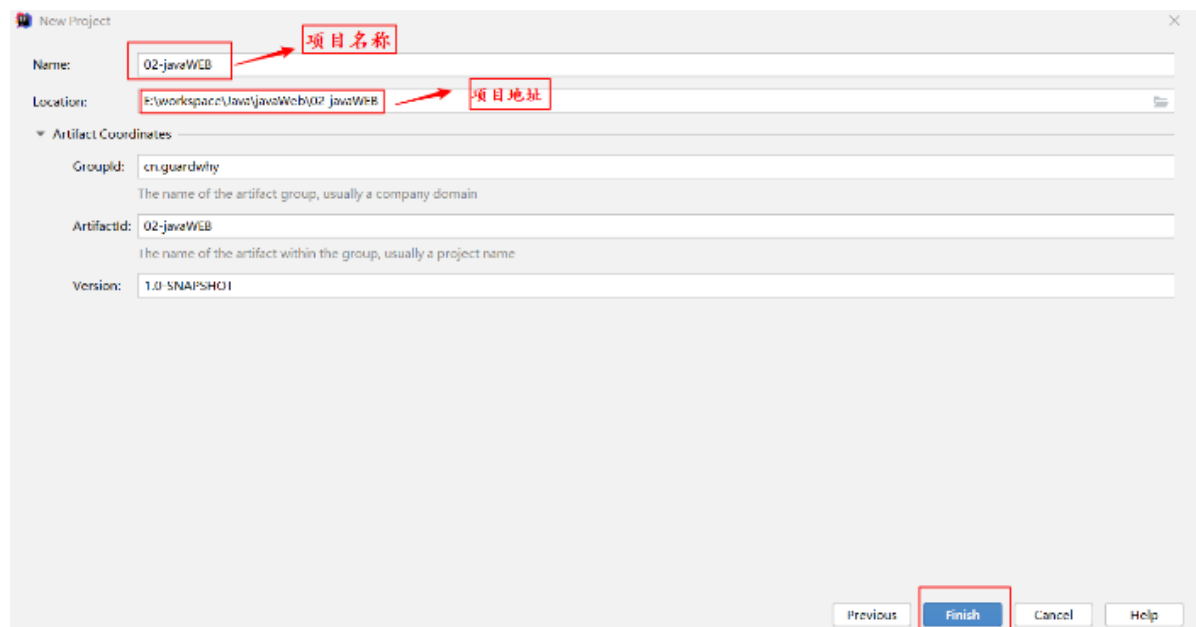
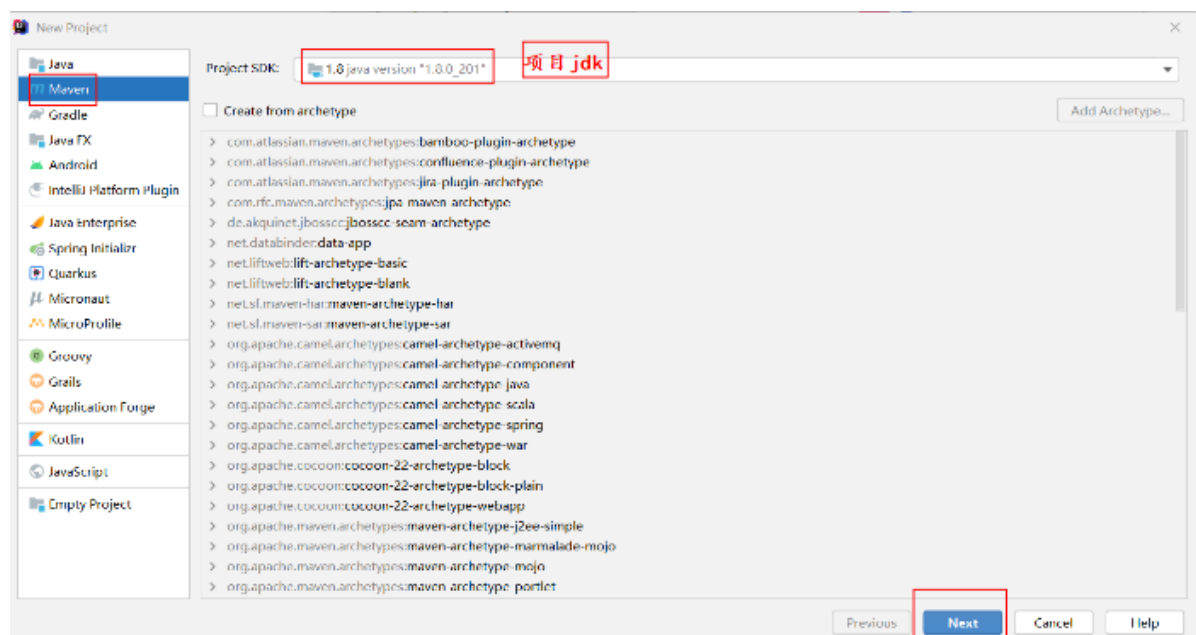
```
<localRepository>E:\Develop\Repository</localRepository>
```

3.5 创建Maven项目

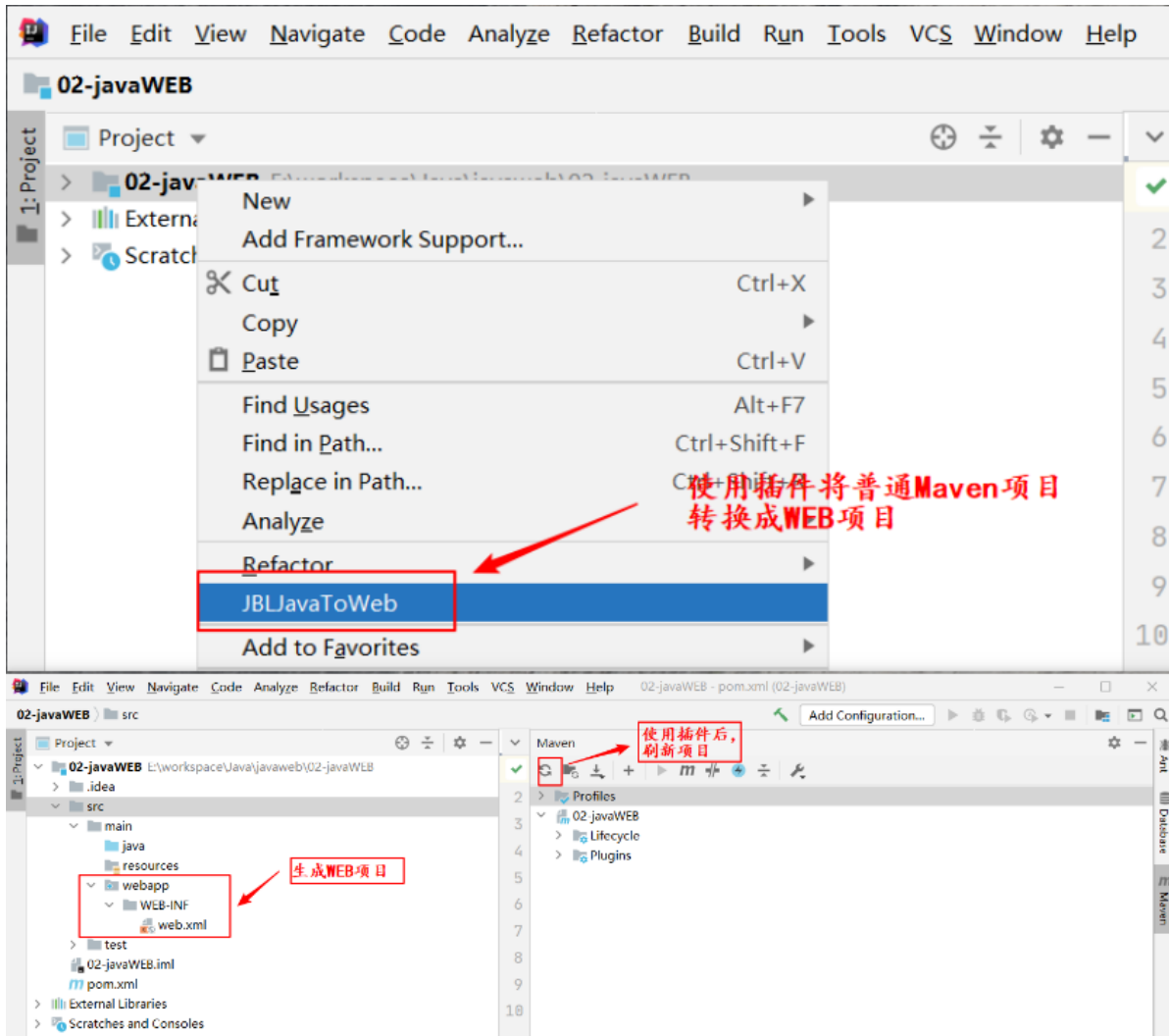
3.5.1 全局设置项目Maven



3.5.2 创建Maven项目



3.5.3 普通项目转换成WEB项目

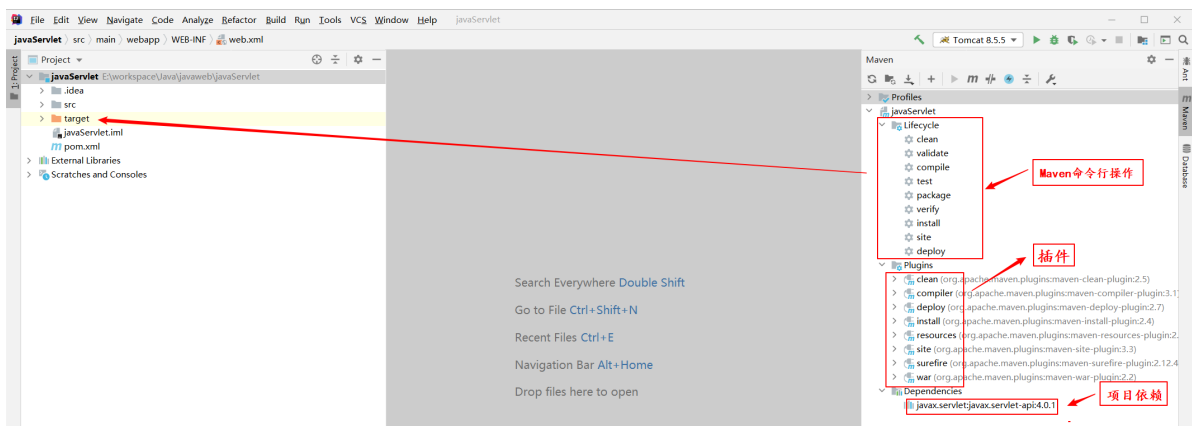


将项目中的web.xml(2.5)替换成web.xml(4.0)

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
  version="4.0">
</web-app>
```

3.5.4 pom文件

pom.xml 是Maven的核心配置文件



导入相关依赖

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.example</groupId>
  <artifactId>javaServlet</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging>

  <!--引入依赖-->
  <dependencies>
    <!--servlet依赖-->
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
      <version>4.0.1</version>
    </dependency>
    <!--基本测试-->
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
    </dependency>
  </dependencies>
</project>
```

maven约定大于配置，可能遇到写的配置文件，无法被导出或者生效的问题

解决方案

```
<build>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <includes>
        <include>**/*.properties</include>
        <include>**/*.xml</include>
      </includes>
      <filtering>true</filtering>
    </resource>
    <resource>
      <directory>src/main/java</directory>
      <includes>
        <include>**/*.properties</include>
        <include>**/*.xml</include>
      </includes>
      <filtering>true</filtering>
    </resource>
  </resources>
</build>
```

4-Servlet

4.1 Servlet基本概念

4.1.1 什么是Servlet

本质上就是一个Java类，运行在Tomcat中，由Tomcat来调用。作用：生成一个网页，输出到浏览器。

4.1.2 Servlet与Java程序区别

- Servlet本质上就是一个Java类
- 所有的Servlet必须要实现javax.servlet.Servlet接口
- 运行在Tomcat容器中,用于接收浏览器的请求，并且做出响应。

4.2 实现Servlet

4.2.1 Servlet2.5的方式开发

创建一个类继承于HttpServlet类，它已经实现了Servlet接口。
重写doGet或doPost方法，用来处理浏览器发送的get或post请求。
配置web/WEB-INF/web.xml文件，配置servlet的访问地址。

1. 编写一个普通类,实现Servlet接口，直接继承HttpServlet。

```
package cn.guardwhy;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;

public class HelloServletDemo01 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // 1.设置响应内容的类型和编码
        response.setContentType("text/html;charset=utf-8");
        // 2.得到打印流
        PrintWriter out = response.getWriter();
        // 3.向浏览器输出文本
        out.print("<h3>Hello, Servlet!!!<h3>");
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        // super.doPost(req, resp);
    }
}
```

- 2.编写Servlet的映射

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
  version="4.0">

  <!--配置Servlet-->
  <servlet>
    <!--servlet的名字-->
    <servlet-name>demo01</servlet-name>
    <!--配置servlet的类全名-->
    <servlet-class>cn.guardwhy.HelloServletDemo01</servlet-class>
  </servlet>
  <!--配置访问地址-->
  <servlet-mapping>
    <servlet-name>demo01</servlet-name>
    <!--访问地址，必须以/开头 /相当于是web这个根目录 -->
    <url-pattern>/demo01</url-pattern>
  </servlet-mapping>
</web-app>
```

4.2.2 访问流程

