Servlet

JavaEE是什么

SUN公司为JAVA程序员提供的一庞大的类库，帮助程序员完成企业级开发

Servlet是什么

- Serv表示服务器
- let表示小程序
- 是SUN公司制定的一套接口，用于处理B/S架构之中的业务

Tomcat是什么

Tomcat是实现了Servlet规范和JSP规范的容器

C/S和B/S模式

C/S　Client/Server　　客户端/服务器

B/S　Browser/Server　　浏览器/服务器，本质上也还是C/S模式，不过客户端比较特殊，是浏览器

B/S的优缺点

优点：

- 不需要安装特定的客户端软件，只需要浏览器就行了，客户体验好
- 升级方便，只需要升级服务端即可

缺点：

- 所有数据在服务器端，一旦发生不可力，数据丢失严重，不安全
- 访问速度取决于网速，网速差时，用户体验不好

C/S端恰巧相反

实现Servlet

1. 将Tomcat加入依赖
2. 新建一个类实现javax.servlet.Servlet接口
3. 实现对应的方法

Servlet接口之中的方法

1. 无参构造方法【尽量别动】
2. init方法

- 和构造方法执行时间基本一致，并且都只执行一次。
- 构造函数执行时，对象正在创建，init方法执行时，对象已创建
- 对于需要在实例化Servlet对象的时候，执行一段代码。尽量放在init里面。写到构造函数里面存在风险，当编写了有参构造函数后，系统不再提供无参构造函数，这时服务器调用无参构造函数新建实例时就会调用失败

