

JavaWeb

XML

第一个 XML 文件

```
<?xml version="1.0" encoding="utf-8" ?> <!--声明这是一个 xml 文件-->

<!--内容, 小型图书馆-->
<books>
    <!--第一本图书-->
    <book sn="SN123456">      <!--声明一个图书, 编号为 123456-->
        <name>人间失格</name>    <!--书名-->
        <zuozhe>太宰治</zuozhe>  <!--作者-->
        <price>75</price>        <!--价格-->
    </book>

    <!--第二本图书-->
    <book sn="SN654321">      <!--声明一个图书, 编号为 123456, sn 为 xml 属性, 追
        加描述-->
        <name>java 入门到放弃</name>  <!--书名-->
        <zuozhe>马佳盛</zuozhe>    <!--作者-->
        <price>3</price>        <!--价格-->
        <teshu><! [CDATA [
            <<我是书名号>>
        ]]></teshu>      <!--特殊字符在 cdata 中-->
    </book>
</books>
```

Idea 导入 XML 包

解

压

 dom4j-1.6.1.zip

2011/5/11 15:03

压缩(zipped)文件...

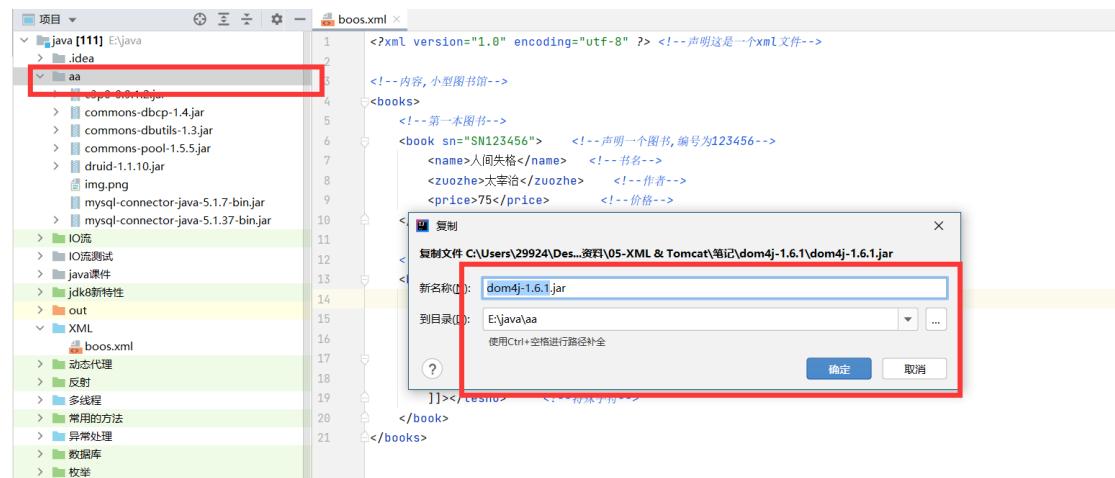
11,418 KB

复制

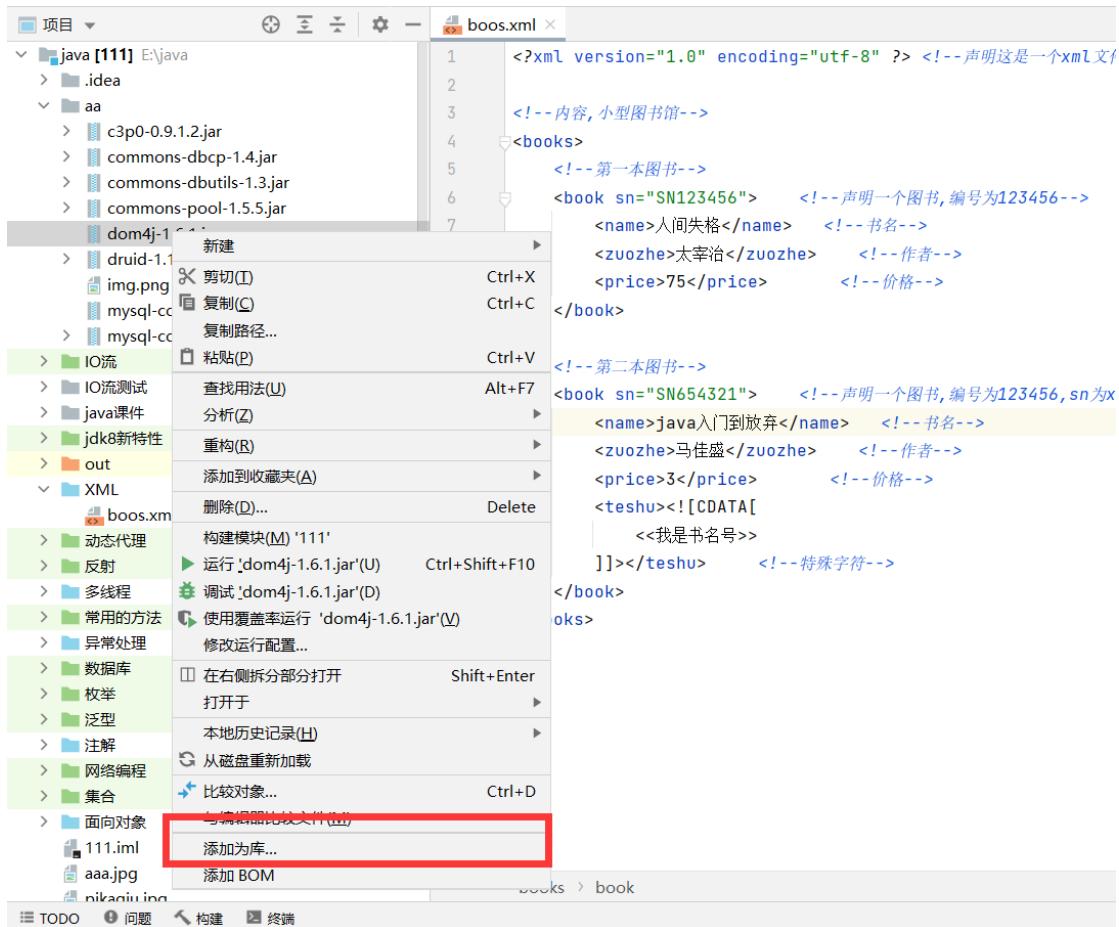
到 idea

| 名称 | 修改日期 | 类型 | 大小 |
|------------------------|-----------------|---------------|--------|
| docs | 2005/5/16 14:28 | 文件夹 | |
| lib | 2005/5/16 14:28 | 文件夹 | |
| src | 2005/5/16 14:28 | 文件夹 | |
| xdocs | 2005/5/16 14:28 | 文件夹 | |
| xml | 2005/5/16 14:28 | 文件夹 | |
| build.xml | 2005/5/16 14:28 | XML 文档 | 27 KB |
| dom4j-1.6.1.jar | 2005/5/16 14:25 | WinRAR 压缩文件 | 307 KB |
| maven.xml | 2005/5/16 14:28 | XML 文档 | 2 KB |
| project.properties | 2005/5/16 14:28 | PROPERTIES 文件 | 3 KB |
| project.xml | 2005/5/16 14:28 | XML 文档 | 10 KB |