```
<!--配置Servlet-->
<servlet>
  <!--servlet的名字--> 3. 找到名字
  <servlet-name>demo01</servlet-name>
  <!--配置servlet的类全名-->
  <servlet-class>cn.guardwhy.HelloServletDemo01</servlet-class>
</servlet>
<!--配置访问地址-->
<servlet-mapping>
  <servlet-name>demo01</servlet-name> 2. 找到servlet名字
  <!--访问地址，必须以/开头 /相当于是web这个根目录 -->
  <url-pattern>/demo01</url-pattern>
</servlet-mapping>
```

1. 服务器地址

4. 通过反射实例化对象Class.forName("类全名")

4.2.3 Servlet3.0的方式开发

1. 创建Servlet类，使用注解@WebServlet，无需配置web.xml。

@WebServlet注解	说明
name	Servlet的名字，相当于<servlet-name>
urlPatterns	配置访问地址，相当于<url-pattern> 可以配置多个访问地址
value	就是访问地址，与urlPatterns是一样的。 如果只有一个value属性设置，value的名字可以省略

2.编写一个普通类,实现Servlet接口

```

package cn.guardwhy;

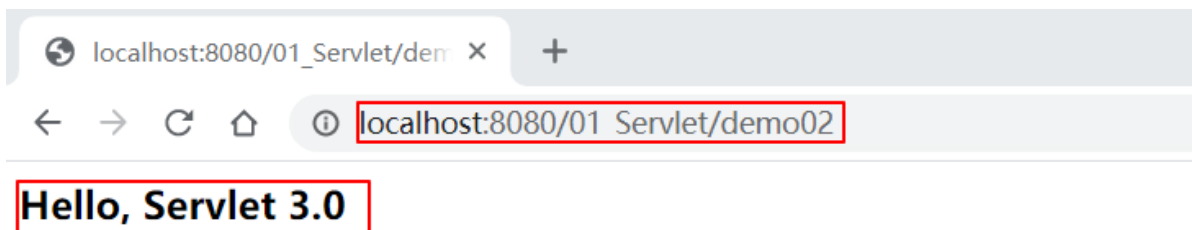
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;

@WebServlet("/demo02")
public class HelloServletDemo02 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // 1.设置响应内容的类型和编码
        response.setContentType("text/html;charset=utf-8");
        // 2.得到打印流
        PrintWriter out = response.getWriter();
        // 3.向浏览器输出文本
        out.print("<h3>Hello, Servlet 3.0<h3>");
    }

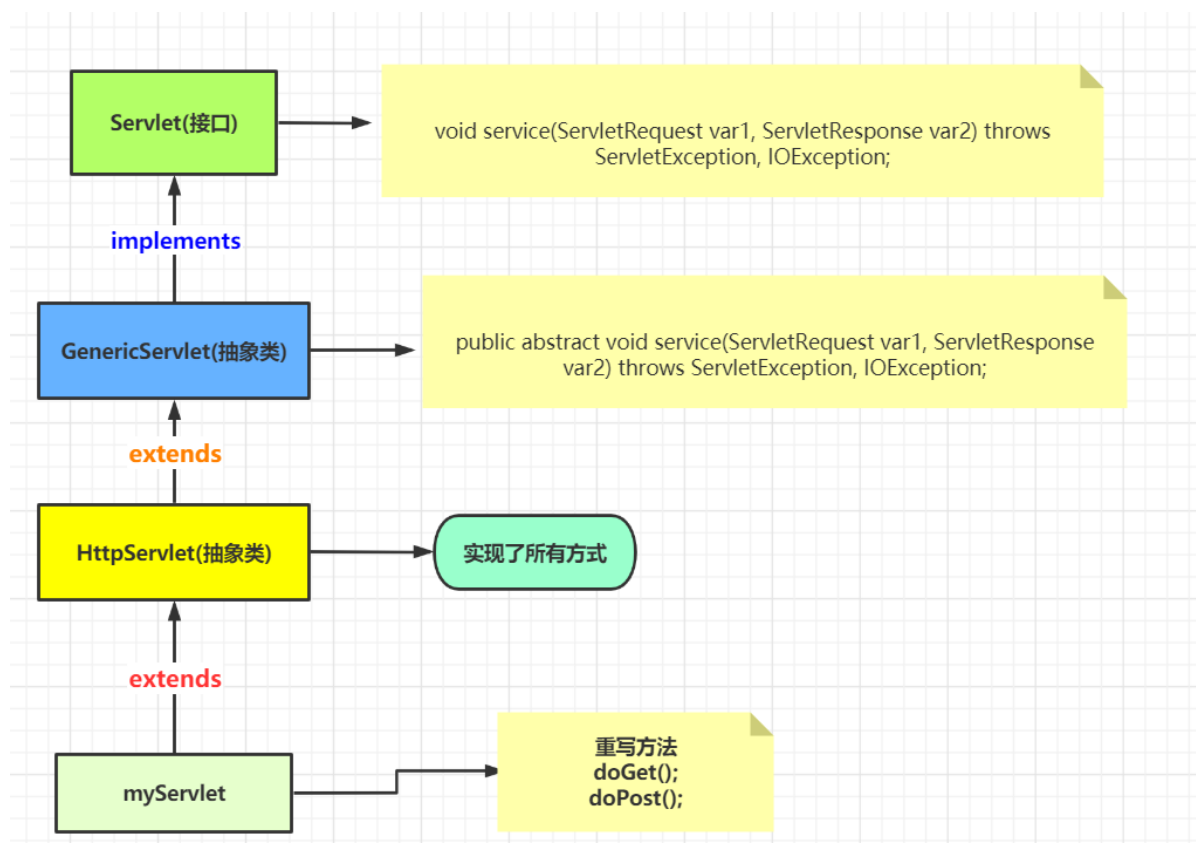
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        // super.doPost(req, resp);
    }
}

```

执行结果



4.3 Servlet继承和实现



4.3.1 Servlet接口

(1) 基本概念

javax.servlet.Servlet接口用于定义所有servlet必须实现的方法。

(2) 常用的方法

方法声明	作用
void init(ServletConfig config)	由servlet容器调用，以向servlet指示servlet正在被放入服务中
void service(ServletRequest req, ServletResponse res)	由servlet容器调用，以允许servlet响应请求
void destroy()	在Servlet销毁的时候执行，服务器关闭的时候执行。
ServletConfig getServletConfig()	返回ServletConfig对象，该对象包含此servlet的初始化和启动参数。
String getServletInfo()	返回有关servlet的信息，如作者、版本和版权

4.3.2 GenericServlet类

(1) 基本概念

- javax.servlet.GenericServlet类主要用于定义一个通用的、与协议无关的servlet，该类实现了Servlet接口。
- 若编写通用servlet，只需重写service抽象方法即可。

(2) 常用的方法

方法声明	作用
abstract void service(ServletRequest req,ServletResponse res)	由servlet容器调用允许servlet响应请求。

4.3.3 HttpServlet类

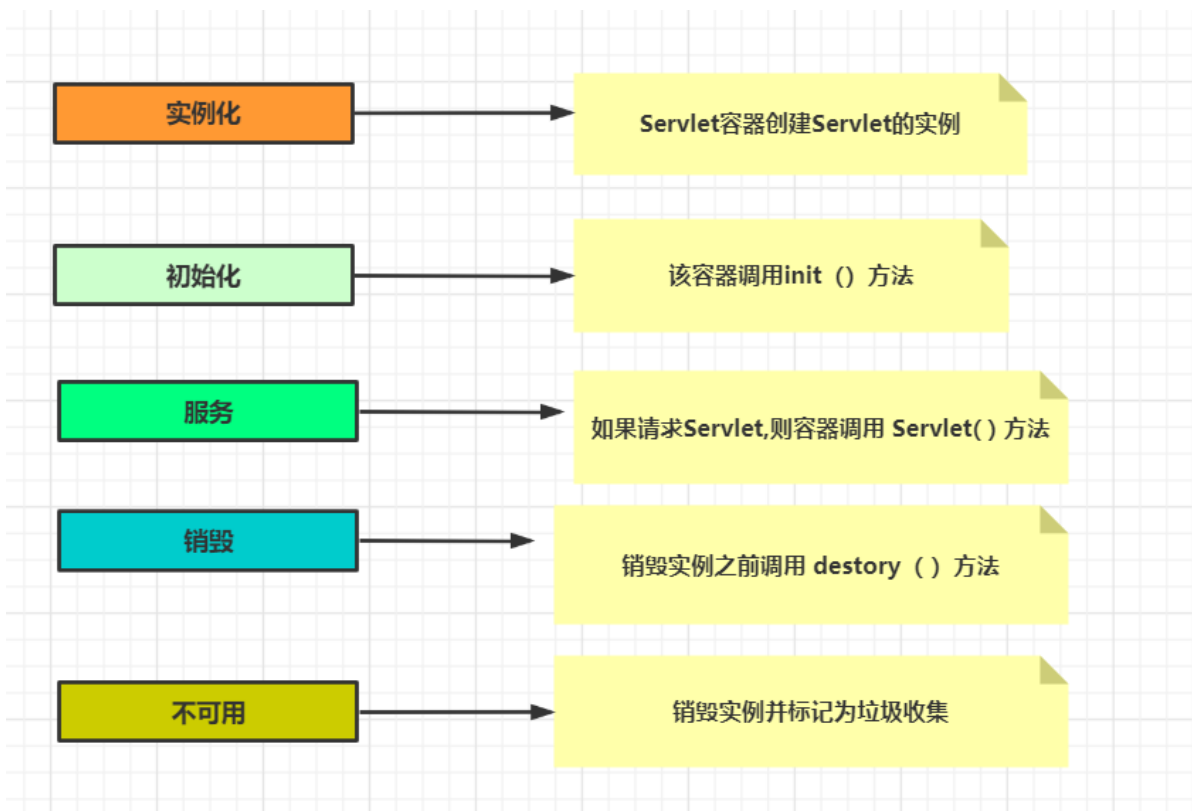
(1) 基本概念

- javax.servlet.http.HttpServlet类是个抽象类并继承了GenericServlet类。
- 用于创建适用于网站的HTTP Servlet，该类的子类必须至少重写一个方法。

(2) 常用的方法

方法声明	作用
void doGet(HttpServletRequest req,HttpServletResponse resp)	处理客户端的GET请求
void doPost(HttpServletRequest req,HttpServletResponse resp)	处理客户端的POST请求
void destroy()	删除实例时释放资源
void init()	进行初始化操作
void service(HttpServletRequest req,HttpServletResponse resp)	根据请求决定调用doGet还是doPost方法

4.4 Servlet的生命周期

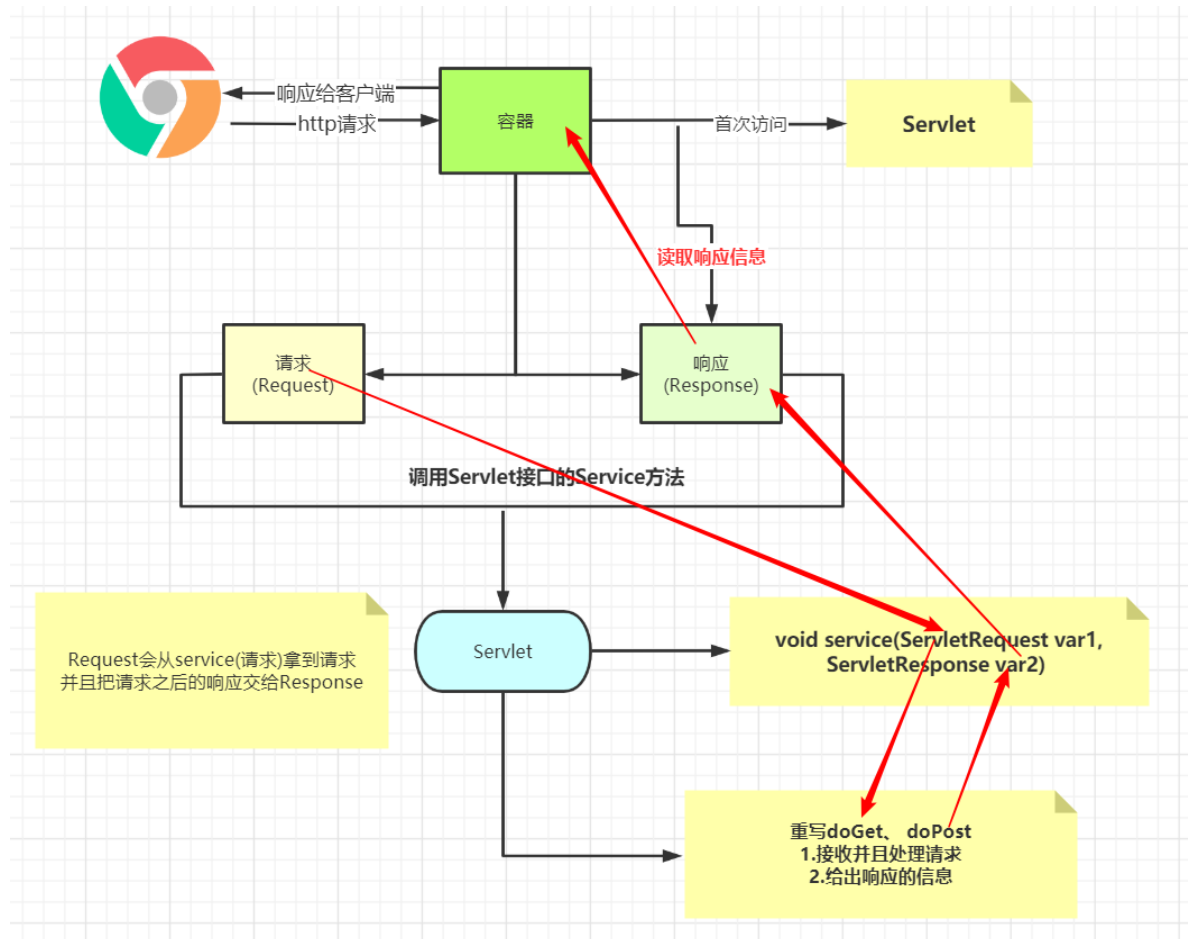


- 构造方法只被调用一次，当第一次请求Servlet时调用构造方法来创建Servlet的实例。
- init方法只被调用一次，当创建好Servlet实例后立即调用该方法实现Servlet的初始化。

- service方法被多次调用，每当有请求时都会调用service方法来用于请求的响应。
- destroy方法只被调用一次，当该Servlet实例所在的Web应用被卸载前调用该方法来释放当前占用的资源。

4.5 Servlet的原理

Servlet是由Web服务器调用，web服务器在收到浏览器请求之后实现以下步骤



5- Request和Response

5.1 POST和GET请求

5.1.1 GET请求

发出GET请求的主要方式：

- (1) 在浏览器输入URL按回车
- (2) 点击超链接
- (3) 点击submit按钮，提交 `<form method="get">` 表单

GET请求特点：

会将请求数据添加到请求URL地址的后面，只能提交少量的数据、不安全

5.1.2 POST请求

发出POST请求的方法如下：

点击submit按钮，提交 `<form method="post">` 表单

POST请求的特点：

请求数据添加到HTTP协议体中，可提交大量数据、安全性好

5.2 ServletRequest接口

5.2.1 基本概念

- javax.servlet.ServletRequest接口主要用于向servlet提供客户端请求信息，可以从中获取到任何请求信息。
- Servlet容器创建一个ServletRequest对象，并将其作为参数传递给Servlet的service方法。

5.2.2 常用的方法

方法声明	作用
String getParameter(String name)	以字符串形式返回请求参数的值，如果该参数不存在，则返回空值。
String[] getParameterValues (String name)	返回一个字符串对象数组，其中包含给定请求参数所具有的所有值，如果该参数不存在，则返回空值。
Enumeration getParameterNames()	返回包含此请求中包含的参数名称的字符串对象的枚举。如果请求没有参数，则方法返回空枚举。
Map<String, String[]> getParameterMap()	返回请求参数的键值对，一个键可以对应多个值。
String getRemoteAddr()	根据请求决定调用doGet还是doPost方法
String getRemoteAddr()	返回发送请求的客户端或最后一个代理的IP地址
int getRemotePort()	返回发送请求的客户端或最后一个代理的端口号

5.3 HttpServletRequest接口

5.3.1 基本概念

- javax.servlet.http.HttpServletRequest接口是ServletRequest接口的子接口，主要用于提供HTTP请求信息的功能。
- 不同于表单数据，在发送HTTP请求时，HTTP请求头直接由浏览器设置。
- 可直接通过HttpServletRequest对象提供的一系列get方法获取请求头数据。

5.3.2 常用的方法

方法声明	作用
String getRequestURI()	以字符串形式返回请求参数的值，如果该参数不存在，则返回空值。
StringBuffer getRequestURL()	返回一个字符串对象数组，其中包含给定请求参数所具有的所有值，如果该参数不存在，则返回空值。
String getMethod()	返回包含此请求中包含的参数名称的字符串对象的枚举。如果请求没有参数，则方法返回空枚举。
String getQueryString()	返回请求参数的键值对，一个键可以对应多个值。
String getServletPath()	根据请求决定调用doGet还是doPost方法

5.3.3 Servlet接收中文乱码

接收乱码原因

浏览器在提交表单时，会对中文参数值进行自动编码。当Tomcat服务器接收到浏览器请求后自动解码，当编码与解码方式不一致时，就会导致乱码。

解决POST接收乱码

接收之前设置编码方式：`request.setCharacterEncoding("utf-8")`
 注意：必须在调用`request.getParameter("name")`之前设置

解决GET接收乱码

将接收到的中文乱码重新编码：