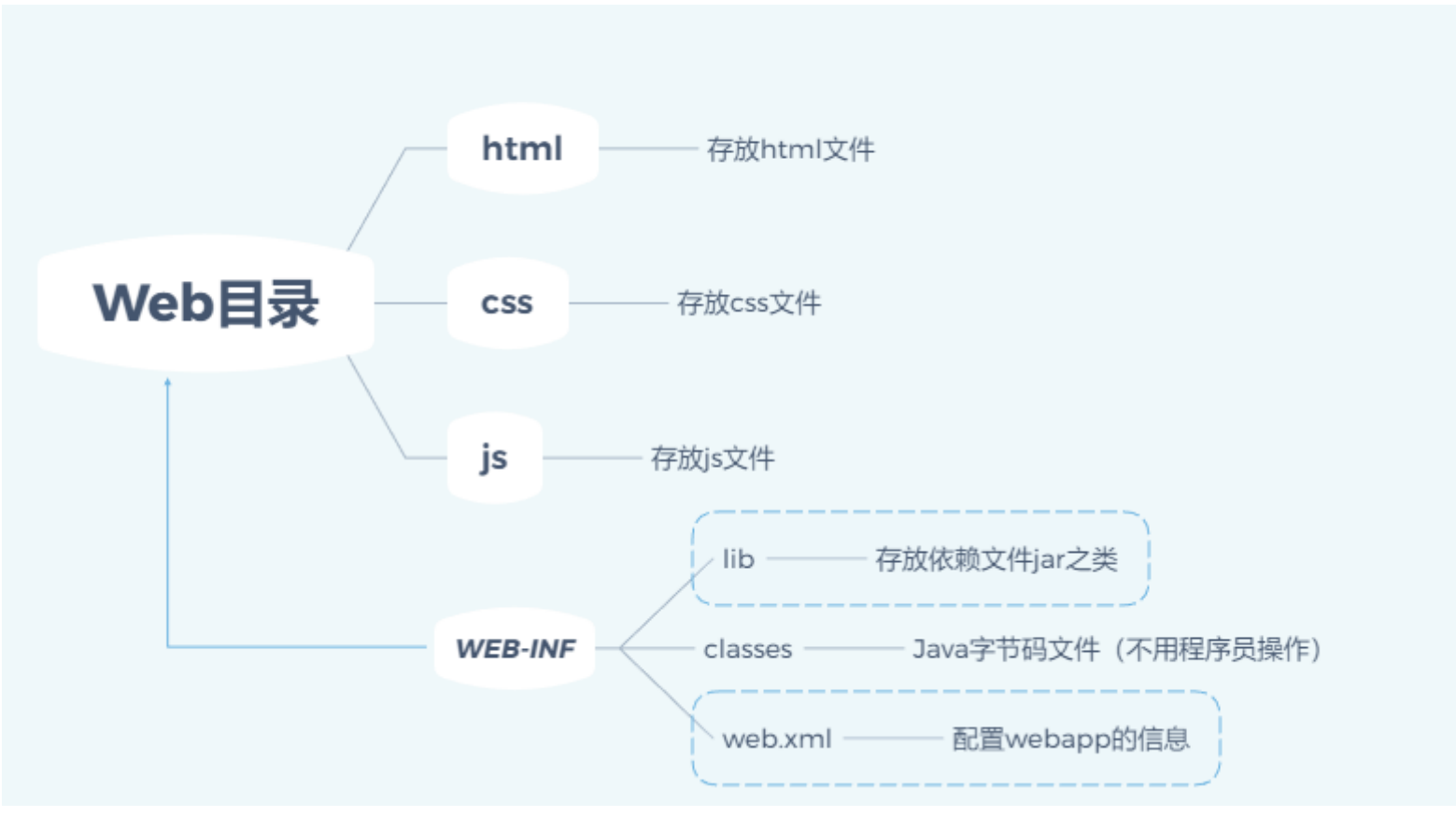
3. service方法

- 必然要重写，在这里完成业务逻辑的处理，请求和响应的处理。是最有价值的。

4. destroy方法

- 在Servlet实例对象即将被销毁的时候被调用， 希望在Servlet对象销毁时执行一段代码，把这段代码放到destroy方法里面。

web目录说明



Servlet 的生命周期

1、什么是生命周期？

生命周期表示一个java对象从最初被创建到最终被销毁，经历的所有过程。

2、Servlet对象的生命周期是谁来管理的？程序员可以干涉吗？

Servlet对象的生命周期，javaweb程序员是无权干涉的，包括该Servlet对象的相关方法的调用，javaweb程序员也是无权干涉的。

Servlet对象从最初的创建，方法的调用，以及最对象的销毁，整个过程，是由WEB容器来管理的。

Web Container管理Servlet对象的生命周期。

3、“默认情况下”，Servlet对象在WEB服务器启动阶段不会被实例化。【若希望在web服务器启动阶段实例化Servlet对象，需要进行特殊的设置】

4、描述Servlet对象生命周期

1）用户在浏览器地址栏上输入URL：http://localhost:8080/prj-servlet-03/testLifeCycle

2）web容器截取请求路径：/prj-servlet-03/testLifeCycle

3）web容器在容器上下文中找请求路径 /prj-servlet-03/testLifeCycle对应的Servlet对象

4）若没有找到对应的Servlet对象

4.1）通过web.xml文件中相关的配置信息，得到请求路径/testLifeCycle对应的Servlet完整类名

4.2）通过反射机制，调用Servlet类的无参数构造方法完成Servlet对象的实例化

4.3）web容器调用Servlet对象的init方法完成初始化操作

4.4）web容器调用Servlet对象的service方法提供服务

5）若找到对应的Servlet对象

5.1）web容器直接调用Servlet对象的service方法提供服务

6）web容器关闭的时候/webapp重新部署的时候/该Servlet对象长时间没有用户再次访问的时候，web容器会将该Servlet对象销毁，在销毁该对象之前，web容器会调用Servlet对象的destroy方法，完成销毁之前的准备。

7、Servlet对象是单例，但是不符合单例模式，只能称为伪单例。真单例的构造方法是私有化的，Tomcat服务器是支持多线程的。所以Servlet对象在单实例多线程的环境下运行的。那么Servlet对象中若有实例变量，并且实例变量涉及到修改操作，那么这个Servlet对象一定会存在线程安全问题，不建议在Servlet对象中使用实例变量，尽量使用局部变量。

8、若希望在web服务器启动阶段实例化Servlet对象，需要在web.xml文件中进行相关的配置，例如：

```
<servlet>
    <servlet-name>testLifeCycle</servlet-name>
    <servlet-class>com.bjpowernode.javaweb.servlet.HelloServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>testLifeCycle</servlet-name>
    <url-pattern>/testLifeCycle</url-pattern>
</servlet-mapping>

<servlet>
    <servlet-name>welcomeServlet</servlet-name>
    <servlet-class>com.bjpowernode.javaweb.servlet.WelcomeServlet</servlet-class>
    <load-on-startup>0</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>welcomeServlet</servlet-name>
    <url-pattern>/welcome</url-pattern>
</servlet-mapping>
```

注意：<load-on-startup>1</load-on-startup> 自然数越小优先级越高

9、Servlet对象实例化之后，这个Servlet对象被存储到哪里了？

大多数的WEB容器都是将该Servlet对象以及对应的url-pattern存储到Map集合中了：

在WEB容器中有这样一个Map集合

Map<String,Servlet> 集合

key	value
/login	LoginServlet对象引用
/delete	DeleteServlet对象引用
/save	SaveServlet对象引用

10、服务器在启动的时候就会解析各个webapp的web.xml文件，做了什么？

将web.xml文件中的url-pattern和对应的Servlet完整类名存储到Map集合中了：

在WEB容器中有这样一个Map集合

Map<String,String> 集合

key	value
/login	com.bjpowernode.javaweb.servlet.LoginServlet
/delete	com.bjpowernode.javaweb.servlet.DeleteServlet
.....	

11、Servlet接口中的这些方法中编写什么代码？什么时候使用这些方法？

1）无参数构造方法【以后就不需要再考虑构造函数了，尽量别动构造函数】

2）init方法

以上两个方法执行时间几乎是相同的，执行次数都是一次，构造方法执行的时候对象正在创建，init方法执行的时候对象已经创建：

若系统要求在对象创建时刻执行一段特殊的程序，这段程序尽量写到init方法中。

为什么不建议将代码编写到构造函数中呢？

存在风险！

当程序员编写构造方法的时候，可能会导致无参数构造方法不存在。

一个类不编写任何构造函数，默认有一个无参数的构造方法，但是一旦编写一个有参数的构造方法之后，系统则不再提供无参数构造函数。

Servlet中的init方法是SUN公司为javaweb程序员专门提供的一个初始化时刻，若希望在初始化时刻执行一段特殊的程序，这个程序可以编写到init方法，将来会被自动调用。

3）service方法

这个方法是必然要重写的，因为在这个方法需要完成业务逻辑的处理，请求的处理，以及完成响应。

而且这个方法中的代码是最有价值的。

也是最难写的，因为最难编写的就是业务代码啦。

4）destroy方法

这个方法也是SUN公司为javaweb程序员提供的一个特殊的时刻，这个特殊的时刻被称为对象销毁时刻，

若希望在销毁时刻执行一段特殊的代码，需要将这段代码编写到destroy方法，自动被容器调用。

回顾：

类加载时刻执行程序，代码写到哪里？

编写到静态代码块中。

结论：

sun公司为我们程序员提供了很多个不同的时刻。若在这个特殊时刻执行特殊程序，这些程序是有位置编写的。

将响应结果输出到浏览器

```
// 将响应内容类型设置为html，并且字符集为UTF-8
response.setContextType("text/html;charset=UTF-8");
// 获取PrintWriter对象
PrintWriter out = response.getWriter();
// 将内容输出到浏览器
our.println("<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Test Page</title>
</head>
<body>
    <p align="center">
        welcome to web!
    </p>
</body>
</html>");
// 这样就能够在浏览器看到输出的内容。后面动态加载界面原理类似
```