到 idea



添加库



读取 XML

//1. 读取 XML 文件

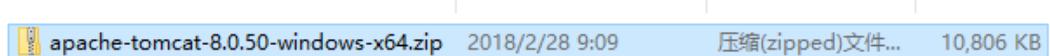
```
SAXReader reader = new SAXReader();      // 创建读取对象
Document read = reader.read("XML/boos.xml"); // 读取 xml
Element rootElement = read.getRootElement(); // 获取根元素
List<Element> elements = rootElement.elements();
for(Element element : elements){
    String name = element.elementText("name");
    String zuozhe = element.elementText("zuozhe");
    String price = element.elementText("price");
    String sn = element.attributeValue("sn"); // 获取属性名
    // 追加到 Books 类
    System.out.println(new
Books(name, zuozhe, Double.parseDouble(price)));
    // books.setName(name);
    // books.setZuozhe(zuozhe);
    // books.setPrice();
}
```

}

Tomcat

安装

解压压缩包即可



目录结构

| | | | |
|---------------|----------------|------|---------------|
| bin | 2018/2/7 22:09 | 文件夹 | 存放tomcat可执行程序 |
| conf | 2018/2/7 22:08 | 文件夹 | 存放tomcat配置文件 |
| lib | 2018/2/7 22:08 | 文件夹 | 存放tomcat的jar包 |
| logs | 2018/2/7 22:06 | 文件夹 | 日志 |
| temp | 2018/2/7 22:09 | 文件夹 | 临时文件 |
| webapps | 2018/2/7 22:09 | 文件夹 | 存放tomcat工程 |
| work | 2018/2/7 22:06 | 文件夹 | 以后慢慢体会 |
| LICENSE | 2018/2/7 22:08 | 文件 | |
| NOTICE | 2018/2/7 22:08 | 文件 | |
| RELEASE-NOTES | 2018/2/7 22:08 | 文件 | |
| RUNNING.txt | 2018/2/7 22:08 | 文本文档 | |

启动、停止 tomcat

开启

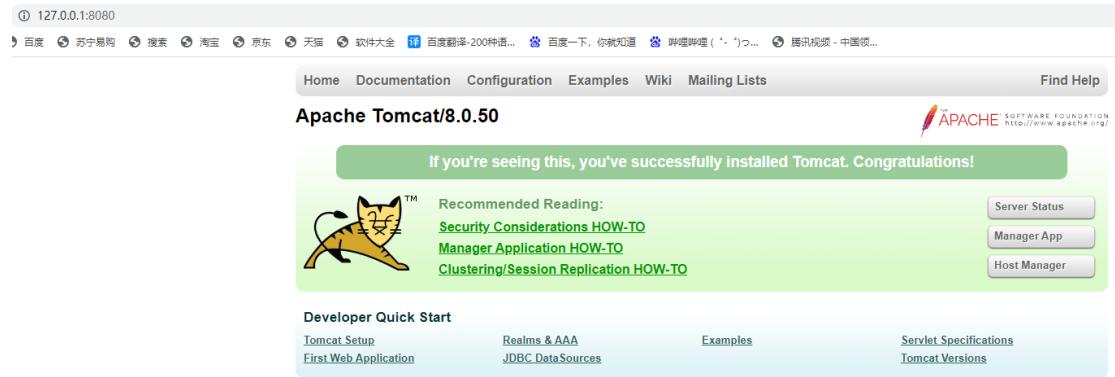
找到 lib -> startup.bat 双击运行就行了

停止

找到 lib -> shutdown.bat 双击运行就行了

测试

访问 <http://127.0.0.1:8080>



常见问题

双击运行小窗口直接消失,去环境变量找 JAVA_HOME

修改端口号

Tomcat -> conf -> server.xml

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

部署 web 项目

方式一

直接把 web 工程复制到 tomcat -> webapps 就可以

方式二

找到 tomcat -> conf -> Catalina -> locathost

创建一个自定义名 xml 文件

| 此电脑 > 学习 (E) > Tomcat > apache-tomcat-8.0.50 > conf > Catalina > localhost | | | |
|--|----------------|--------|------|
| 名称 | 修改日期 | 类型 | 大小 |
| aaa.xml | 2021/9/8 10:07 | XML 文档 | 0 KB |
| | | | |