```
// 接收到get请求的中文字符串
String name = request.getParameter("name");
// 将中文字符重新编码，默认编码为ISO-8859-1
String userName = new String(name.getBytes("ISO-8859-1"), "utf-8");
```

5.3.4 代码示例

1. 前端页面

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>请求参数的获取</title>
</head>
<body>
  <!-- 表单 -->
  <form action="parameter" method="post">
    姓名: <input type="text" name="name"/><br/>
    密码: <input type="password" name="password"/><br/>
    爱好: <input type="checkbox" name="hobby" value="basketball"/> 篮球
         <input type="checkbox" name="hobby" value="Java"/> Java
         <input type="checkbox" name="hobby" value="run"/> 跑步<br/>
    <input type="submit" value="提交">
  </form>
```

```
</body>
</html>
```

2. Servlet 服务端实现

```
package cn.guardwhy.servlet03;
/*
Request 相关方法
*/
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.Enumeration;
import java.util.Map;
import java.util.Set;

@WebServlet("/parameter")
public class RequestServlet extends HttpServlet {

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws
        ServletException, IOException {
        // 1. 设置编码
        request.setCharacterEncoding("utf-8");
        // 1. 获取指定参数名称对应的参数值
        String name = request.getParameter("name");
        System.out.println("用户名:" + name);

        String[] hobbies = request.getParameterValues("hobby");
        System.out.print("爱好:");

        for(String hb : hobbies){
            System.out.print(hb + " ");
        }
        System.out.println();

        System.out.println("+++++++");

        // 2. 获取所有参数的名称
        Enumeration<String> parameterNames = request.getParameterNames();
        System.out.print("获取到的所有参数:");
        while (parameterNames.hasMoreElements()){
            System.out.print(parameterNames.nextElement() + " ");
        }
        System.out.println();

        System.out.println("=====");

        // 3. 获取请求参数名和对应值
        Map<String, String[]> maps = request.getParameterMap();
        // 遍历集合
        Set<Map.Entry<String, String[]>> entries = maps.entrySet();
        for (Map.Entry<String, String[]> entry : entries){
```

```

        System.out.print(entry.getKey() + ":");
        for(String value : entry.getValue()){
            System.out.print(value + " ");
        }
        System.out.println();
    }
    System.out.println("~~~~~");

    // 4.获取客户端请求信息
    System.out.println("请求的IP地址: " + request.getRemoteAddr());
    System.out.println("请求的端口号: " + request.getRemotePort());
    System.out.println("请求资源的路径: " + request.getRequestURI());
    System.out.println("请求资源的完整路径为: " + request.getRequestURL());
    System.out.println("请求方式: " + request.getMethod());
    System.out.println("请求附带参数: " + request.getQueryString());
    System.out.println("请求Servlet路径: " + request.getServletPath());

}

@Override
protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws
    ServletException, IOException {

}
}

```

执行结果

```

用户名:guardwhy
爱好:basketball Java run
+++++++
获取到的所有参数:name password hobby
=====
name:guardwhy
password:113456
hobby:basketball Java run
~~~~~
请求的IP地址: 127.0.0.1
请求的端口号: 8000
请求资源的路径: /01_Servlet/parameter
请求资源的完整路径为: http://localhost:8080/01\_Servlet/parameter
请求方式: POST
请求附带参数: null
请求Servlet路径: /parameter

```

5.4 ServletResponse接口

5.4.1 基本概念

- javax.servlet.ServletResponse接口用于定义一个对象来帮助Servlet向客户端发送响应。
- Servlet容器创建ServletResponse对象，并将其作为参数传递给servlet的service方法。

5.4.2 常用方法

方法声明	作用
PrintWriter getWriter()	返回可向客户端发送字符文本的PrintWriter对象。
String getCharacterEncoding()	获取响应内容的编码方式。
void setContentType(String type)	如果尚未提交响应，则设置发送到客户端响应的内容类型。内容类型 可以包括字符编码规范，例如text/html;charset=UTF-8

5.5 HttpServletResponse接口

5.5.1 基本概念

- javax.servlet.http.HttpServletResponse接口继承ServletResponse接口，以便在发送响应时提供特定于HTTP的功能。

5.5.2 常用的方法

方法声明	作用
void sendRedirect(String location)	使用指定的重定向位置URL向客户端发送临时重定向响应

5.5.3 代码示例

```
package cn.guardwhy.servlet03;
/*
Response 相关方法
*/
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Enumeration;
import java.util.Map;
import java.util.Set;

@WebServlet("/parameter")
public class RequestServlet extends HttpServlet {

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {

        // 向浏览器发出响应数据
```

```

        System.out.println("$$$$$$$$$$$$$$$$$$$$");

        // 1. 设置服务器和浏览器的编码方式
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        // 2. 向浏览器中打印
        out.write("hello, ResponseServlet");
        System.out.println("服务器发送数据成功");
        // 3. 关闭资源
        out.close();

    }

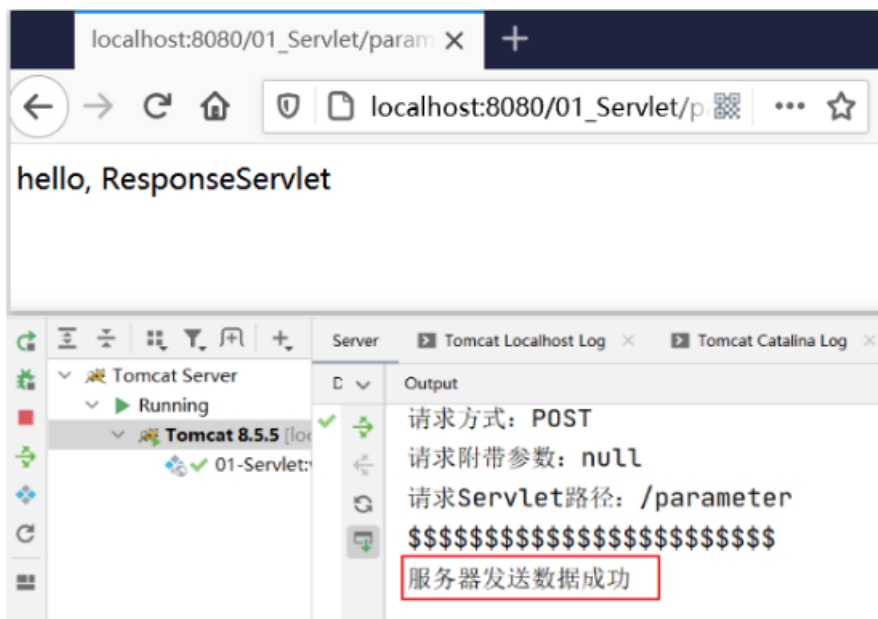
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {

    }

}

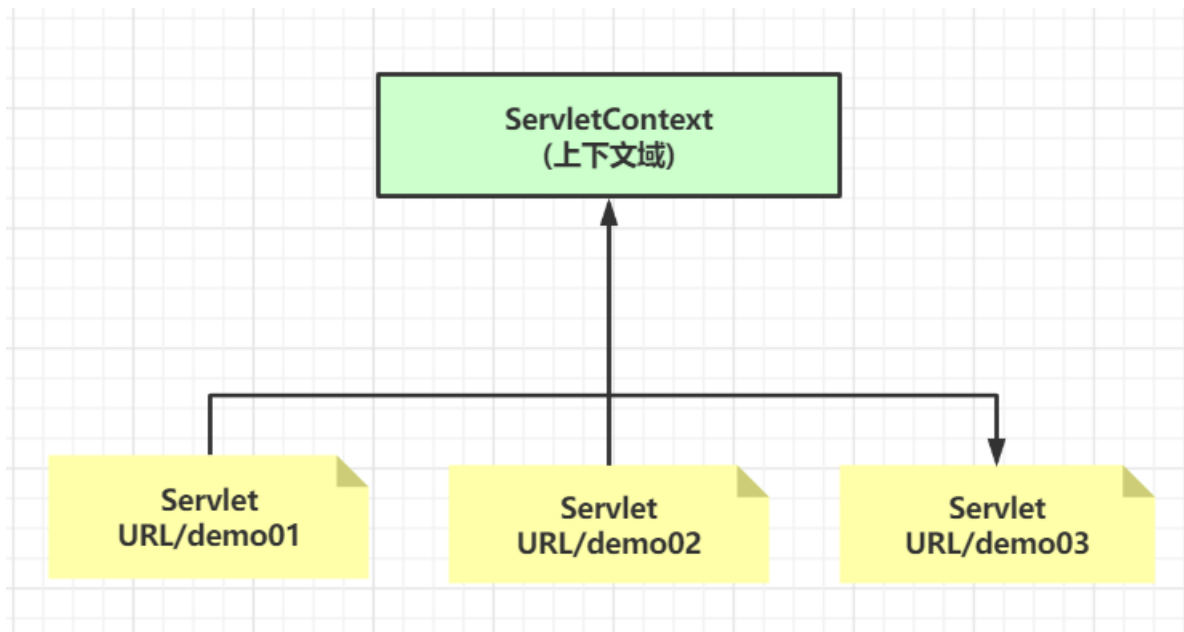
```

执行结果



5.6 ServletContext接口

5.6.1 基本概念



- javax.servlet.ServletContext接口主要用于定义一组方法，Servlet使用这些方法与它的Servlet容器通信。
- 服务器容器在启动时会为每个项目创建唯一的一个ServletContext对象，用于实现多个Servlet之间的信息共享和通信。
- Servlet中通过this.getServletContext()方法可以获得ServletContext对象。

5.6.2 常用的方法

方法声明	作用
String getInitParameter(String name)	返回包含初始化参数值的字符串，如果该参数不存在，则返回null
Enumeration getInitParameterNames()	将servlet的初始化参数的名称作为字符串对象的枚举返回，如果servlet没有初始化参数，则返回空枚举
void setAttribute(String name, Object object)	将指定的属性名和属性值绑定到当前对象。
Object getAttribute(String name)	根据执行的属性名获取属性值。
void removeAttribute(String name)	删除指定的属性名信息。

5.6.3 配置方式