ServletConfig Servlet 配置对象

文档说明：servlet 容器使用的 servlet 配置对象，该对象在初始化期间将信息传递给 servlet。

一个Servlet对象对应一个ServletConfig对象，一百个Servlet对象就有一百个ServletConfig对象

xml配置文件：

```
<?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">
    <servlet>
        <servlet-name>one</servlet-name>
        <servlet-class>com.fightersu.controller.OneServlet</servlet-class>
        <--! 写在Servlet标签里面的参数通过ServletConfig对象来获取 -->
        <init-param>
            <param-name>driver</param-name>
            <param-value>com.mysql.jdbc.Driver</param-value>
        </init-param>
        <init-param>
            <param-name>url</param-name>
            <param-value>jdbc:mysql://localhost:3306/su</param-value>
        </init-param>
        <init-param>
            <param-name>user</param-name>
            <param-value>root</param-value>
        </init-param>
        <init-param>
            <param-name>password</param-name>
            <param-value>4869</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>one</servlet-name>
        <url-pattern>/one</url-pattern>
    </servlet-mapping>
    <servlet>
        <servlet-name>two</servlet-name>
        <servlet-class>com.fightersu.controller.TwoServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>two</servlet-name>
        <url-pattern>/two</url-pattern>
    </servlet-mapping>
</web-app>
```

Servlet类编写：

```
response.setContentType("text/html;charset=UTF-8");
PrintWriter out = response.getWriter();
// 通过初始化参数的name获取value
String driver = config.getInitParameter("driver");
String url = config.getInitParameter("url");
String user = config.getInitParameter("user");
String password = config.getInitParameter("password");

// 获取全部参数名字的集合
Enumeration<String> names = config.getInitParameterNames();
while(names.hasMoreElements()){
    String name = names.nextElement();
    String value = config.getInitParameter(name);
    out.println(name + " = " + value);
    out.println("<br>");
}
```

```java
// 获取servletName ： <servlet-name>servletName</servlet-name>
String servletName = config.getServletName();
out.print("<servlet-name>" + servletName + "</servlet-name>");

// 获取 Servlet上下文对象
ServletContext application = config.getServletContext();
System.out.println(application);
```

ServletContext Servlet上下文对象

1. 一个Servlet对象对应一个ServletConfig对象
2. 多个Servlet对象对应一个ServletContext对象
3. 一个webapp只有一个ServletContext对象
4. 一个webapp只有一个web.xml文件
5. ServletContext对应的是web.xml文件

- web.xml文件在服务器启动阶段被解析，所以ServletContext对象也在这时被实例化，在服务器关闭时进行销毁
- 因为它是对应这多个Servlet的，如果需要在同一个webapp中的多个Servlet之间通信，可以将信息存放在ServletContext对象之中
- 同时，因为是共享变量，进行修改时会存在线程安全问题

```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">
    <context-param>
        <param-name>user</param-name>
        <param-value>fighter</param-value>
    </context-param>
    <context-param>
        <param-name>password</param-name>
        <param-value>4869</param-value>
    </context-param>
    <servlet>
        <servlet-name>one</servlet-name>
        <servlet-class>com.fightersu.controller.OneServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>one</servlet-name>
        <url-pattern>/one</url-pattern>
    </servlet-mapping>
    <servlet>
        <servlet-name>two</servlet-name>
        <servlet-class>com.fightersu.controller.TwoServlet</servlet-class>
    </servlet>
```

```
    <servlet-mapping>
        <servlet-name>two</servlet-name>
        <url-pattern>/two</url-pattern>
    </servlet-mapping>
</web-app>
```

ServletContext常用函数：

```
// 在init()函数里面获取ServletConfig对象
// private ServletConfig config;
// @Override
// public void init(ServletConfig servletConfig) throws ServletException {
//     config = servletConfig;
// }
// 1. 利用ServletConfig对象获取ServletContext对象
ServletContext context = config.getServletContext();
// 2. 利用ServletConText对象获取在web.xml文件里面<context-param>标签的值
String user = context.getInitParameter("user");
String password = context.getInitParameter("password");
System.out.println("user = " + user + ";password = " + password);
// 3. 获取预设全部参数的名字及值
Enumeration<String> names = context.getInitParameterNames();
while(names.hasMoreElements()){
    String name = names.nextElement();
    String value = context.getInitParameter(name);
    System.out.println(name + " = " + value);
}
// 4. 向ServletContext对象之中添加值
context.setAttribute("user", "fighterSu");
context.setAttribute("date", new Date());

// 5. 在别的Servlet里面获取存在ServletContext对象之中的值，以及删除值
// ServletContext context = config.getServletContext();
// response.setContentType("text/html;charset=UTF-8");
// PrintWriter out = response.getWriter();
// out.println(context.getAttribute("user"));
// out.println("<br>");
// out.println(context.getAttribute("date"));
// context.removeAttribute("date");

// 6. 获取从web目录开始的文件的绝对路径
String realPath = context.getRealPath("/resources/db.properties");
System.out.println("realPath = " + realPath);
```

欢迎界面

- 优点：访问更加方便，用户体验好

```
// 这是Tomcat的conf文件夹下的web.xml文件设置的，即默认欢迎页面
<welcome-file-list>
        <welcome-file>index.html</welcome-file>
```

```
        <welcome-file>index.htm</welcome-file>
        <welcome-file>index.jsp</welcome-file>
</welcome-file-list>

<?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">
    <!-- 不配置时使用全局设置，欢迎界面依次为index.html,index.htm,index.jsp -->
    <!-- 局部设置优先于全局设置 -->
    <!-- 设置优点：优化用户体验；简化访问 -->
    <welcome-file-list>
        <!-- 欢迎界面不一定是html等文件，也可以是Servlet -->
        <!-- <welcome-file>one</welcome-file>      -->
        <welcome-file>html/welcome.html</welcome-file>
        <!-- 可以设置多个文件，在上面的资源优先，在找不到上面的资源时，依次会启用下面的资源 -->
        <welcome-file>index.html</welcome-file>
    </welcome-file-list>
</web-app>
```