在里面输入

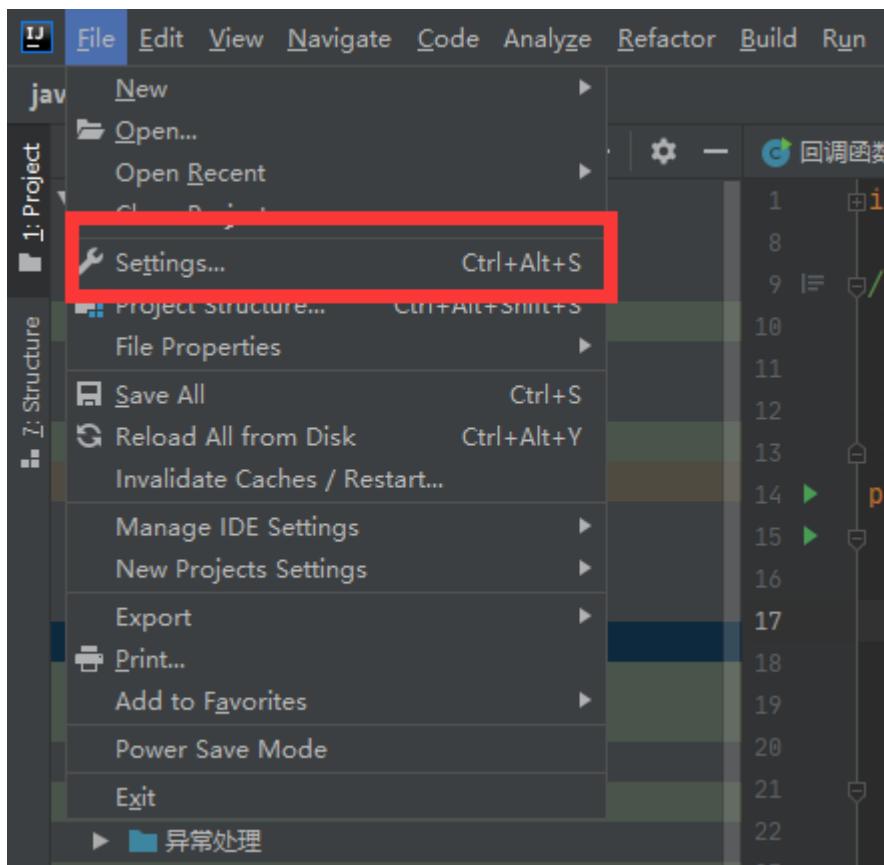


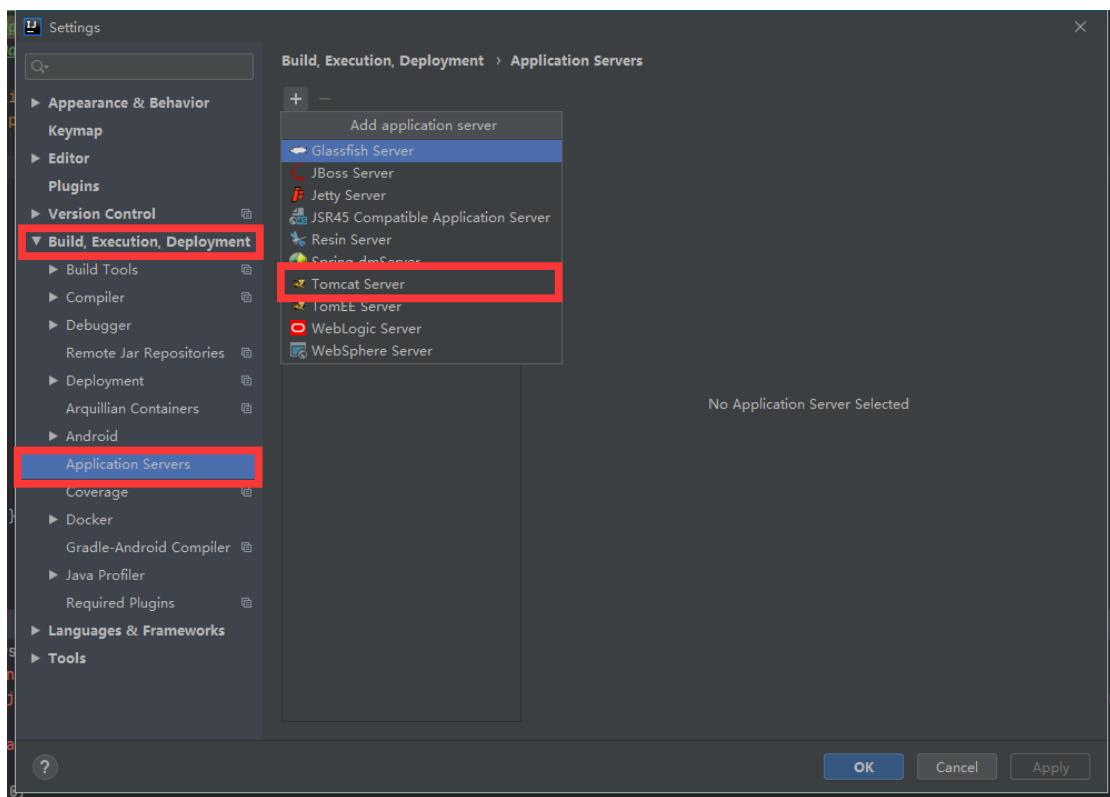
```
<Context path="/aaa" docBase="E:/book">
```