```

<!-- 对于ServletContext对象的参数进行配置 -->
<context-param>
    <param-name>username</param-name>
    <param-value>guardwhy</param-value>
</context-param>
<context-param>
    <param-name>password</param-name>
    <param-value>hxy162</param-value>
</context-param>

```

5.6.4 代码示例

```

package cn.guardwhy.servlet03;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.Enumeration;

/*
 *   ServletContext:上下文域
 */

@WebServlet("/ContextServlet")
public class ContextServlet extends HttpServlet {

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        // 1.1配置参数的获取
        ServletContext servletContext = this.getServletContext();
        Enumeration<String> initParameterNames =
servletContext.getInitParameterNames();
        // 1.2遍历集合
        while (initParameterNames.hasMoreElements()){
            String str = initParameterNames.nextElement();
            System.out.println(str + "==>" +
servletContext.getInitParameter(str));
        }

        // 2.1 设置属性和获取属性信息
        servletContext.setAttribute("key", "guardwhy");
        Object key = servletContext.getAttribute("key");
        System.out.println("属性值:" + key);    // guardwhy

        // 2.2 移除属性
        servletContext.removeAttribute("key");
        key = servletContext.getAttribute("key");
        System.out.println("属性值:" + key);    // null
    }

    @Override

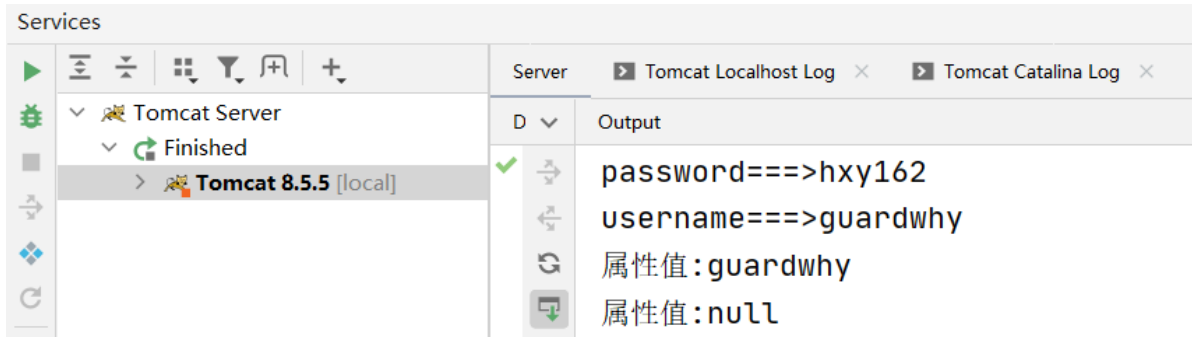
```

```

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

```

执行结果



6- 重定向和转发

6.1 重定向

6.1.1 重定向的概念

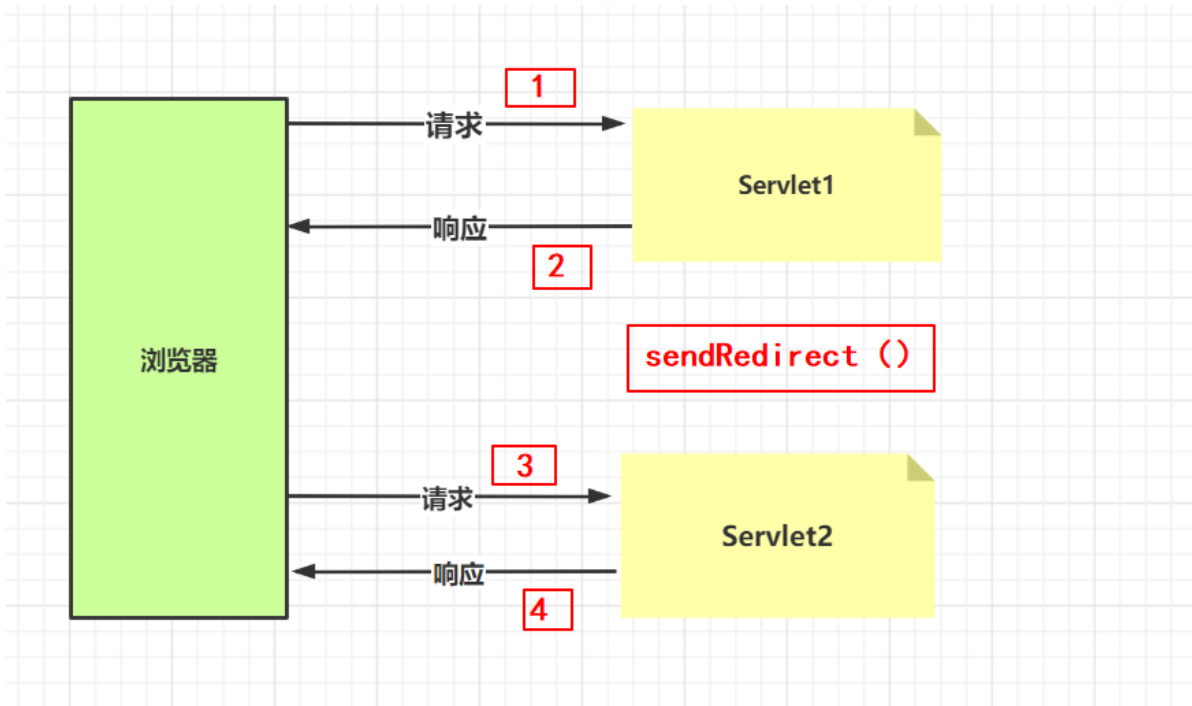
- 首先客户浏览器发送http请求，当web服务器接受后发送302状态码响应及对应新的location给客户浏览器。
- 客户浏览器发现是302响应，则自动再发送一个新的http请求，请求url是新的location地址，服务器根据此请求寻找资源并发送给客户。

6.1.2 重定向的实现

- 实现重定向需要借助javax.servlet.http.HttpServletResponse接口中的以下方法：

方法声明	作用
void sendRedirect(String location)	使用指定的重定向位置URL向客户端发送临时重定向响应

6.1.3 重定向的原理



6.1.4 重定向的特点

- 重定向之后，浏览器地址栏的URL会发生改变。
- 重定向过程中会将前面Request对象销毁，然后创建一个新的Request对象。
- 重定向的URL可以是其它项目工程。

6.1.5 代码示例

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>重定向跳转</title>
</head>
<body>
<form action="redirectServlet" method="post">
  <input type="submit" value="重定向">
</form>
</body>
</html>
```

服务端

```
package cn.guardwhy.demo01;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet("/redirectServlet")
public class RedirectServlet extends HttpServlet {
```

```

@Override
protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
    System.out.println("接收到了浏览器的请求");
    // 1.重定向，给浏览器发送一个新的位置
    response.sendRedirect(request.getContextPath() + "/test01.html");
}

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

```

页面端

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>重定向后的页面</title>
</head>
<body>
    <h3>服务器重新指定位置后的页面</h3>
</body>
</html>

```

执行结果



服务器重新指定位置后的页面

6.2 转发

6.2.1 转发的概念

一个Web组件（Servlet/JSP）将未完成的处理通过容器转交给另外一个Web组件继续处理，转发的各个组件会共享Request和Response对象。

6.2.2 转发的实现

- 绑定数据到Request对象

方法声明	作用
Object getAttribute(String name)	将指定属性值作为对象返回，若给定名称属性不存在，则返回空值。
void setAttribute(String name, Object o)	在此请求中存储属性值。

- 获取转发器对象

方法声明	作用
RequestDispatcher getRequestDispatcher(String path)	返回一个RequestDispatcher对象，该对象充当位于给定路径上的资源的包装器

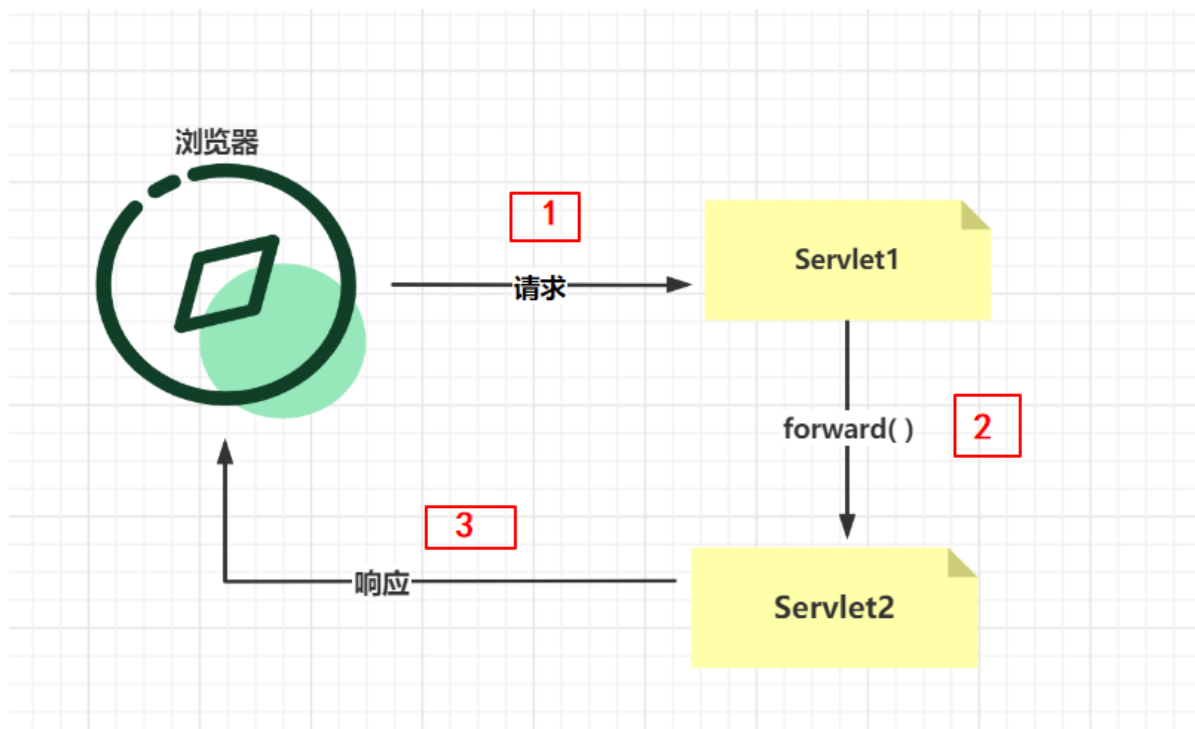
- 转发操作

方法声明	作用
void forward(ServletRequest request, ServletResponse response)	将请求从一个servlet转发到服务器上的另一个资源（Servlet、JSP文件或HTML文件）

6.2.3 转发的特点

- 转发之后浏览器地址栏的URL不会发生改变。
- 转发过程中共享Request对象。
- 转发的URL不可以是其它项目工程。

6.2.4 转发的原理



6.2.5 代码示例

前端页面

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>转发跳转</title>
</head>
<body>
<form action="forwardServlet01" method="post">
    <input type="submit" value="转发">
</form>
</body>
</html>
```

服务端

```
package cn.guardwhy.demo02;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet("/forwardServlet01")
public class ForwardServlet01 extends HttpServlet {
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        System.out.println("接收到了浏览器的请求");
        // 1.向请求域中添加键值对
        request.setAttribute("username", "kobe");
        // 2.转发,也就是让web组件将任务转交给另外一个web组件
        request.getRequestDispatcher("/forwardServlet02").forward(request,
response);
    }

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

```
package cn.guardwhy.demo02;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
```

```

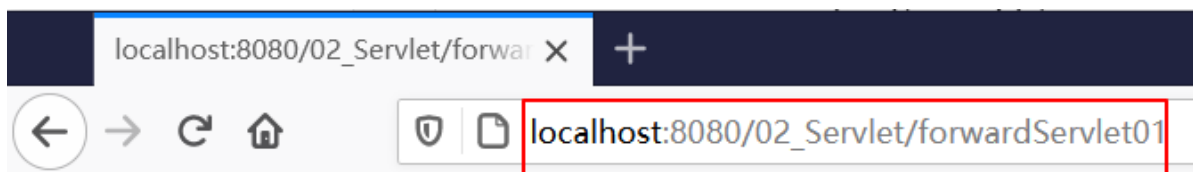
import java.io.PrintWriter;