常见错误状态码

- 404 – Not Found 【资源未找到：请求的资源路径写错了】
- 500 – Server Inner Error 【服务器内部错误：一般是Java程序出现异常】

以上HTTP协议状态码是W3C制定的，所有浏览器和服务器都必须遵守

正常响应：200【OK】

404 – Not Found

HTTP状态 500 – 内部服务器错误

```
@Override
public void service(ServletRequest req, ServletResponse res) throws
ServletException, IOException {
        throw new IOException();
}
```

HTTP状态 500 - 内部服务器错误

类型 异常报告

描述 服务器遇到一个意外的情况，阻止它完成请求。

例外情况

```
java.io.IOException
        com.fightersu.controller.TestServlet.service(TestServlet.java:23)
        org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:53)
```

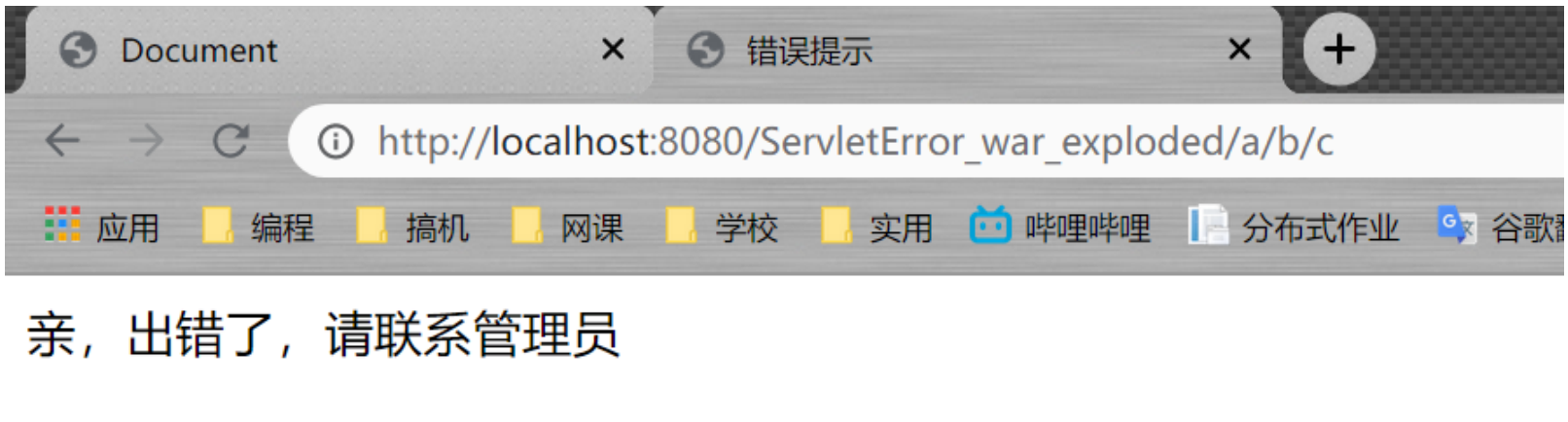
):注意 主要问题的全部 stack 信息可以在 server logs 里查看

Apache Tomcat/9.0.53

可以指定发生对应错误时，显示指定画面，对客户更加友好，他们不需要知道那些复杂的东西

在web.xml之中进行配置：

```
    <error-page>
        <error-code>404</error-code>
        <location>/error/error.html</location>
    </error-page>
    <error-page>
        <error-code>500</error-code>
        <location>/error/error.html</location>
    </error-page>
```



请求

组成

- 请求行：请求方式 URI 协议版本号
- 消息报头:
- 空白行：为了分隔消息报头和请求体
- 请求体
- 例子

```
POST /HttpServlet_war_exploded/system/one HTTP/1.1                    请求行
Accept: text/html, application/xhtml+xml, image/jxr, */*              消息报头
X-HttpWatch-RID: 87007-10033
Referer: http://localhost:8080/HttpServlet_war_exploded/
Accept-Language: zh-CN
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
Host: localhost:8080
Content-Length: 29
Connection: Keep-Alive
Cache-Control: no-cache
                                                                      空白行
username=admin&password=*****                                         请求体
```

响应

组成

- 状态行： 协议版本号 状态码 状态描述信息
- 响应报头
- 空白行：为了分隔响应报头和请求体
- 响应体
- 例子

```
HTTP/1.1 200                                    状态行，200表示正常
Content-Type: text/html;charset=utf-8           响应报头
Content-Length: 14
Date: Wed, 29 Sep 2021 09:25:10 GMT
Keep-Alive: timeout=20
Connection: keep-alive
                                                空白行
Hello,World!                                    响应体
```

GET和POST请求的区别

	GET	POST
发送数据位置	请求行	请求体
提交数据长度	在请求行之中，有长度限制	在请求体之中，无长度限制
请求结果是否会被浏览器缓存	最终结果会被浏览器缓存	不会被浏览器缓存
提交是否会被显示在地址栏上	会	不会，会对密码进行加密
传送数据	字符串	所有类型

对密码的处理

GET请求会直接将密码不做处理，直接放到请求行之中发送，POST则是进行加密，再放入请求体之中发送

```
GET /HttpServlet_war_exploded/system/one?username=admin&password=admin HTTP/1.1

POST /HttpServlet_war_exploded/system/one HTTP/1.1

username=admin&password=*****
```

结果被缓存

- GET多数是从服务器读取资源，短时间内不会发生变化，所以其请求结果被浏览器缓存了起来，方便下次加载
- POST是为了修改服务器的资源，而每一次请求结果基本不同，没有保存的需要