Path 为访问网页最后目录,docbase 为 web 项目工程绝对路径
测试

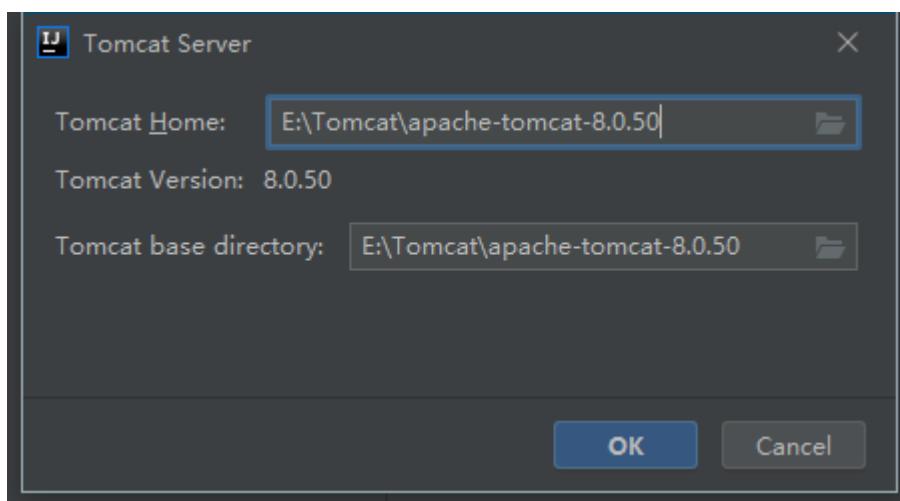
<https://127.0.0.1:8080/aaa>

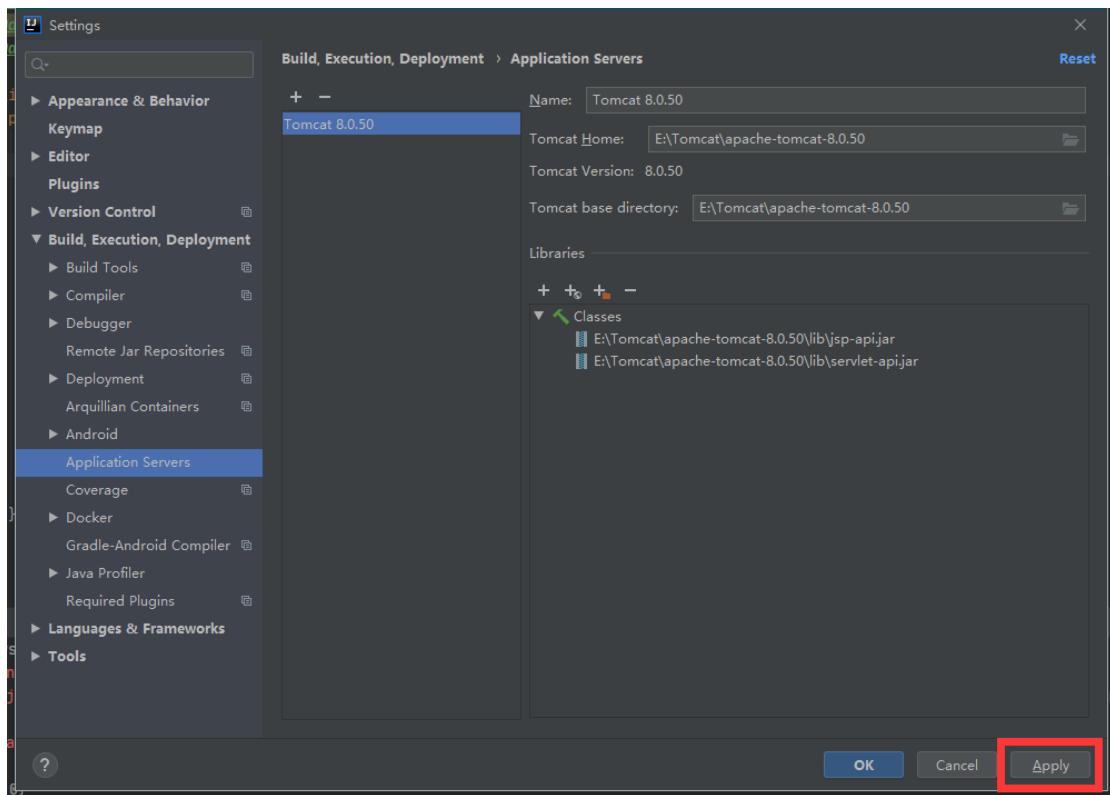
idea 配置 tomcat



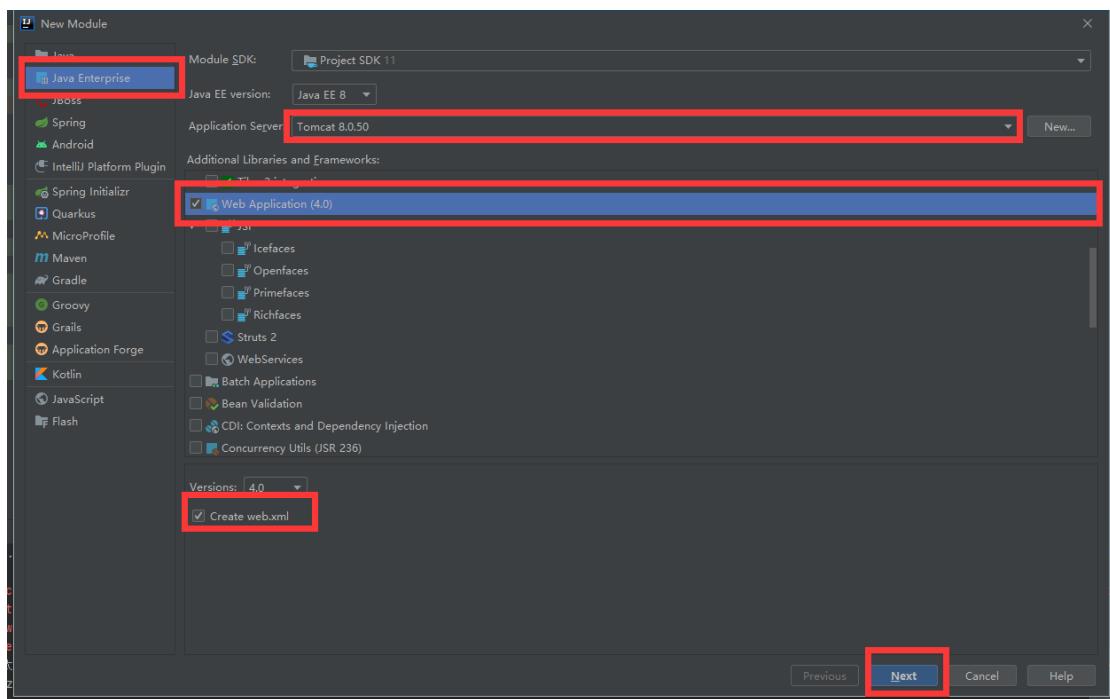


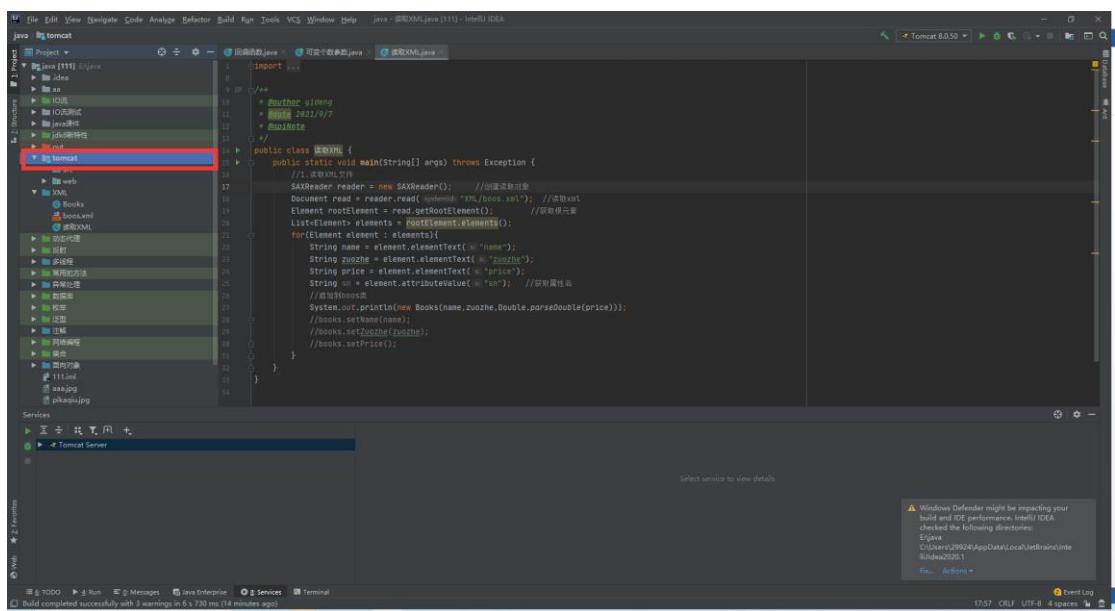
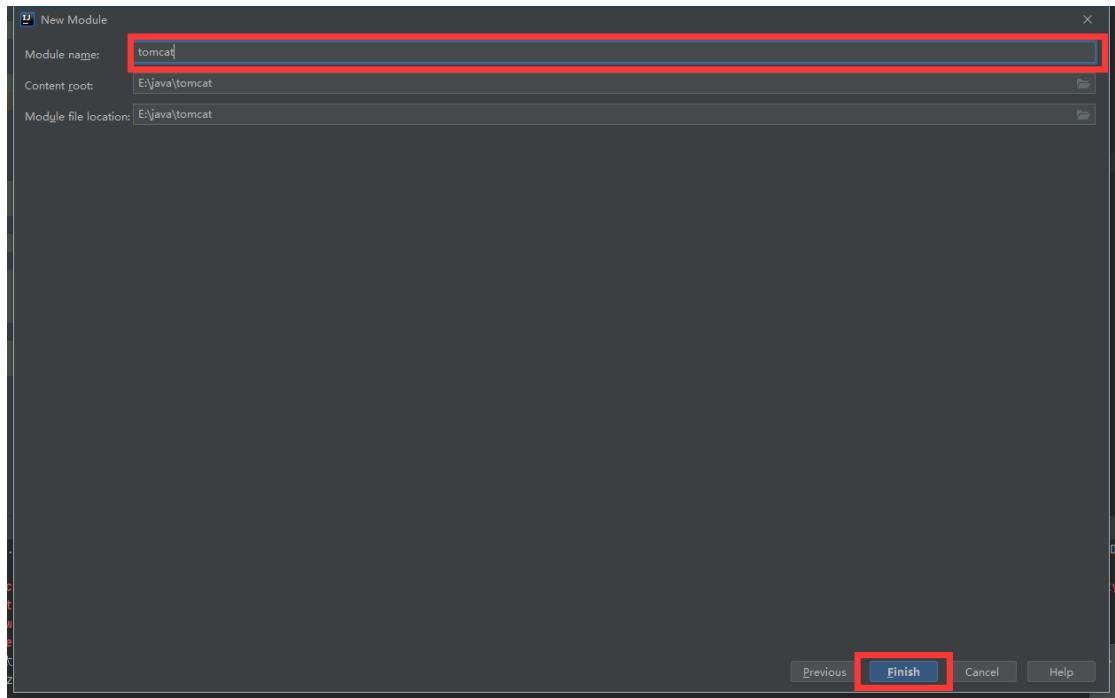
选择 tomcat 服务器路径



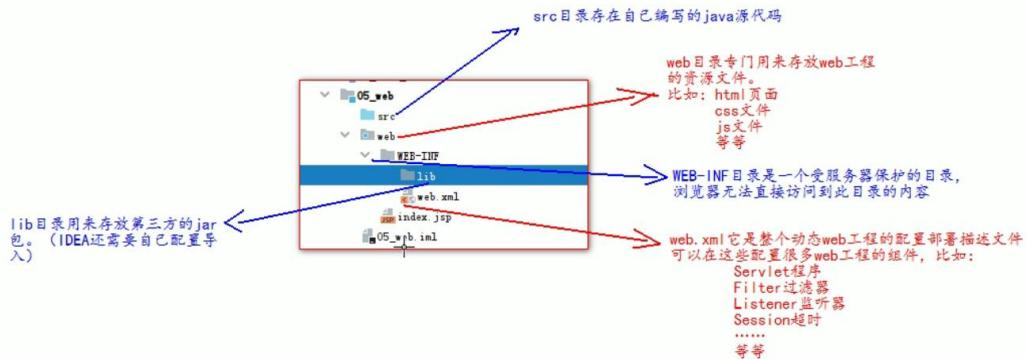


创建 tomcat





tomcat 项目目录结构



Servlet

第一个 Servlet 程序

在创建好的 tomcat 工程下 -> web -> WEB-INF -> web.xml

在里面添加

```
<servlet>
    <servlet-name>oneServlet</servlet-name>      <!--别名-->
    <servlet-class>oneServlet</servlet-class>    <!--java 类-->
</servlet>

<servlet-mapping>
    <!--对应的别名-->
    <servlet-name>oneServlet</servlet-name>
    <!--访问的路径, 必须/开头-->
    <url-pattern>/tomcat</url-pattern>
</servlet-mapping>
```

然后创建 java 类继承 Servlet 接口

```
public class oneServlet implements Servlet
```

实现接口的方法，在 service 中写入程序

```
@Override
public void service(ServletRequest servletRequest, ServletResponse servletResponse) throws
```

```
ServletException, IOException {  
    System.out.println("欢迎");  
}
```

然后运行就可以了

Servlet 生命周期

1. 构造器
2. Init 方法
3. Service 方法
4. Destroy 销毁方法

当 tomcat 服务器运行起来的时候会执行构造器方法 -> init 方法 ->service 方法,tomcat 每加载一次,service 方法都会加载一次,而构造器方法和 init 方法从运行到结束只会运行一次

获取表单 get 还是 post 请求

实现接口方式

表单填写为 tomcat 服务器运行时类的超链接,在 service 中输入

```
@Override  
public void service(ServletRequest servletRequest, ServletResponse servletResponse) throws  
ServletException, IOException {  
    //本来的 servletRequest 没有方法,需要用它的子类 HttpServletRequest  
    HttpServletRequest httpser = (HttpServletRequest) servletRequest;  
    //getMethod():获取 get 还是 post 请求  
    String method = httpser.getMethod();  
    if("POST".equals(method)){
```

```

        doPost();
    }else if("GET".equals(method)){
        doGet();
    }
}

```

继承方式

继承 `HTTPSServlet` 重写 `get` 和 `post` 方法, 自动获取表单连接方式根据方式调用 `doGet` 或 `doPost`, 去掉 `super`