@WebServlet("/forwardServlet02")
public class ForwardServlet02 extends HttpServlet {
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        System.out.println("数据转发过来了");
        // 1.从请求域中取出用户名
        String username = (String) request.getAttribute("username");
        System.out.println("获取的用户名:" + username);
        // 2.设置编码
        response.setContentType("text/html;charset=utf-8");
        PrintWriter out = response.getWriter();
        out.print("从请求域中取出用户:" + username);
        out.write("<h3>转发成功...</h3>");
    }

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

```

执行结果



从请求域中取出用户:kobe

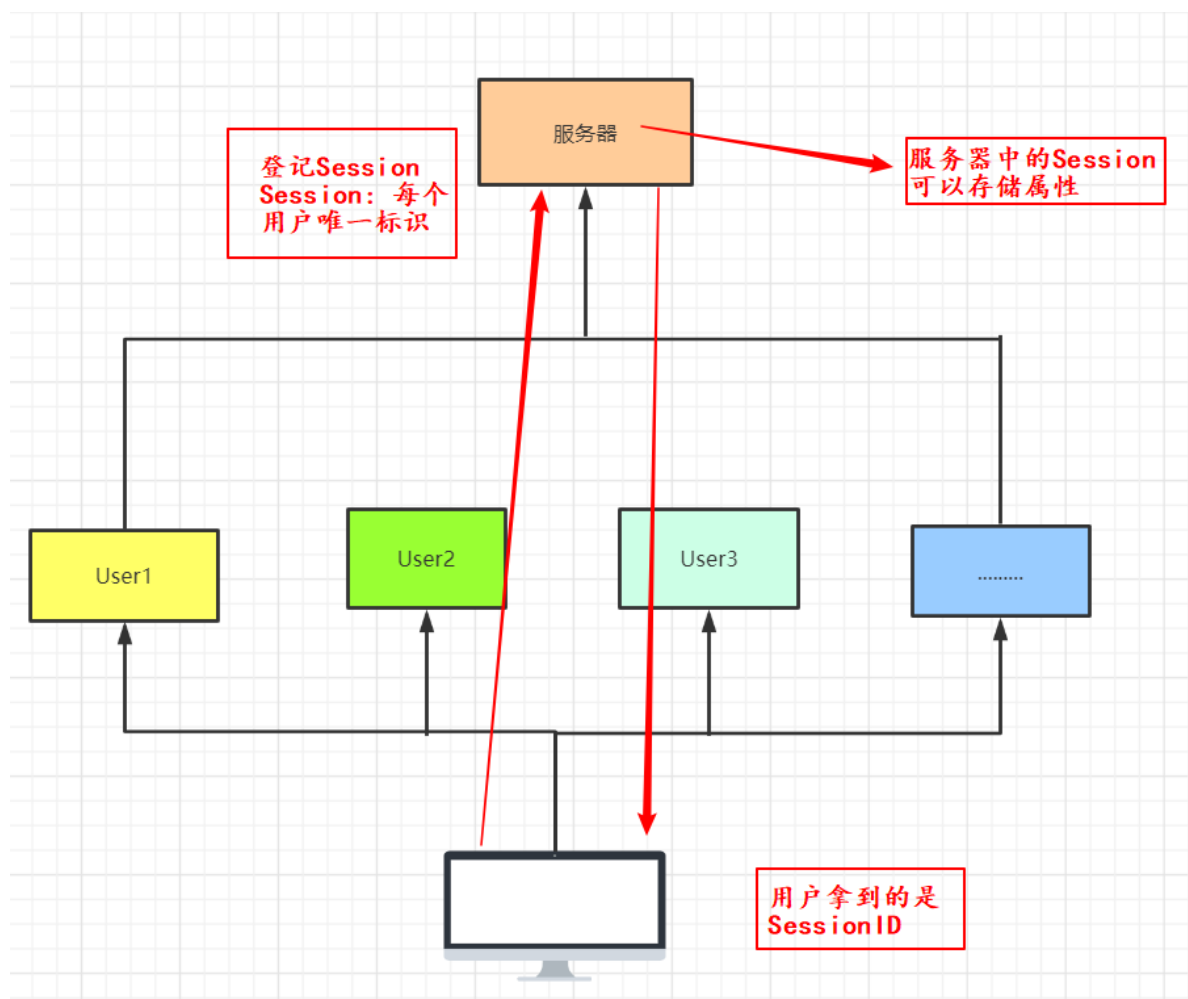
转发成功...

6.2.6 重定向和转发的区别

区别	转发forward()	重定向sendRedirect()
地址栏	不会变化	会显示新的地址
哪里跳转	服务器端	浏览器端
请求域中数据	不会丢失	会丢失
根目录	http://localhost:8080/ 项目名	http://localhost:8080

7- Session技术

7.1.1 基本概念



- Session本意为"会话"的含义，是用来维护一个客户端和服务端关联的一种技术。浏览器访问服务器时，服务器会为每一个浏览器都在服务器端的内存中分配一个空间，用于创建一个Session对象。
- 该对象有一个id属性且该值唯一，称为SessionId，并且服务器会将这个SessionId以Cookie方式发送给浏览器存储。
- 浏览器再次访问服务器时会把SessionId发送给服务器，服务器可以依据SessionId查找相对应的Session对象。

7.1.2 相关的方法

- 使用`javax.servlet.http.HttpServletRequest`接口的成员方法实现Session的获取。

方法声明	作用
<code>HttpSession getSession()</code>	返回此请求关联的当前Session，若此请求没有则创建一个

- 使用`javax.servlet.http.HttpSession`接口的成员方法实现判断和获取。

方法声明	作用
<code>boolean isNew()</code>	判断是否为新创建的Session

代码示例


```

package cn.guardwhy.demo01;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import java.io.IOException;

@WebServlet(name = "SessionServlet1", urlPatterns = "/session1")
public class SessionServlet1 extends HttpServlet {
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        // 1.获取Session对象,得到会话域
        HttpSession session = request.getSession();
        // 2.判断Session对象是否为新建的对象
        System.out.println(session.isNew() ? "新创建的Session对象": "已有的Session对
象");
        // 3.获取编号并且打印
        String id = session.getId();
        System.out.println("获取到的Session编号:" + id);
    }

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

```

执行结果

```

10-Jan-2021 21:24:25.144 INFO [RMI TCP Connection(3)-127.0.0.1] o
[2021-01-10 09:24:25,169] Artifact 03-Session:war exploded: Artif
[2021-01-10 09:24:25,169] Artifact 03-Session:war exploded: Deplo
10-Jan-2021 21:24:34.546 INFO [localhost-startStop-1] org.apache.
10-Jan-2021 21:24:34.572 INFO [localhost-startStop-1] org.apache.
已有的Session对象
获取到的Session编号:81D427569D3504F60D20CABAC547EF59
获取到失效时间:1800
获取失效时间:1000

```

- 使用javax.servlet.http.HttpSession接口的成员方法实现属性的管理。

方法声明	作用
Object getAttribute(String name)	返回在此会话中用指定名称绑定的对象，如果没有对象在该名称下绑定，则返回空值。
void setAttribute(String name, Object value)	使用指定的名称将对象绑定到此会话。
void removeAttribute(String name)	从此会话中删除与指定名称绑定的对象。

代码示例

```
package cn.guardwhy.demo01;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import java.io.IOException;

@WebServlet(name = "SessionServlet2", urlPatterns = "/session2")
public class SessionServlet2 extends HttpServlet {
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // 1.获取Session对象,得到会话域
        HttpSession session = request.getSession();
        // 2.会话域中存入属性名和属性值
        session.setAttribute("username", "guardwhy");
        // 3.从会话域中取出对应的属性值
        System.out.println("属性值:" + session.getAttribute("username"));
        // 4.删除指定的属性名
        session.removeAttribute("username");
        // 5.从会话域中取出对应的属性值
        System.out.println("属性值:" + session.getAttribute("username"));
    }

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

执行结果

The screenshot shows the IDE's 'Services' and 'Output' tabs. In the 'Services' tab, 'Tomcat 8 [local]' is listed under 'Running'. In the 'Output' tab, the console output is visible, showing the result of the application's execution: '属性值:guardwhy' (Attribute value: guardwhy) and '属性值:null' (Attribute value: null). The first line is highlighted with a red box.

7.1.3 Session的生命周期

- 为了节省服务器内存空间资源，服务器会将空闲时间过长的Session对象自动清除掉，服务器默认的超时限制一般是30分钟。
- 使用javax.servlet.http.HttpSession接口的成员方法实现失效实现的获取和设置。

方法声明	作用
int getMaxInactiveInterval()	获取失效时间。
void setMaxInactiveInterval(int interval)	设置失效时间

代码示例

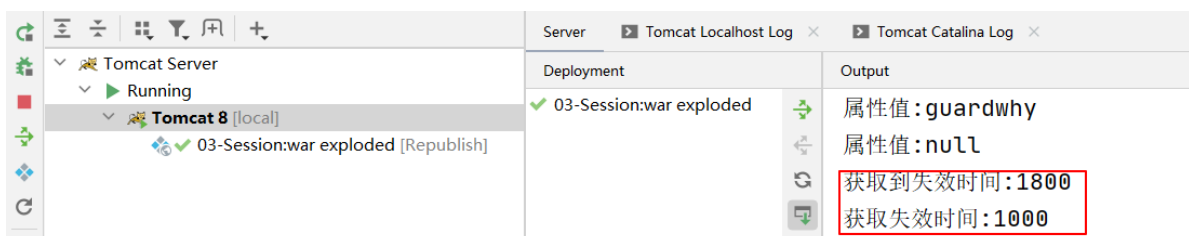
```
package cn.guardwhy.demo01;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import java.io.IOException;

@WebServlet(name = "SessionServlet3", urlPatterns = "/session3")
public class SessionServlet3 extends HttpServlet {
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // 1.获取Session对象,得到会话域
        HttpSession session = request.getSession();
        // 2.获取对象的默认失效时间
        int maxInactiveInterval = session.getMaxInactiveInterval();
        System.out.println("获取到失效时间:" + maxInactiveInterval);    // 1800
        // 3.修改失效时间
        session.setMaxInactiveInterval(1000);
        maxInactiveInterval = session.getMaxInactiveInterval();
        System.out.println("获取失效时间:" + maxInactiveInterval);    // 1000
    }

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

执行结果



- 配置web.xml文件修改失效时间

```
<session-config>
    <!--分钟-->
    <session-timeout>10</session-timeout>
</session-config>
```

7.1.4 Session的特点

- 数据比较安全。
- 能够保存的数据类型丰富，而Cookie只能保存字符串。
- 能够保存更多的数据，而Cookie大约保存4KB。
- 数据保存在服务器端会占用服务器的内存空间，如果存储信息过多、用户量过大，会严重影响服务器的性能。

7.1.5 Session和cookie的区别

- Cookie是把用户的数据写给用户的浏览器，浏览器保存（可以保存多个），Session对象由服务创建。
- Session把用户的数据写到用户独占Session中，服务器端保存（保存重要的信息，减少服务器资源的浪费）

8-EL表达式和JSTL技术

8.1 EL表达式

8.1.1 基本定义

EL（Expression Language）表达式提供了在JSP中简化表达式的方法，可以方便地访问各种数据并输出。

区别	JSP表达式	EL表达式
语法	<%=变量名或表达式%>	\${变量名或表达式}
输出哪里的值	是脚本变量值	作用域中值，如果要使用EL取出变量的值必须先将变量保存在作用域中。

8.1.2 主要功能

- 依次访问pageContext、request、session和application作用域对象存储的数据。
- 获取请求参数值。访问Bean对象的属性。
- 访问集合中的数据。输出简单的运算结果。

8.1.3 内置对象

访问方式

```
<%=request.getAttribute(" varName")%>
用EL表达式实现：${ varName }
```

代码示例

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

```

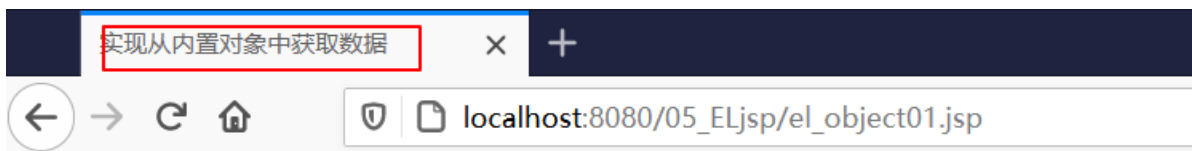
<html>
<head>
    <title>内置对象中获取数据</title>
</head>
<body>
<%
    pageContext.setAttribute("username1", "pageContext对象中的属性值:Kobe");
    request.setAttribute("username2", "request对象中的属性值:Curry");
    session.setAttribute("username3", "session对象中的属性值:James");
    application.setAttribute("username4", "session对象中的属性值:Harden");
%>

<!--使用JSP中原始方式获取数据-->
<!--
<%= "nam1的数值为:" + pageContext.getAttribute("username1")%><br/>
<%= "nam2的数值为:" + request.getAttribute("username2")%><br/>
<%= "nam3的数值为:" + session.getAttribute("username3")%><br/>
<%= "nam4的数值为:" + application.getAttribute("username4")%><br/>
--%>

<!--使用EL表达式实现获取数据-->
<h3>username1的数值为:${username1}</h3>
<h3>username2的数值为:${username2}</h3>
<h3>username3的数值为:${username3}</h3>
<h3>username4的数值为:${username4}</h3>
</body>
</html>

```

执行结果



username1的数值为:pageContext对象中的属性值:Kobe

username2的数值为:request对象中的属性值:Curry

username3的数值为:session对象中的属性值:James

username4的数值为:session对象中的属性值:Harden

8.1.4 请求参数数据

- 在EL之前使用下列方式访问请求参数的数据