缓存的资源是和某个特定路径绑定在一起的，只有发送请求路径一致，才会去读取缓存内容

所以想GET请求结果不被浏览器缓存，可以选择添加时间戳的方式实现，这样每次请求连接不同，那么就不会被调取缓存

如何选择GET和POST

- 有 敏感数据 必须用POST
- 传送数据 不是字符串 ， 必须用POST
- 传送数据多，POST
- 请求是为了修改服务器资源，POST
- 其它都用GET

适配器

1. 目前直接实现Servlet接口，但很多方法不需要，只需要service()方法，
   直接实现Servlet接口代码丑陋，有必要添加一个适配器，以后所有的
   Servlet类不再实现Servlet接口，应该直接去继承适配器类。
2. 适配器除了让代码优雅之外，我们应该在适配器之中提供大量方法，子类继承之后，方便使用，方便编
   程

前端页面发送请求方式应该和服务器端需要的请求方式一致

- 服务器需要前端发送POST请求，若发送GET请求，服务器应当提示错误信息
- 服务器需要前端发送GET请求，若发送POST请求，服务器应当提示错误信息
  怎么实现
  -- 在JavaWeb服务器之中获取该请求是什么类型的，POST？还是GET?
  -- 获取到后进行判断，进行相应处理
  HTTP的请求信息被自动封装到javax.servlet.http.HttpServletRequest对象之中
  -- 在该类之中，有一个方法 String getMethod() 方法，可以来获取请求的方式
- 下面是手动实现判断：

```
package com.fightersu.controller;

import javax.servlet.GenericServlet;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;

/**
 * @author Platina
 * @date 2021/9/25 11:27
 **/
public class HttpServletTest extends GenericServlet {
    @Override
    public void service(ServletRequest req, ServletResponse resp) throws
ServletException, IOException {
        // 将ServletRequest,  ServletResponse强制类型转换成带Http的接口类型
        HttpServletRequest request = (HttpServletRequest) req;
        HttpServletResponse response = (HttpServletResponse) resp;

        response.setContentType("text/html;charset=utf-8");
        PrintWriter writer = response.getWriter();

        // 获取请求方式
        String method = request.getMethod();
        System.out.println(method);
        String targetMethod = "POST";
        // 请求不符合要求
        if(!targetMethod.equals(method)){
            // 前台显示错误界面
            writer.println("405-您应当发送POST请求");
            // 后台显示错误信息
            System.out.println("405-您应当发送POST请求");
            throw new RuntimeException("405-您应当发送POST请求");
```

```
        }

        writer.println("正在登录.....");
    }
}
```

HttpServlet

对于以上的需求：前端页面发送请求方式应该和服务器端需要的请求方式一致，SUN公司提供了实现，即javax.servlet.http.HttpServlet 类

部分源码：

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
    String msg = lStrings.getString("http.method_get_not_supported");
    this.sendMethodNotAllowed(req, resp, msg);
}
protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
    String msg = lStrings.getString("http.method_post_not_supported");
    this.sendMethodNotAllowed(req, resp, msg);
}

// 发送错误信息，提示该请求不被允许
private void sendMethodNotAllowed(HttpServletRequest req, HttpServletResponse resp,
String msg) throws IOException {
    String protocol = req.getProtocol();
    if (protocol.length() != 0 && !protocol.endsWith("0.9") &&
!protocol.endsWith("1.0")) {
        resp.sendError(405, msg);
    } else {
        resp.sendError(400, msg);
    }
}

// 主要方法
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);
    }

}
```

以上代码我们可以知道，当子类继承了HttpServlet方法，没有重写对应doXXX()方法时，必定抛出异常

HTTP状态 405 - 方法不允许

类型 状态报告

消息 此URL不支持Http方法POST

描述 请求行中接收的方法由源服务器知道，但目标资源不支持

Apache Tomcat/9.0.53

目标是哪个类型的就重写那个类型的doXXX()方法，当发送其它请求时就会抛出405异常

- Servlet类应当继承HttpServlet类，get请求重写doGet()，post请求重写doPost()
- doGet(),doPost()可以看作main函数，在里面做各种操作

模板方法设计模式

行为型设计模式

典型代表：HttpServlet

特点：一般方法都是doXXX()，doYYY()，doZZZ()等

当一系列类，核心算法骨架相同，那么一般就需要使用模板方法设计模式，不然必定会出现代码冗余

并且写一系列相似的方法，算法不能够受到保护，算法可能被修改造成算法错误

- templateMethod是一个模板方法，定义了核心的骨架，具体实现步骤延迟到子类完成
- 需要使用finnal修饰，保护核心骨架
- 模板类一般是抽象类，具体实现方法是抽象方法
- 在不改变算法的前提下，改变算法具体执行步骤

HttpServletRequest

接口实现类是由WEB容器实现的，程序员只需要面向接口编程即可

封装信息

- HTTP协议的全部内容
  - 请求方式
  - URI
  - 协议版本号
  - 表单提交的数据

```
// 使用最多的方法 request.getParameter(String) 获取数据数组的第一个元素
String username = request.getParameter("username");
String password = request.getParameter("password");
String sex = request.getParameter("sex");
String grade = request.getParameter("grade");
// 常用的方法 request.getParameterValues(String) 获取整个数据数组
String[] interests = request.getParameterValues("interests");
String simpleInfo = request.getParameter("simpleInfo");
// 获取上下文路径【webapp的根路径】
String contextPath = request.getContextPath();
```

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>Index Page</title>
    </head>
    <body>
        <form action="/HttpServletRequest_war_exploded/one" method="post">
            用户名: <label>
            <input type="text" name="username">
        </label><br>
```

```
          密码: <label>
        <input type="password" name="password">
    </label><br>
        性别: <label>
        <input type="radio" name="sex" value="man">男  
        <input type="radio" name="sex" value="woman">女
    </label><br>
        学历: <label>
        <select name="grade">
            <option value="高中">高中</option>
            <option value="本科">本科</option>
            <option value="硕士">硕士</option>
            <option value="博士">博士</option>
        </select>
    </label><br>
        兴趣爱好: <label>
        <input type="checkbox"  value="run">跑步
        <input type="checkbox" name="interests" value="ball">打球
        <input type="checkbox" name="interests" value="sing">唱歌
        <input type="checkbox" name="interests" value="dance">跳舞
    </label><br>
        简介: <label>
        <input type="text" name="simpleInfo">
    </label><br>
        <input type="submit" name="提交"> 
        <input type="reset" name="重置">
    </form>
  </body>
</html>
```