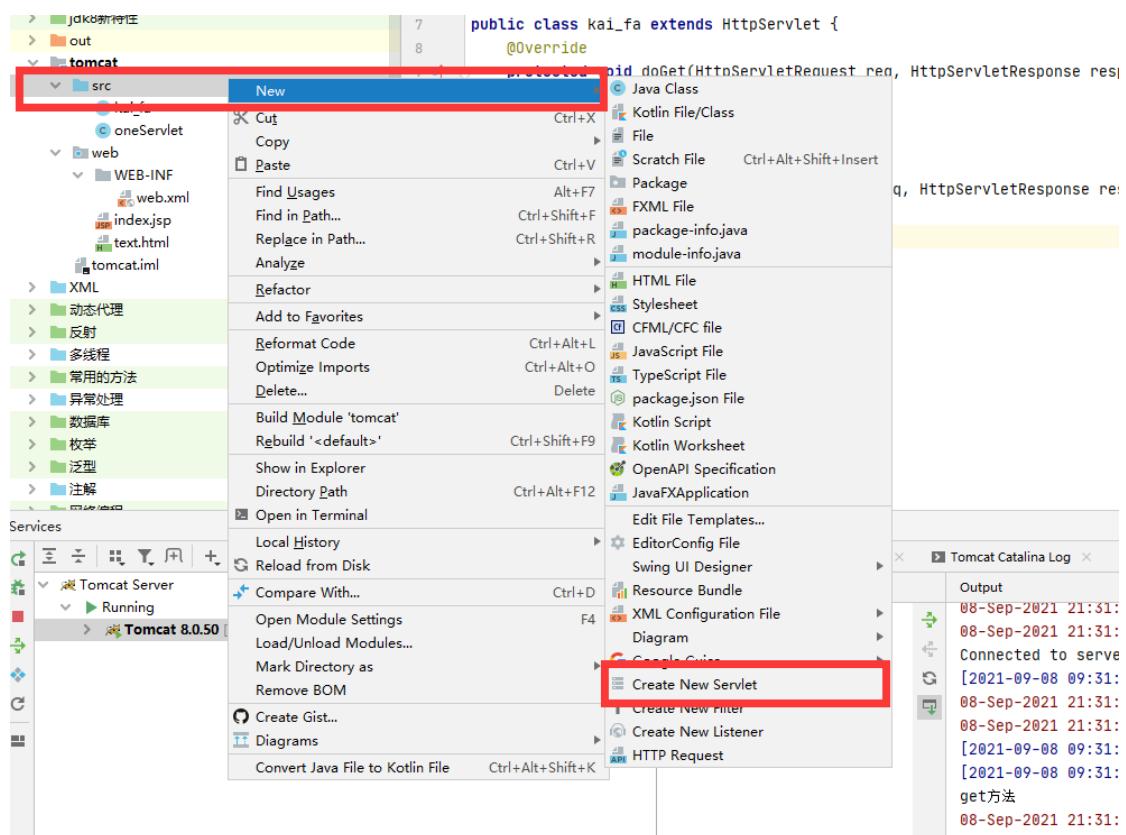
```

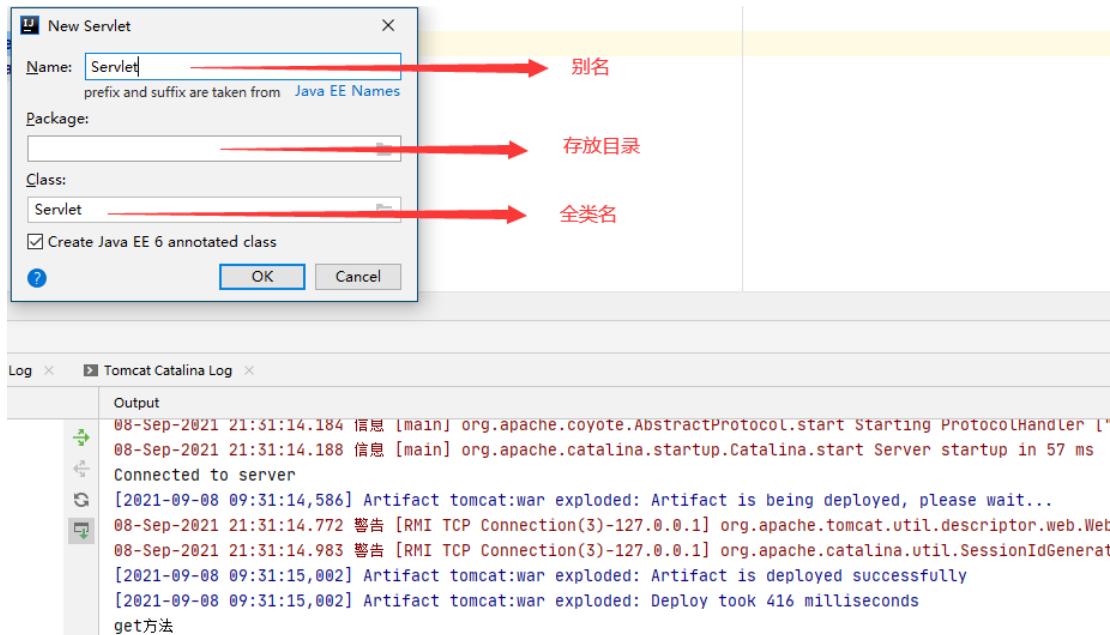
public class kai_fa extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        System.out.println("get方法");
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        System.out.println("post方法");
    }
}

```

开发版方式





然后自动创建 XML 和 java 类, XML 需要创建 Servlet-mapping

```

<servlet>
    <servlet-name>kaifa_zuizhong</servlet-name>
    <servlet-class>Servletdir.kaifa_zuizhong</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>kaifa_zuizhong</servlet-name>
    <url-pattern>/kaifa_zuizhong</url-pattern>
</servlet-mapping>

```

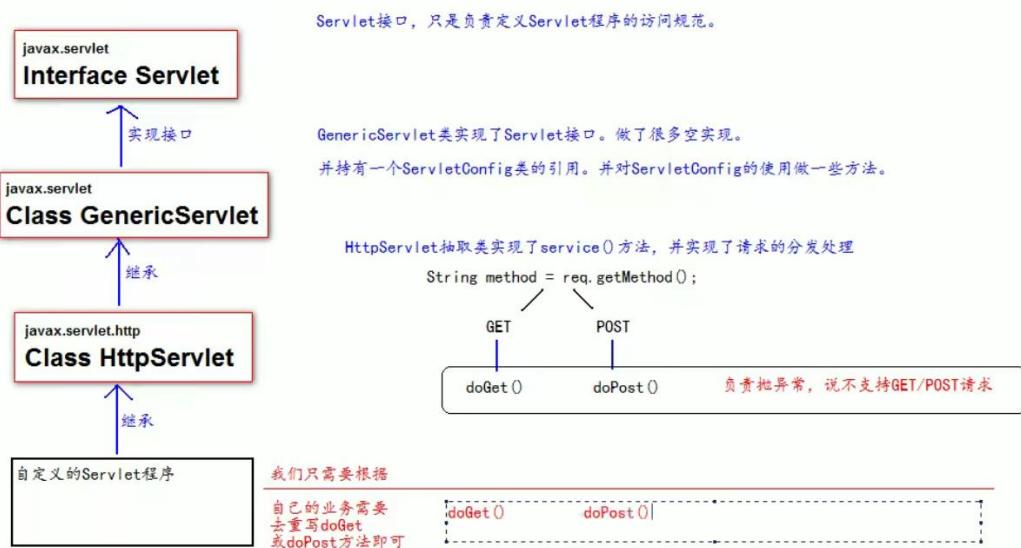
The screenshot shows the content of the web.xml file.

```

web.xml  kaifa_zuizhong.java  text.html  oneServlet.java  kai_fa.java
1 package Servletdir;
2
3 import ...
4
5 public class kaifa_zuizhong extends HttpServlet {
6     protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
7
8     }
9
10    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
11
12    }
13
14}
15
16
17
18