```

request.getParameter(name);
request.getParameterValues(name);

```

- 在EL中使用下列方式访问请求参数的数据

param: 接收的参数只有一个值。
paramValues: 接受的参数有多个值。

代码示例

传递参数

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>传递参数</title>
</head>
<body>
<form action="el_param02.jsp" method="post">
    用户名: <input type="text" name="name"/><br/>
    爱好: <input type="checkbox" name="hobby" value="Java"/>Java<br/>
    <input type="checkbox" name="hobby" value="Vue.js"/>Vue.js<br/>
    <input type="checkbox" name="hobby" value="Python"/>Python<br/>
    <input type="submit" value="提交"/><br/>
</form>
</body>
</html>
```

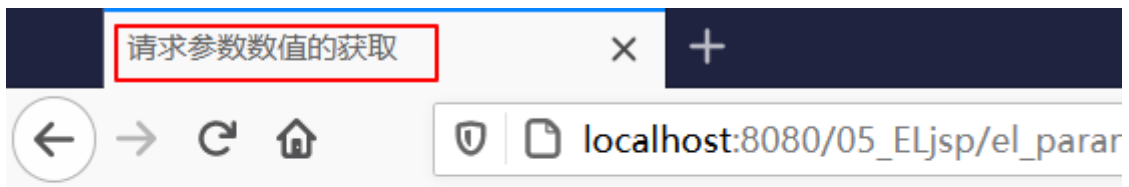
请求参数数值的获取

```
<%@ page import="java.util.Arrays" %>
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>请求参数数值的获取</title>
</head>
<body>
<!--设置编码-->
<%
    request.setCharacterEncoding("utf-8");
%>
<!--原始方式获取请求参数值 --%>

<!--
<%= "用户名:" + request.getParameter("name") %><br/>
<%= "爱好:" + Arrays.toString(request.getParameterValues("hobby")) %><br/>
--%>

<!--2.使用EL表达式获取参数值--%>
用户名: ${param.name}<br/>
爱好: ${paramValues.hobby[0]}<br/>
</body>
</html>
```

执行结果



用户名: guardwhy
爱好: Java

8.1.5 对象属性获取

代码示例

```
<%@ page import="cn.guardwhy.pojo.Person" %>
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>Bean对象中属性的获取</title>
</head>
<body>
<%
    Person person = new Person();
    person.setName("Curry");
    person.setAge(32);

    pageContext.setAttribute("person", person);

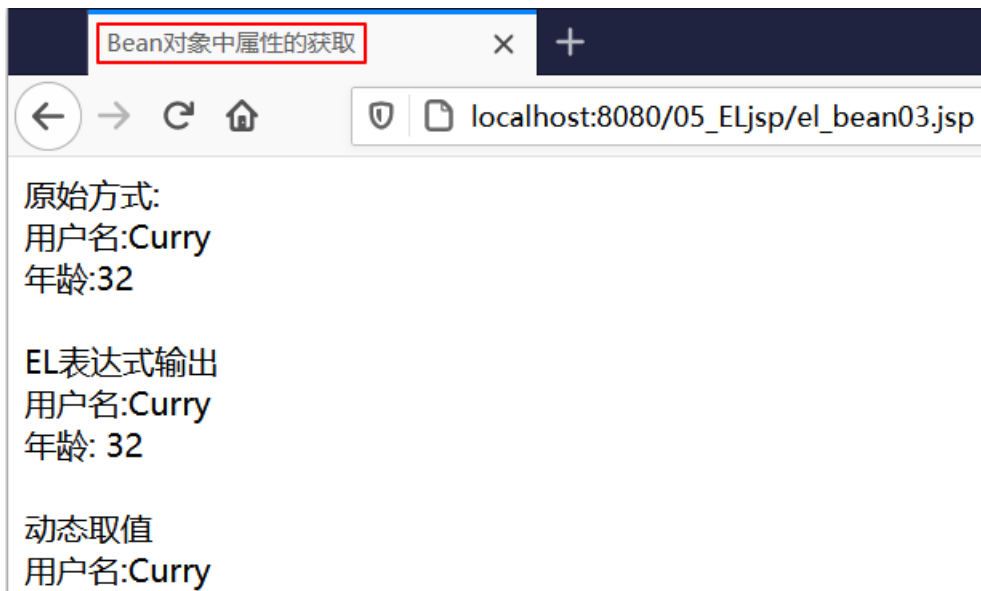
    pageContext.setAttribute("let1", "name");
    pageContext.setAttribute("let2", "age");
%>

<!-- 1. 原始方式输出属性 -->
<%= "原始方式:" %><br/>
<%= "用户名:" + person.getName() %><br/>
<%= "年龄:" + person.getAge() %><br/><br/>

<!-- 2. EL表达式输出 -->
<%= "EL表达式输出" %><br/>
<!--
    用户名: ${person.name}<br/>
    年龄: ${person.age}<br/>
-->
    用户名: ${person["name"]} <br/>
    年龄: ${person["age"]} <br/><br/>

<!-- 3. 动态取值 -->
<%= "动态取值" %><br/>
    用户名: ${person[let1]}
</body>
</html>
```

执行结果



8.1.6 获取集合元素

代码示例

```
<%@ 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" %>
<html>
<head>
    <title>获取集合中数据内容</title>
</head>
<body>
<%
// 1.创建集合添加元素
List<String> list = new ArrayList<>();
list.add("curry");
list.add("kobe");
list.add("james");
// 2.将整个集合放入到指定内置对象中
pageContext.setAttribute("list", list);

%>
<!-- EL表达式获取集合元素 -->
下标为0元素:${list[0]}<br/>
下标为1元素:${list[1]}<br/>
下标为2元素:${list[2]}<br/>
<hr/>

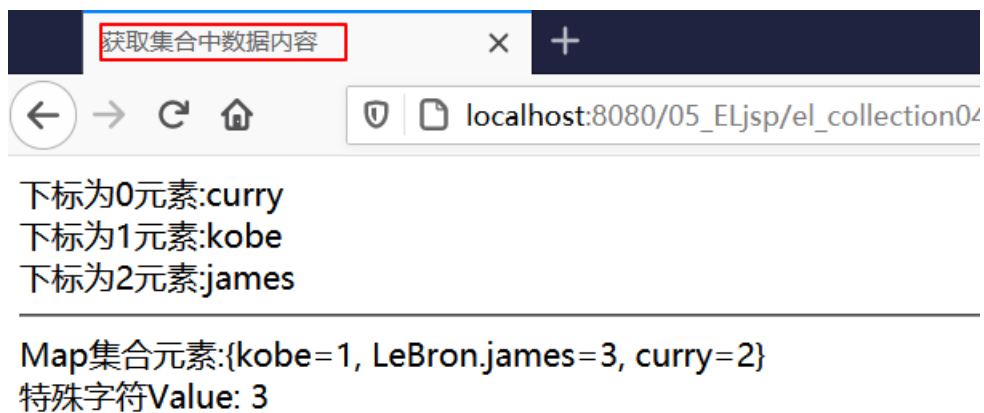
<%
// 1.创建map集合添加元素
Map<String, Integer> map = new HashMap<>();
map.put("kobe", 1);
map.put("curry", 2);
map.put("LeBron.james", 3);
// 2.将整个集合放入到指定内置对象中
pageContext.setAttribute("map", map);

%>
<!-- EL表达式获取Map集合元素-->
```



```
Map集合元素:${map}<br/>
特殊字符value: ${map["LeBron.james"]}<br/>
</body>
</html>
```

执行结果



8.1.7 常用的内置对象

类别	标识符	基本描述
JSP	pageContext	PageContext 处理当前页面
页面域	pageScope	同页面作用域属性名称和值有关的Map类
请求域	requestScope	同请求作用域属性的名称和值有关的Map类
会话域	sessionScope	同会话作用域属性的名称和值有关的Map类
上下文域	applicationScope	同应用程序作用域属性的名称和值有关的Map类
请求参数	param	根据名称存储请求参数的值的Map类
	paramValues	把请求参数的所有值作为一个String数组来存储的Map类

8.2 JSTL标签

8.2.1 基本概念

- JSTL(JSP Standard Tag Library) 被称为JSP标准标签库。
- 可以利用这些标签取代JSP页面上的Java代码，从而提高程序的可读性，降低程序的维护难度。

8.2.2 导入相关依赖

```

<!-- JSTL表达式的依赖 -->
<dependency>
    <groupId>javax.servlet.jsp.jstl</groupId>
    <artifactId>jstl-api</artifactId>
    <version>1.2</version>
</dependency>

<!-- standard标签库 -->
<dependency>
    <groupId>taglibs</groupId>
    <artifactId>standard</artifactId>
    <version>1.1.2</version>
</dependency>

```

8.2.3 常用核心标签

输出标签

`<c:out></c:out>` 用来将指定内容输出的标签

```

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>输出标签</title>
</head>
<body>
<!--输出标签-->
<c:out value="hello jsp"></c:out>
</body>
</html>

```

设置标签

`<c:set></c:set>` 用来设置属性范围值的标签

```

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>set标签设置</title>
</head>
<body>
<!--设置属性--%>
<c:set var="username" value="Curry" scope="page"></c:set>
<!--使用out标签打印--%>
<c:out value="username:${username}"></c:out><br/>

<!--设置对象属性值并且打印--%>
<jsp:useBean id="person" class="cn.guardwhy.pojo.Person" scope="page">
</jsp:useBean>
<c:set property="name" value="guardwhy" target="${person}"></c:set>
<c:set property="age" value="26" target="${person}"></c:set>

```

```
<c:out value="username:${person.name}"></c:out><br/>
<c:out value="age:${person.age}"></c:out>
</body>
</html>
```

单条件判断标签

```
<c:if test="EL条件表达式">
    满足条件执行
</c:if>
```

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>if标签</title>
</head>
<body>
<!--设置变量--%>
<c:set var="age" value="21" scope="page"></c:set>
<c:out value="年龄是:${age}"></c:out><br/>

<!--判断年龄是否成年--%>
<c:if test="${age >=18}">
    <!--输出结果--%>
    <c:out value="恭喜你，正式长大了"></c:out>
</c:if>
</body>
</html>
```

remove标签

`<c:remove></c:remove>` 用来删除指定数据的标签

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>remove标签</title>
</head>
<body>
<!--设置属性--%>
<c:set var="username" value="guardwhy" scope="page"></c:set>
<c:out value="username:${username}"></c:out>
<hr/>

<!--删除属性--%>
<c:remove var="username" scope="page"></c:remove>
<c:out value="username:${username}" default="NULL"></c:out>
</body>
</html>
```

多条件判断标签

```

<c:choose >
<c:when test ="EL表达式">
    满足条件执行
    </c:when>
...
<c:otherwise>
    不满足上述when条件时执行
</c:otherwise>
</c:choose>

```

```

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>实现choose标签</title>
</head>
<body>
<!-- 设置变量并指定数值 --%>
<c:set var="age" value="17" scope="page"></c:set>
<c:out value="age:${age}"></c:out>
<hr/>

<!--进行多条件判断--%>
<c:choose>
    <c:when test="${age > 18}">
        <c:out value="恭喜你，已经成年了。可以好好的happy了"></c:out>
    </c:when>
    <c:when test="${age == 18}">
        <c:out value="才刚刚18岁，好好学习"></c:out>
    </c:when>
    <c:otherwise>
        <c:out value="未成年，晚上早点休息"></c:out>
    </c:otherwise>
</c:choose>
</body>
</html>

```

常用函数标签

```

<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

```

```

<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>常用函数标签</title>
</head>
<body>
<%
    pageContext.setAttribute("let", "Hello JSP");
%>
原生字符串: ${let}<br/>
判断字符串是否包含指定字符串:${fn:contains(let,"Hello")}<br/>
将原生字符串转换为大写:${fn:toUpperCase(let)}<br/>
将原生字符串转换为小写:${fn:toLowerCase(let)}<br/>

```

```
</body>
</html>
```

循环标签

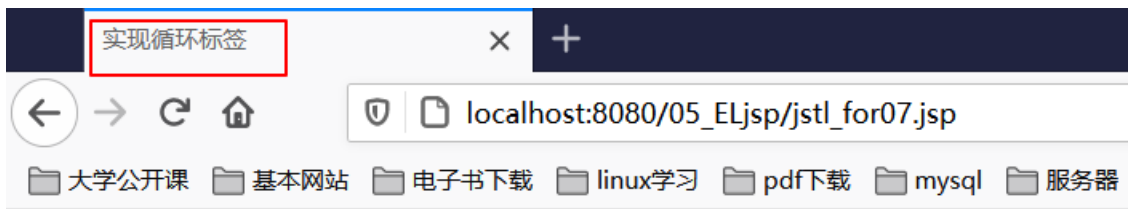
```
<c:forEach var="循环变量" items="集合">
...
</c:forEach>
```

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>实现循环标签</title>
</head>
<body>
<%
// 1. 定义数组
String[] array = {"12", "21", "31", "28", "56"};
// 2. 设置值
pageContext.setAttribute("array", array);
%>
<!-- 循环遍历数组中的所有元素 --%>
<c:out value="排序后:"></c:out>
<c:forEach var="arr" items="${array}">
    <c:out value="${arr}"></c:out>
</c:forEach>
<hr/>