项目之中的中文乱码问题

乱码位置

1. 数据“传递”过程之中的乱码
2. 数据“展示”过程之中的乱码
3. 数据“保存”过程之中的乱码

数据“保存”之中的乱码

最终保存数据到数据库表中的时候，数据出现乱码

1. 在保存之前就乱码，保存到数据库一定乱码
2. 保存之前不乱码，由于数据库不支持简体中文，保存之后出现乱码

数据“展示”之中的乱码

最终显示在网页上的数据出现中文乱码

经过执行Java程序之后，Java程序负责向浏览器响应的时候，中文出现乱码

1. 设置响应的内容类型，以及对应的字符编码方式：
    1. response.setContext("text/html;charset=UTF-8");

没有经过Java程序，直接访问静态页面html文件等

2. 指定html文件等的编码
    1. < mata context="text/html;charset=UTF-8">
    2. < mata charset="UTF-8">

数据“传递”之中的乱码

将数据从浏览器发给服务器的时候，服务器收到的是乱码

使用ISO-8859-1的编码，国际标准码，不支持中文，称为latin1编码

浏览器将数据发送给服务器，统一使用ISO-8859-1编码，web服务器不知道之前是什么文字，所以出现乱码

解决方案：将请求体数据按照ISO-8859-1进行编码，得到原始数据，再使用UTF-8格式新建String来解码即可

```
String dname = request.getParameter("dname");
        System.out.println("原编码直接输出：部门名称：" + dname);
        dname = new
String(request.getParameter("dname").getBytes(StandardCharsets.ISO_8859_1),
StandardCharsets.UTF_8);
        System.out.println("UTF-8编码输出：部门名称：" + dname);
// get请求
// 原编码直接输出：部门名称：人事部
// UTF-8编码输出：部门名称：???
// post请求
// 原编码直接输出：部门名称：??????é?¨
// UTF-8编码输出：部门名称：人事部
```

Tomcat URI默认使用UTF-8编码，所以get请求数据直接获取即可

post请求数据在请求体之中，使用了IOS-8859-1进行了编码，所以需要处理才能获取正确数据

解决方式：

```
// 1. 将请求体数据按照ISO-8859-1进行编码，得到原始数据，再使用UTF-8格式新建String来解码即可
String dname = new
String(request.getParameter("dname").getBytes(StandardCharsets.ISO_8859_1),
StandardCharsets.UTF_8);
        System.out.println("UTF-8编码输出：部门名称：" + dname);

// 2. 在读取数据之前，设置HttpServletRequest请求对象的字符编码
request.setCharacterEncoding("UTF-8");
String dname = request.getParameter("dname");
System.out.println("原编码直接输出：部门名称：" + dname);
```

web系统之中资源跳转

跳转方式

1. 转发：forward
2. 重定向：redirect

转发和重定向代码

```
// 转发
request.getRequestDispatcher("/two").forward(request, response);
// 重定向
response.sendRedirect(request.getContextPath() + "/two");
```

转发和重定向的异同

同：都可以完成资源跳转

异：

项目	转发	重定向
触发对象	request对象	response对象
请求次数	1次，浏览器地址栏不会变化【/one】	2次，浏览器地址栏会发生变化【/one，/two】
跳转路径	/one	webapp根路径 + /two
跳转范围	只能是本项目内部	可以完成跨app跳转

这里的webapp根路径一般使用request.getContextPath()来获取

跳转的下一个资源是什么

可以是web服务器之中任何一种资源：Servlet，HTML，JSP等等

转发和重定向的选择【大部分使用重定向】

若需要跨app跳转，必须重定向

若在上一个资源中向request范围之中存储了数据，想要在下一个资源之中取出来，必须使用转发

重定向可以解决浏览器刷新问题

原理：重定向后，地址栏是第二次请求地址，刷新不会出现问题，而转发地址栏还是第一次地址，刷新后会再次发送一次请求，进行数据插入的时候就可能重复插入数据

重定向原理

response.sendRedirect("/jd/login")

1. 程序执行到以上代码，将请求路径/jd/login反馈给浏览器
2. 浏览器自动又向web服务器发送了一次全新的请求：/jd/login
3. 浏览器地址栏最终路径是：/jd/login

借钱例子：

我找张三借钱，他现在没钱

转发：他找李四借了钱然后把钱借给我，我只是找了张三一个人借了钱

重定向：他说自己没钱，他告诉我李四有钱，李四在教四；然后我又去找李四借钱，随后我总共找了两个人借钱

Cookie

是什么，有什么作用

- 可以保存会话状态，保存在客户端上
- 只要cookie清除，或者失效，那么这个会话状态就没有了
- 保存在浏览器的缓存之中的cookie关闭浏览器后即失效
- 保存在硬盘文件上，关闭浏览器后还有
- 十天内免登录，未登录保存商品至购物车等功能实现
- 不仅仅在JavaWeb之中有，只要是web开发都有，基于HTTP协议就有cookie，由http规定