```

Servlet 的类结构



ServletConfig 类

ServletConfig 类的三大作用：

1. 获取 servlet 程序的别名 Servlet-name 的值

```
<servlet>
    <servlet-name>kaifa_zuizhong</servlet-name>
    <servlet-class>Servletdir.kaifa_zuizhong</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>kaifa_zuizhong</servlet-name>
    <url-pattern>/kaifa_zuizhong</url-pattern>
</servlet-mapping>
```

2. 获取初始化参数 init-param 的值

```
<!-->
<servlet>
    <servlet-name>kaifa_zuizhong</servlet-name>
    <servlet-class>Servletdir.Kaifa_zuizhong</servlet-class>
    <init-param>
        <param-name>username</param-name>
        <param-value>root</param-value>
    </init-param>
    <init-param>
        <param-name>password</param-name>
        <param-value>Qwer1234</param-value>
    </init-param>
</servlet>
```

3. 获取 ServletContext 对象

实现：重写 init 必须要 super 父类的 init 方法

```
@Override
public void init(ServletConfig config) throws ServletException {
    super.init(config);
    // 获取 XML 的别名
    String servletName = config.getServletName();
    // 获取 init-param
    String username = config.getInitParameter("username");
    String password = config.getInitParameter("password");
    // 获取 ServletContext 对象
    ServletContext servletContext = config.getServletContext();
    System.out.println("别名为：" + servletName + "\n用户名为：" + username + "\npassword：" + password + "\n" + servletContext);
}
```

ServletContext 类

ServletContext 类的四大作用

1. 获取 web.xml 文件中的 context-param

```
<servlet>
    <servlet-name>serCountext</servlet-name>
    <servlet-class>Servletdir.serCountext</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>serCountext</servlet-name>
    <url-pattern>/serCountext</url-pattern>
</servlet-mapping>
<context-param>
    <param-name>username</param-name>
    <param-value>root</param-value>
</context-param>
<context-param>
    <param-name>password</param-name>
    <param-value>Qwer1234</param-value>
</context-param>
```

2. 获取当前工程路径, 格式:/工程路径

3. 获取工程部署后的服务器硬盘上的绝对路径

4. 存储键值对集合

5. 实现

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    ServletContext servletContext = getServletContext();
    // 获取xml context 的信息
    String username = servletContext.getInitParameter("username");
    String password = servletContext.getInitParameter("password");
    // 获取工程相对路径
    String contextPath = servletContext.getContextPath();
    // 获得绝对路径
    String realPath = servletContext.getRealPath("/WEB-INF/web.xml");
    System.out.println("username:" + username + "\npassword:" + password + "\n项目的相对路径是：" + contextPath + "\nweb.xml的绝对路径是" + realPath);
}
```

存储 key-value 的值, 这个是整个项目的, java1 类调用能用, java2 调用也能用, 知道 tomcat 服务器停止

SetAttribute(key,value) // 设置一个 map

GetAttribute(key,value) // 获取一个 map

RemoveAttribute(key) // 删除一个 map

```
context.setAttribute(name: "key1", object: "value1");
```

HttpServletRequest 类

作用:

每次访问 tomcat 服务器, tomcat 服务器就会把请求过来的 http 信息封装到 request 对象中, 然后传递到 service 方法 (doget,dopost) 让我们使用, 我们可以通过 HttpServletRequest

获取所有请求的信息

常用方法:

| | |
|-------------------------|-------------|
| getRequestURI() | 获取请求的资源路径 |
| getRequestURL() | 获取请求的绝对路径 |
| getRemoteHost() | 获取客户端 ip 地址 |
| getHeader() | 获取请求头 |
| getParameter() | 获取请求参数 |
| getParameterValues() | 获取请求参数(多个值) |
| setAttribute(key-value) | 设置域数据 |
| getAttribute(key) | 获取域数据 |
| getRequestDispatcher() | 获取请求转发对象 |

实现:如果是 post 请求,值有中文会出现乱码

```
<form action="http://localhost:8080/tomcat/kaifa_zuizhong" method="GET">
    账户:<input type="text" name="username"><br>
    密码:<input type="password" name="password"><br>
    性别:<input type="radio" name="sex" value="nan">男
        <input type="radio" name="sex" value="nv">女<br>
    兴趣爱好:<input type="checkbox" value="java" name="aihao">java
        <input type="checkbox" value="C++" name="aihao">C++
        <input type="checkbox" value="javascript" name="aihao">javascript<br>
    <input type="submit" value="提交" |>
</form>
```

```

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    //获取类的相对路径
    System.out.println("uri:"+request.getRequestURI());
    //获取请求的网页地址
    System.out.println("\nurl:"+request.getRequestURL());
    //获取请求客户端的ip地址
    System.out.println("\nhostname:"+request.getRemoteHost());
    //获取客户端的请求头(string)放入想要得到的信息
    System.out.println("\n获取请求头"+request.getHeader("User-Agent"));
    //获取请求方式 get || post
    System.out.println("\n请求方式为:"+request.getMethod());
}

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    //获取表单内的值
    String username = request.getParameter("username");
    String password = request.getParameter("password");
    String sex = request.getParameter("sex");
    String[] aihao = request.getParameterValues("aihao");
    String aihao ="";
    for (int i = 0; i <aihao.length ; i++) {
        aihao+=aihao[i]+" ";
    }
    System.out.println("账号:"+username+"\n密码:"+password+"\n性别:"+sex+"\n兴趣爱好:"+aihao);
}

```

解决乱码问题,在 doPost 中修改编码,必须卸载获取值前面