<!-- 跳跃性遍历间隔为2 --%>
<c:out value="排序后(跳跃遍历):"></c:out>
<c:forEach var="arr" items="${array}" step="2">
    <c:out value="${arr}"></c:out>
</c:forEach>
<hr/>

<!-- 指定起始和结尾位置 从下标1开始到3结束 --%>
<c:out value="排序后(指定位置):"></c:out>
<c:forEach var="arr" items="${array}" begin="1" end="3">
    <c:out value="${arr}"></c:out>
</c:forEach>
</body>
</html>
```

执行结果



排序后: 12 21 31 28 56

排序后(跳跃遍历): 12 31 56

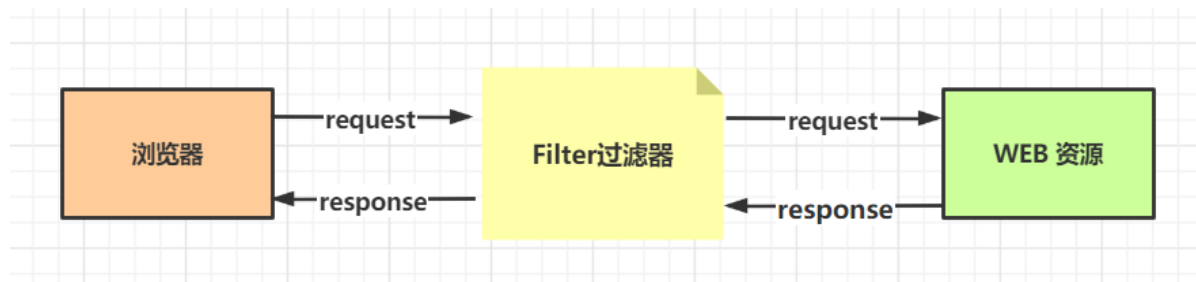
排序后(指定位置): 21 31 28

9- Filter过滤器

9.1 基本概念

- Filter本意为“过滤”的含义，是JavaWeb的三大组件之一，三大组件为：Servlet、Filter、Listener。
- 过滤器是向 Web 应用程序的请求和响应处理添加功能的 Web 服务组件。
- 过滤器相当于浏览器与Web资源之间的一道过滤网，在访问资源之前通过一系列的过滤器对请求进行修改、判断以及拦截等，也可以对响应进行修改、判断以及拦截等。

9.2 Filter原理



9.2 Filter基本使用

9.2.1 login.jsp

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>登录页面</title>
</head>
<body>
<form action="login" method="post">
    用户姓名: <input type="text" name="userName"><br/>
    用户密码: <input type="password" name="password"><br/>
    <input type="submit" value="注册"/>
</form>
</body>
</html>
```

9.2.2 LoginServlet

```
package cn.guardwhy.servlet;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

/*
控制器:Servlet
*/
@WebServlet(name = "LoginServlet", urlPatterns = "/login")
public class LoginServlet extends HttpServlet {
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        // 1.获取请求中的用户名和密码
        String userName = request.getParameter("userName");
        System.out.println("获取到的用户名:" + userName);
        String password = request.getParameter("password");
        System.out.println("获取到的密码:" + password);
        // 2.用户名和密码校验
        if("Curry".equals(userName) && "1234".equals(password)){
            System.out.println("登录成功,欢迎使用");
            // 3.存储用户信息
            request.getSession().setAttribute("userName", userName);
            // 4.重定向
            response.sendRedirect("main.jsp");
        }else {
            System.out.println("用户名或者密码错误,请重新输入...");
            // 5.转发
            request.getRequestDispatcher("login.jsp").forward(request,
response);
        }
    }

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

9.2.3 main.jsp

```

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>页面主题</title>
</head>
<body>
<h3>登录成功, 欢迎${sessionScope.username}使用</h3>
</body>
</html>

```

9.2.4 LoginFilter

```

package cn.guardwhy.filter;

import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import java.io.IOException;
/*
拦截器
*/
public class LoginFilter implements Filter {
    @Override
    // web服务器启动, 过滤器开始初始化, 随时等待过滤对象出现。
    public void init(FilterConfig filterConfig) throws ServletException {
        System.out.println("过滤器开始初始化");
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
                        FilterChain filterChain) throws IOException,
    ServletException {
        // 0. 设置编码
        request.setCharacterEncoding("utf-8");
        response.setCharacterEncoding("utf-8");
        response.setContentType("text/html; charset=UTF-8");

        // 1. 获取session对象
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        HttpSession session = httpRequest.getSession();
        // 2. 得到用户名
        Object userName = session.getAttribute("userName");
        // 3. 获取Servlet请求路径
        String servletPath = httpRequest.getServletPath();
        // 4. 假设没有登录, 则返回登录页面
        if(null == userName && !servletPath.contains("login") ){
            // 转发
            request.getRequestDispatcher("login.jsp").forward(request,
response);
        }else{
            // 5. 登录则放行
            filterChain.doFilter(request, response);
        }
    }

    @Override

```

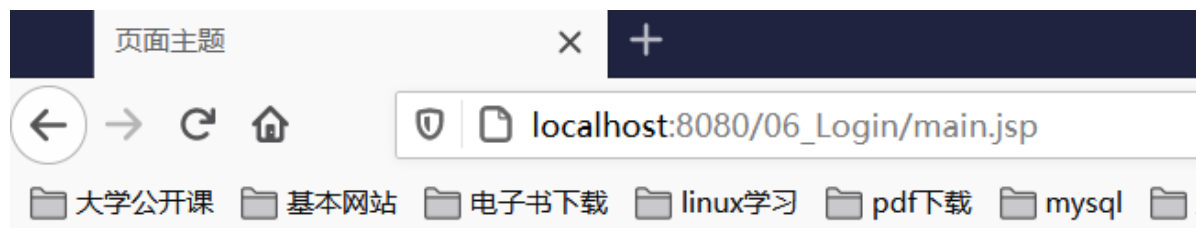


```
// web服务器关闭的时候,过滤器销毁
public void destroy() {
    System.out.println("过滤器销毁。。");
}
}
```

9.2.5 web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
    version="4.0">
    <!--过滤器-->
    <filter>
        <filter-name>LoginServlet</filter-name>
        <filter-class>cn.guardwhy.filter.LoginFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>LoginServlet</filter-name>
        <url-pattern>/main.jsp</url-pattern>
        <!--<url-pattern>/*</url-pattern>-->
    </filter-mapping>
</web-app>
```

执行结果



登录成功,欢迎使用

9.3 Filter接口

9.3.1 基本概念

javax.servlet.Filter接口主要用于描述过滤器对象，可以对资源的请求和资源的响应操作进行筛选操作。

9.3.2 常用的方法

方法声明	作用
void init(FilterConfig filterConfig)	实现过滤器的初始化操作。
void doFilter(ServletRequest request, ServletResponse response,FilterChain chain)	执行过滤操作的功能。
void destroy()	实现过滤器的销毁操作。

9.4 FilterConfig接口

9.4.1 基本概念

javax.servlet.FilterConfig接口主要用于描述过滤器的配置信息。

9.4.2 常用方法

方法声明	作用
String getFilterName()	获取过滤器的名字。
String getInitParameter(String name)	获取指定的初始化参数信息。
Enumeration getInitParameterNames()	获取所有的初始化操作名称
ServletContext getServletContext()	获取ServletContext对象。

代码示例

```
package cn.guardwhy.filter;

import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import java.io.IOException;
import java.util.Enumeration;

/*
拦截器
*/
public class LoginFilter implements Filter {
    @Override
    // web服务器启动,过滤器开始初始化, 随时等待过滤对象出现。
    public void init(FilterConfig filterConfig) throws ServletException {
        System.out.println("初始化操作正在火热进行中...");
        System.out.println("获取到的过滤器名称为: " + filterConfig.getFilterName());
        String userName = filterConfig.getInitParameter("userName");
        System.out.println("获取到指定初始化参数的数值为: " + userName);
        Enumeration<String> initParameterNames =
filterConfig.getInitParameterNames();
        while (initParameterNames.hasMoreElements()) {
            System.out.println("获取到的初始化参数名为: " +
initParameterNames.nextElement());
        }
    }
}
```

```

        ServletContext servletContext = filterConfig.getServletContext();
        System.out.println("获取到的上下文对象是: " + servletContext);
    }

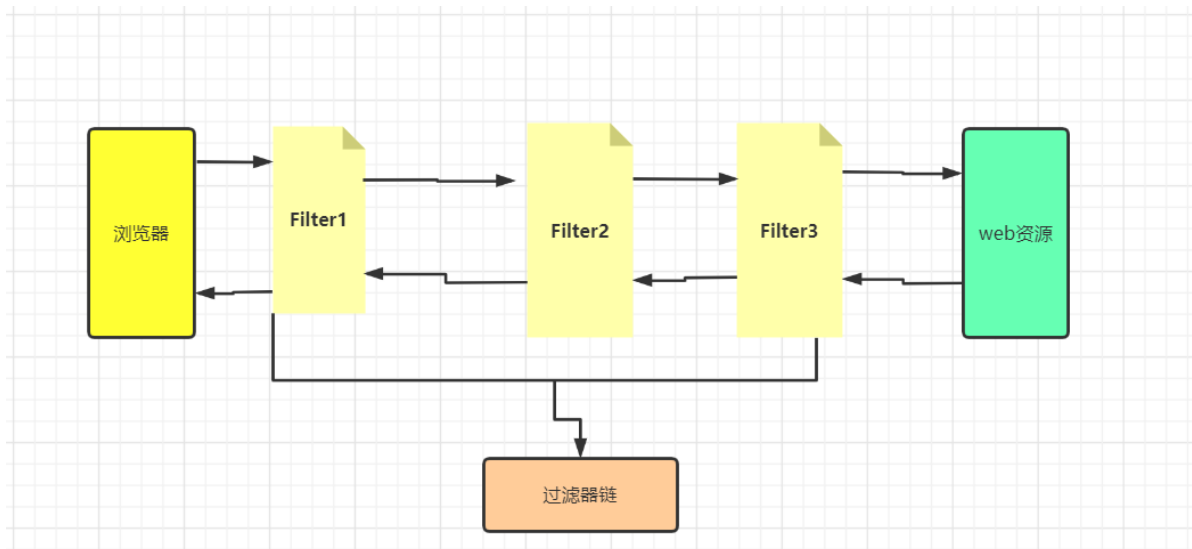
    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain filterChain) throws IOException,
        ServletException {
    }

    @Override
    public void destroy() {
    }
}

```

9.5 多个过滤器

如果有多个过滤器都满足过滤的条件，则容器依据映射的先后顺序来调用各个过滤器。



10- Listener监听器

10.1 基本概念

- Servlet规范中定义的一种特殊的组件，用来监听Servlet容器产生的事件并进行相应的处理。
- 底层原理是采用接口回调的方式实现。

10.2 基本分类

监听器类型	作用
javax.servlet.ServletRequestListener	监听request作用域的创建和销毁
javax.servlet.ServletRequestAttributeListener	监听request作用域的属性状态变化
javax.servlet.http.HttpSessionListener	监听session作用域的创建和销毁
javax.servlet.http.HttpSessionAttributeListener	监听session作用域的属性状态变化
javax.servlet.ServletContextListener	监听application作用域的创建和销毁
javax.servlet.ServletContextAttributeListener	监听application作用域的属性状态变化
javax.servlet.http.HttpSessionBindingListener	监听对象与session的绑定和解除
javax.servlet.http.HttpSessionActivationListener	监听session数值的钝化和活化

10.3 监听器详解

10.3.1 ServletRequestListener

- 在ServletRequest创建和关闭时都会通知ServletRequestListener监听器。

常用方法

方法类型	作用
void requestInitialized(ServletRequestEvent sre)	实现ServletRequest对象的初始化
void requestDestroyed(ServletRequestEvent sre)	实现ServletRequest对象的销毁

代码示例

RequestListener

```
package cn.guardwhy.listener;

import javax.servlet.ServletRequestEvent;
import javax.servlet.ServletRequestListener;

public class RequestListener01 implements ServletRequestListener {
    @Override
    public void requestDestroyed(ServletRequestEvent servletRequestEvent) {
        System.out.println("请求销毁了....");
    }

    @Override
    public void requestInitialized(ServletRequestEvent servletRequestEvent) {
        System.out.println("创建请求...");
    }
}
```

10.3.2 ServletRequestAttributeListener

- 向ServletRequest添加、删除或者替换一个属性的时候，将会通知ServletRequestAttributeListener监听器。

常用方法

方法类型	作用
void attributeAdded(ServletRequestAttributeEvent srae)	增加属性时触发
void attributeReplaced(ServletRequestAttributeEvent srae)	修改属性时触发
void attributeRemoved(ServletRequestAttributeEvent srae)	删除属性时触发

代码示例

RequestAttributeListener

```
package cn.guardwhy.listener;

import javax.servlet.ServletRequestAttributeEvent;
import javax.servlet.ServletRequestAttributeListener;

public class RequestAttributeListener02 implements
    ServletRequestAttributeListener {
    @Override
    public void attributeAdded(ServletRequestAttributeEvent
        servletRequestAttributeEvent) {
        System.out.println("增加了属性:" +
            servletRequestAttributeEvent.getName());
    }

    @Override
    public void attributeRemoved(ServletRequestAttributeEvent
        servletRequestAttributeEvent) {
        System.out.println("属性:" + servletRequestAttributeEvent.getName() + "被
            删除了");
    }

    @Override
    public void attributeReplaced(ServletRequestAttributeEvent
        servletRequestAttributeEvent) {
        System.out.println("修改属性:" + servletRequestAttributeEvent.getName());
    }
}
```

requestAttribute.jsp

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>属性状态的改变</title>
</head>
<body>
<%
// 1. 实现属性的添加
```