使用

在Java之中，Cookie被当作类处理，使用new创建Cookie类对象，然后添加到浏览器之中

- 可以一次向浏览器发送多个Cookie

```
Cookie cookie = new Cookie(String cookieName, String cookieValue);
Cookie cookie1 = new Cookie("username", "fighterSu");
Cookie cookie2 = new Cookie("password", "4869");
response.addCookie(cookie1);
response.addCookie(cookie2);
```

截图：

```
HTTP/1.1 200
Set-Cookie: username=fighterSu
Set-Cookie: password=4869
Content-Type: text/html;charset=UTF-8
Content-Length: 21
Date: Sat, 02 Oct 2021 08:18:22 GMT
Keep-Alive: timeout=20
Connection: keep-alive

æ·»å cookieæåï¼
```

什么时候会将Cookie发送给服务器

- 浏览器会不会提交发送这些Cookie给服务器，和请求路径有关系。
- 请求路径和Cookie是紧密关联的。
- 不同的请求路径会发送提交不同的Cookie

Cookie的路径绑定

- 默认情况
  - /prj-servlet-18/test/createAndSendCookieToBrowser 请求服务器，服务器生成Cookie，并将Cookie发送给浏览器客户端

    这个浏览器中的Cookie会默认和“test/”这个路径绑定在一起。

    也就是说，以后只要发送“test/”请求，Cookie一定会提交给服务器。

```
GET /Cookie_war_exploded/ HTTP/1.1
Accept: text/html, application/xhtml+xml, image/jxr, */*
X-HttpWatch-RID: 79341-10018
Accept-Language: zh-Hans-CN,zh-Hans;q=0.5
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko
Accept-Encoding: gzip, deflate
Host: localhost:8080
If-Modified-Since: Sat, 02 Oct 2021 07:54:55 GMT
If-None-Match: W/"330-1633161295011"
Connection: Keep-Alive
Cookie: username=fighterSu; password=4869; Idea-aa5f2f9f=cec92e0f-d47e-42bd-bdf4-0e966b1b2434
```

  - /prj-servlet-18/a 请求服务器，服务器生成Cookie，并将Cookie发送给浏览器客户端

    这个浏览器中的Cookie会默认和“prj-servlet-18/”这个路径绑定在一起。

    也就是说，以后只要发送“prj-servlet-18/”请求，Cookie一定会提交给服务器。

- 设置绑定路径
  - 其实路径是可以指定的，可以通过java程序进行设置，保证Cookie和某个特定的路径绑定在一起。

    假设，执行了这样的程序：cookie.setPath("/prj-servlet-18/king");

    那么：Cookie将和"/prj-servlet-18/king"路径绑定在一起

    只有发送“/prj-servlet-18/king”请求路径，浏览器才会提交Cookie给服务器。

- 当发送其它路径时，浏览器将不会提交cookie

```
HTTP/1.1 200
Set-Cookie: username=fighterSu; Path=/Cookie_war_exploded/fighter
Set-Cookie: password=4869; Path=/Cookie_war_exploded/su
Content-Type: text/html;charset=UTF-8
Content-Length: 21
Date: Sat, 02 Oct 2021 09:39:11 GMT
Keep-Alive: timeout=20
Connection: keep-alive

æ·»å cookieæåï¼
```

- 可以看到，设置了路径之后，cookie后面多了Path限制，同时只有访问指定路径，浏览器才会发送cookie

将Cookie保存到本地文件之中

默认情况下，没有设置Cookie的有效时长，该Cookie被默认保存在浏览器的缓存当中，只要浏览器不关闭Cookie存在，只要关闭浏览器Cookie消失，我们可以通过设置Cookie的有效时长，以保证Cookie保存在硬盘文件当中。但是这个有效时长必须是>0的。换句话说，只要设置Cookie的有效时长大于0，则该Cookie会被保存在客户端硬盘文件当中。有效时长过去之后，则硬盘文件当中的Cookie失效。

cookie有效时长 = 0 直接被删除

cookie有效时长 < 0 不会被存储

cookie有效时长 > 0 存储在硬盘文件当中

cookie.setMaxAge(60 * 60); 1小时有效，参数是秒为单位的

服务器如何接收Cookie

```
Cookie[] cookies = request.getCookies();
if(cookies != null){
    for(Cookie cookie : cookies){
        String cookieName = cookie.getName();
        String cookieValue = cookie.getValue();
        out.println(cookieName + " = " + cookieValue);
    }
}
```

禁用Cookie

- 表示服务器发送过来的Cookie，我浏览器不要，不接收。
- 服务器还是会发送Cookie的，只不过浏览器不再接收。
- 大多数网站依托Cookie完成一系列设定，不推荐禁用Cookie

关于url-pattern的编写方式和路径的总结

路径的编写形式

-
- 项目名通常使用request.getContextPath()获取
-
- 重定向：response.sendRedirect("/项目名/资源路径");
- 转发：request.getRequestDispatcher("/资源路径").forward(request,response);
- 欢迎页面

  <welcome-file-list>

  <welcome-file>资源路径</welcome-file>

  </welcome-file-list>
- servlet路径

  <servlet>

  <servlet-name>hello</servlet-name>

  <servlet-class>com.bjpowernode.javaweb.servlet.HelloServlet</servlet-class>

  </servlet>

  <servlet-mapping>

  <servlet-name>hello</servlet-name>

  <url-pattern>/资源路径</url-pattern>

  </servlet-mapping>
- Cookie设置path

  cookie.setPath("/项目名/资源路径");
- ServletContext

  ServletContext application = config.getServletContext();

  application.getRealPath("/WEB-INF/classes/db.properties");

  application.getRealPath("/资源路径");

url-pattern的编写方式

url-pattern可以编写多个

精确匹配

<url-pattern>/hello</url-pattern>

<url-pattern>/system/hello</url-pattern>

扩展匹配

<url-pattern>/abc/*</url-pattern>

后缀匹配

<url-pattern>*.action</url-pattern>

<url-pattern>*.do</url-pattern>

全部匹配

<url-pattern>/*</url-pattern>

web编程中的Session

1、Session表示会话，不止是在javaweb中存在，只要是web开发，都有会话这种机制。

2、在java中会话对应的类型是：javax.servlet.http.HttpSession，简称session/会话

3、Cookie可以将会话状态保存在客户端，HttpSession可以将会话状态保存在服务器端。

4、HttpSession对象是一个会话级别的对象，一次会话对应一个HttpSession对象。

5、什么是一次会话?