```

request.setAttribute("name", "Curry");
// 2. 修改属性修改
request.setAttribute("name", "Kobe");
// 3. 删除属性
request.removeAttribute("name");
%>
</body>
</html>

```

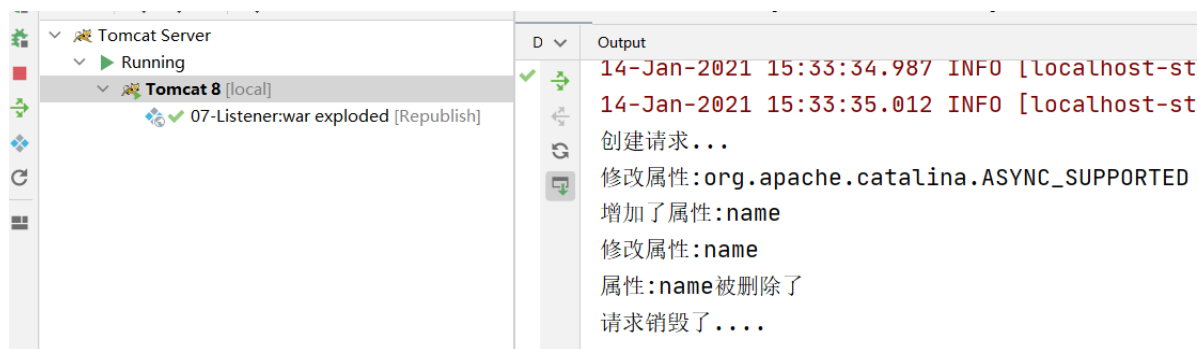
web.xml

```

<listener>
    <listener-class>cn.guardwhy.listener.RequestAttributeListener02</listener-
class>
</listener>

```

执行结果



10.3.3 HttpSessionListener

当一个HttpSession刚被创建或者失效（invalidate）的时候，将会通知HttpSessionListener监听器。

常用方法

方法类型	作用
void sessionCreated(HttpSessionEvent se)	当一个HttpSession对象被创建时会调用这个方法
void sessionDestroyed(HttpSessionEvent se)	当一个HttpSession超时或者调用HttpSession的 invalidate()方法让它销毁时，将会调用这个方法

SessionListener

```

package cn.guardwhy.listener;

import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;

public class SessionListener03 implements HttpSessionListener {
    @Override
    public void sessionCreated(HttpSessionEvent se) {
        System.out.println("创建了session");
    }
}

```

```

@Override
public void sessionDestroyed(HttpSessionEvent se) {
    System.out.println("销毁了session");
}
}

```

web.xml

```

<!--监听器-->
<listener>
    <listener-class>cn.guardwhy.listener.SessionListener03</listener-class>
</listener>
<session-config>
    <session-timeout>1</session-timeout>
</session-config>

```

10.3.4 HttpSessionAttributeListener

- HttpSession中添加、删除或者替换一个属性的时候，将会通知HttpSessionAttributeListener监听器。

常用方法

方法类型	作用
void attributeAdded(HttpSessionBindingEvent se)	当往会话中加入一个属性的时候会调用这个方法
void attributeRemoved(HttpSessionBindingEvent se)	当从会话中删除一个属性的时候会调用这个方法
void attributeReplaced(HttpSessionBindingEvent se)	当改变会话中的属性的时候会调用这个方法

代码示例

SessionAttributeListener

```

package cn.guardwhy.listener;

import javax.servlet.http.HttpSessionAttributeListener;
import javax.servlet.http.HttpSessionBindingEvent;

public class SessionAttributeListener04 implements HttpSessionAttributeListener
{
    @Override
    public void attributeAdded(HttpSessionBindingEvent httpSessionBindingEvent)
    {
        System.out.println("增加了属性:" + httpSessionBindingEvent.getName());
    }

    @Override
    public void attributeRemoved(HttpSessionBindingEvent
httpSessionBindingEvent) {
        System.out.println("属性:" + httpSessionBindingEvent.getName() + "删除");
    }
}

```

```

@Override
public void attributeReplaced(HttpSessionBindingEvent
httpSessionBindingEvent) {
    System.out.println("修改属性:" + httpSessionBindingEvent.getName());
}
}

```

requestAttribute.jsp

```

<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>属性状态的改变</title>
</head>
<body>
<%
// 1.实现属性的添加
request.setAttribute("name", "Curry");
// 2.修改属性修改
request.setAttribute("name", "Kobe");
// 3.删除属性
request.removeAttribute("name");
%>
</body>
</html>

```

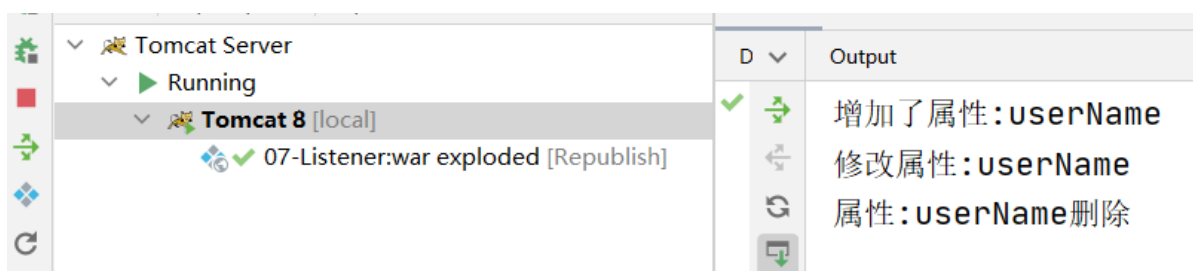
web.xml

```

<listener>
    <listener-class>cn.guardwhy.listener.SessionAttributeListener04</listener-
class>
</listener>

```

执行结果



10.3.5 ServletContextListener

在ServletContext创建和关闭时都会通知ServletContextListener监听器。

常用方法

方法类型	作用
void contextInitialized(ServletContextEvent sce)	当ServletContext创建的时候，将会调用这个方法
void contextDestroyed(ServletContextEvent sce)	当ServletContext销毁的时候（例如关闭应用服务器或者重新加载应用），将会调用这个方法

代码示例

```
package cn.guardwhy.listener;

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

public class ContextListener05 implements ServletContextListener {
    @Override
    public void contextInitialized(ServletContextEvent servletContextEvent) {
        System.out.println("ServletContext对象创建");
    }

    @Override
    public void contextDestroyed(ServletContextEvent servletContextEvent) {
        System.out.println("ServletContext对象销毁");
    }
}
```

10.3. 6 ServletContextAttributeListener

向ServletContext添加、删除或者替换一个属性的时候，将会通知ServletContextAttributesListener监听器。

常用方法

方法类型	作用
void attributeAdded(ServletContextAttributeEvent scae)	往ServletContext中加入一个属性的时候触发
void attributeRemoved(ServletContextAttributeEvent scae)	从ServletContext中删除一个属性的时候触发
void attributeReplaced(ServletContextAttributeEvent scae)	改变ServletContext中属性的时候触发

代码示例

ServletContextAttributeListener

```
package cn.guardwhy.listener;
```

```

import javax.servlet.ServletContextAttributeEvent;
import javax.servlet.ServletContextAttributeListener;

public class ContextAttributeListener06 implements
ServletContextAttributeListener {
    @Override
    public void attributeAdded(ServletContextAttributeEvent
servletContextAttributeEvent) {
        System.out.println("增加了属性" + servletContextAttributeEvent.getName());
    }

    @Override
    public void attributeRemoved(ServletContextAttributeEvent
servletContextAttributeEvent) {
        System.out.println("属性" + servletContextAttributeEvent.getName() + "删除!");
    }

    @Override
    public void attributeReplaced(ServletContextAttributeEvent
servletContextAttributeEvent) {
        System.out.println("修改属性" + servletContextAttributeEvent.getName());
    }
}

```

contextAttribute.jsp

```

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>ServletContext对象属性</title>
</head>
<body>
<%
// 1.添加属性
application.setAttribute("userName", "Curry");
// 2.修改属性
application.setAttribute("userName", "Kobe");
// 3.删除属性
application.removeAttribute("userName");
%>
</body>
</html>

```

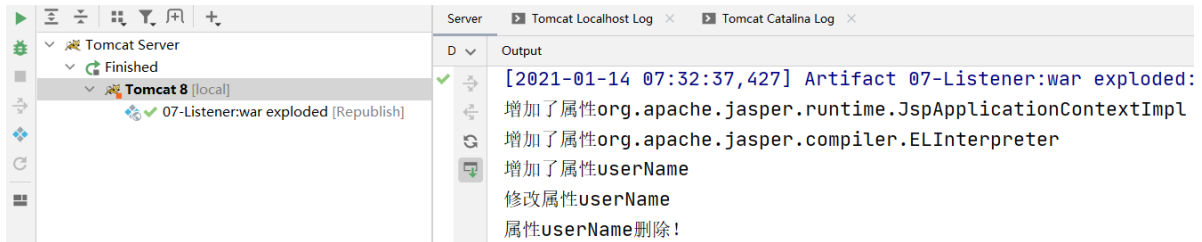
web.xml

```

<listener>
    <listener-class>cn.guardwhy.listener.ContextAttributeListener06</listener-
class>
</listener>

```

执行结果



10.3.7 HttpSessionBindingListener

HttpSession中绑定和解除绑定时，将会通知HttpSessionListener监听器。

方法类型	作用
void valueBound(HttpSessionBindingEvent event)	有对象绑定时调用该方法
void valueUnbound(HttpSessionBindingEvent event)	有对象解除绑定时调用该方法

代码示例

```
package cn.guardwhy.listener;

import javax.servlet.http.HttpSessionBindingEvent;
import javax.servlet.http.HttpSessionBindingListener;

public class Student implements HttpSessionBindingListener {
    private String name;
    private int age;

    public Student() {

    }

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

    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;
    }

    @Override
    public void valueBound(HttpSessionBindingEvent httpSessionBindingEvent) {
```

```

        System.out.println("对象绑定到session中" +
httpSessionBindingEvent.getName());
    }

    @Override
    public void valueUnbound(HttpSessionBindingEvent httpSessionBindingEvent) {
        System.out.println("解除绑定...");
    }
}

```

sessionBind.jsp

```

<%@ page import="cn.guardwhy.listener.Student" %>
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>session中对象的绑定和解除</title>
</head>
<body>
<%
// 创建Student对象
Student student = new Student();
student.setName("Curry");
student.setAge(21);
// 将对象与Session对象进行绑定
session.setAttribute("student", student);
// 解除绑定
session.removeAttribute("student");
%>
</body>
</html>

```

10.4 监听器混合使用

代码示例

OnlineUser

```

package cn.guardwhy.online;

import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;

public class OnlineUser implements HttpSessionListener, ServletContextListener {
    // 1. 声明全局变量
    private ServletContext servletContext = null;
    @Override
    public void contextInitialized(ServletContextEvent servletContextEvent) {
        servletContext = servletContextEvent.getServletContext();
    }

    @Override
    public void contextDestroyed(ServletContextEvent servletContextEvent) {

```

```

        servletContext = null;
    }

    @Override
    public void sessionCreated(HttpSessionEvent httpSessionEvent) {
        System.out.println("新用户上线");
        Object count = servletContext.getAttribute("count");
        // 2.若当前用户为第一,将全局对象中的属性设置为1
        if (null == count){
            servletContext.setAttribute("count", 1);
        }else {
            // 当前用户不是第一个用户,则将全局对象中原有的数据取出来加1后再设置进去
            Integer integer = (Integer) count;
            integer++;
            servletContext.setAttribute("count", integer);
        }
        System.out.println("当前在线用户数量:" +
servletContext.getAttribute("count"));
    }

    @Override
    public void sessionDestroyed(HttpSessionEvent httpSessionEvent) {
        System.out.println("用户已下线...");
    }
}

```

onlineUser.jsp

```

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>当前在线的用户数量</title>
</head>
<body>
<h3>在线用户人数:${applicationScope.count}</h3>
</body>
</html>

```

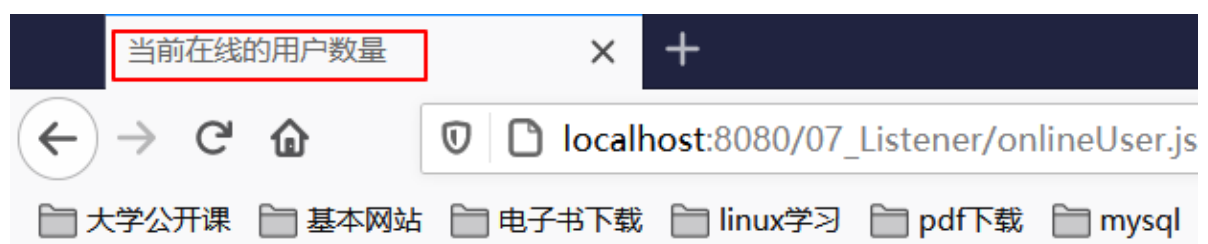
web.xml

```

<listener>
    <listener-class>cn.guardwhy.online.OnlineUser</listener-class>
</listener>

```

执行结果



在线用户人数:2