“目前”可以这样理解：用户打开浏览器，在浏览器上发送多次请求，直到最终关闭浏览器，表示一次完整的会话。

6、在会话进行过程中，web服务器一直为当前这个用户维护着一个会话对象/HttpSession.

7、在WEB容器中，WEB容器维护了大量的HttpSession对象，换句话说，在WEB容器中应该有一个“session列表”，

思考：为什么当前会话中的每一次请求可以获取到属于自己的会话对象？session的实现原理?

- 打开浏览器，在浏览器上发送首次请求
- 服务器会创建一个HttpSession对象，该对象代表一次会话
- 同时生成HttpSession对象对应的Cookie对象，并且Cookie对象的name是JSESSIONID，Cookie的value是32位长度的字符串
- 服务器将Cookie的value和HttpSession对象绑定到session列表中
- 服务器将Cookie完整发送给浏览器客户端
- 浏览器客户端将Cookie保存到缓存中
- 只要浏览器不关闭，Cookie不会消失
- 当再次发送请求的时候，会自动提交缓存当中的Cookie
- 服务器接收到Cookie，验证该Cookie的name确实是：JSESSIONID，然后获取该Cookie的value
- 通过Cookie的value去session列表中检索对应的HttpSession对象。

```
HTTP/1.1 200
Set-Cookie: JSESSIONID=7ABE5A5C8A28A00A095BC22E98A13FB1; Path=/Session; HttpOnly
Content-Length: 0
Date: Sun, 03 Oct 2021 07:40:36 GMT
Keep-Alive: timeout=20
Connection: keep-alive
```

```
GET /Session/ HTTP/1.1
Accept: text/html, application/xhtml+xml, image/jxr, */*
X-HttpWatch-RID: 91874-10012
Accept-Language: zh-Hans-CN,zh-Hans;q=0.5
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko
Accept-Encoding: gzip, deflate
Host: localhost:8080
Connection: Keep-Alive
If-Modified-Since: Sun, 03 Oct 2021 07:29:53 GMT
If-None-Match: W/"308-1633246193708"
Cookie: JSESSIONID=7ABE5A5C8A28A00A095BC22E98A13FB1
```

8、和HttpSession对象关联的这个Cookie的name是比较特殊的，在java中就叫做：jsessionid

9、浏览器禁用Cookie会出现什么问题？怎么解决?

- 浏览器禁用Cookie，则浏览器缓存中不再保存Cookie
- 导致在同一个会话中，无法获取到对应的会话对象
- 禁用Cookie之后，每一次获取的会话对象都是新的

浏览器禁用Cookie之后，若还想拿到对应的Session对象，必须使用URL重写机制,怎么重写URL:

http://localhost/prj-servlet-21/user/accessMySelfSession;jsessionid=D3E9985BC5FD4BD05018BF2966863E94

重写URL会给编程带来难度/复杂度，所以一般的web站点是不建议禁用Cookie的。建议浏览器开启Cookie

10、浏览器关闭之后，服务器端对应的session对象会被销毁吗?为什么?

- 浏览器关闭之后，服务器不会销毁session对象
- 因为B/S架构的系统基于HTTP协议，而HTTP协议是一种无连接/无状态的协议
- 什么是无连接/无状态?
  - 请求的瞬间浏览器和服务器之间的通道是打开的，请求响应结束之后，通道关闭
  - 这样做的目的是降低服务器的压力。

11、session对象在什么时候被销毁?

- web系统中引入了session超时的概念。
- 当很长一段时间（这个时间可以配置）没有用户再访问该session对象，此时session对象超时，web服务器自动回收session对象。
- 可配置:web.xml文件中，默认是30分钟

```
<session-config>
    <session-timeout>120</session-timeout>
</session-config>
```

12、什么是一次会话呢?

- 一般多数情况下，是这样描述的：用户打开浏览器，在浏览器上进行一些操作，然后将浏览器关闭，表示一次会话结束。
- 本质上的描述：从session对象的创建，到最终session对象超时之后销毁，这个才是真正意义的一次完整会话。

13、关于javax.servlet.http.HttpSession接口中常用方法：

-void setAttribute(String name, Object value)

-Object getAttribute(String name)

-void removeAttribute(String name)

-void invalidate() 销毁session

14、ServletContext、HttpSession、HttpServletRequest接口的对比：

14.1 以上都是范围对象:

ServletContext application; 是应用范围

HttpSession session; 是会话范围

HttpServletRequest request; 是请求范围

14.2 三个范围的排序：

application > session > request

14.3

application完成跨会话共享数据、

session完成跨请求共享数据，但是这些请求必须在同一个会话当中、

request完成跨Servlet共享数据，但是这些Servlet必须在同一个请求当中【转发】

14.4 使用原则：有小到大尝试，优先使用小范围。

例如：登录成功之后，已经登录的状态需要保存起来，可以将登录成功的这个状态保存到session对象中。

登录成功状态不能保存到request范围中，因为一次请求对应一个新的request对象。

登录成功状态也不能保存到application范围中，因为登录成功状态是属于会话级别的，不能所有用户共享。

15、补充HttpServletRequest中的方法：

- HttpSession session = request.getSession()；获取当前的session，获取不到，则新建session
- HttpSession session = request.getSession(true)；获取当前的session，获取不到，则新建session
- HttpSession session = request.getSession(false)；获取当前的session，获取不到，则返回null