```

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    request.setCharacterEncoding("UTF-8");
}

```

HttpServletResponse 类

作用:

和 HttpServletRequest 一样,HttpServletResponse 是客户端给服务器发送,而 HttpServletResponse 是服务器对客户端的响应

输出流:

字节流 `getOutputStream()` 用于下载

字符流 `getWriter()` 用于回传数据

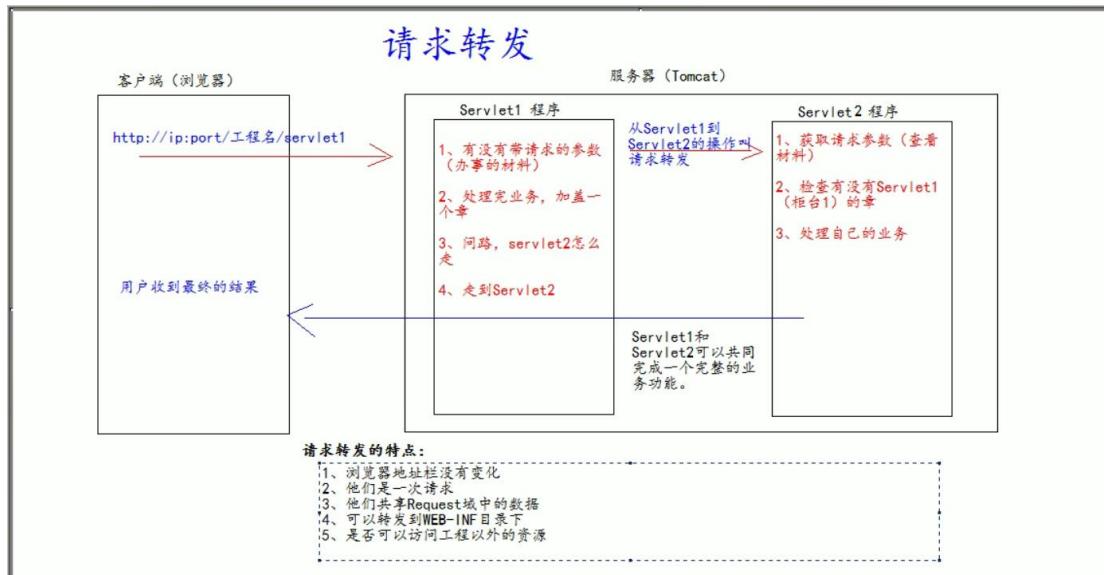
两个流只能选择一个

```

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    // 设置服务器, 客户端, 响应头都为utf-8
    response.setContentType("text/html; charset=UTF-8");
    // 获取字节流
    PrintWriter writer = response.getWriter();
    // 发送数据
    writer.write(s: "今天天气好晴朗");
}

```

Servlet 请求转发过程



实现:准备 2 个 Servlet 程序



Servlet1

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    //验证身份
    String username = request.getParameter("username");
    System.out.println("正在验证身份,身份为:" + username);
    //把数据存入域
    request.setAttribute("username", username);
    //获取Servlet的路径
    RequestDispatcher requestDispatcher = request.getRequestDispatcher("/Servlet2");
    //跳转到Servlet2
    requestDispatcher.forward(request, response);
}
```

Servlet2

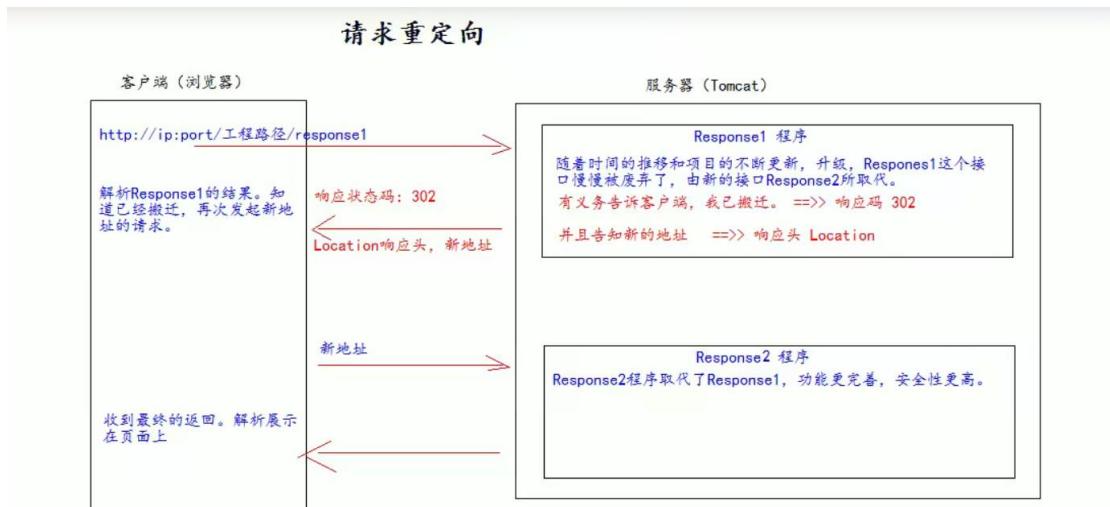
```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    String username = request.getParameter("username");
    System.out.println("Servlet2验证身份,身份为:" + username);

    //二次验证
    Object username1 = request.getAttribute("username");
    System.out.println("二次验证身份,身份为:" + username1);

    System.out.println("验证成功");
}
```

重定向

客户端向服务器发送一个连接,但是服务器的第一代程序不能用了,服务器就要告诉客户端去找第二代程序,然后客户端通过服务器返回的结果去访问第二代程序,第二代程序返回内容



实现：



Servlet1

Servlet2

```

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    response.setContentType("text/html; charset=UTF-8");
    response.getWriter().write( s: "成功访问程序2" );
}

```

JSP

作用：

Response 回传数据比较繁琐,jsp 就取代 response 回传给客户端数据

永远放在 web 目录下

Response 回传数据

```
public class PrintHtml extends HttpServlet {

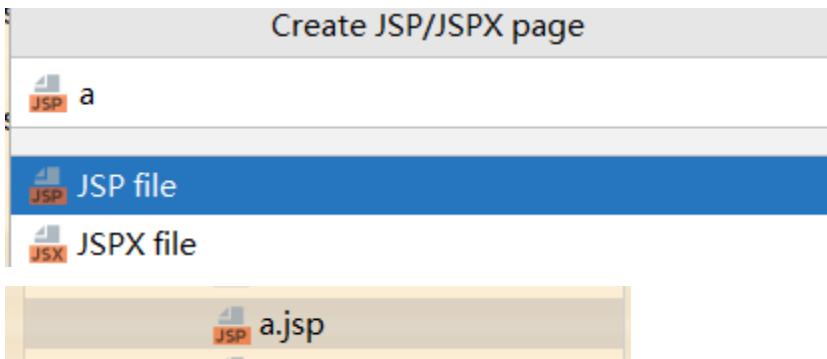
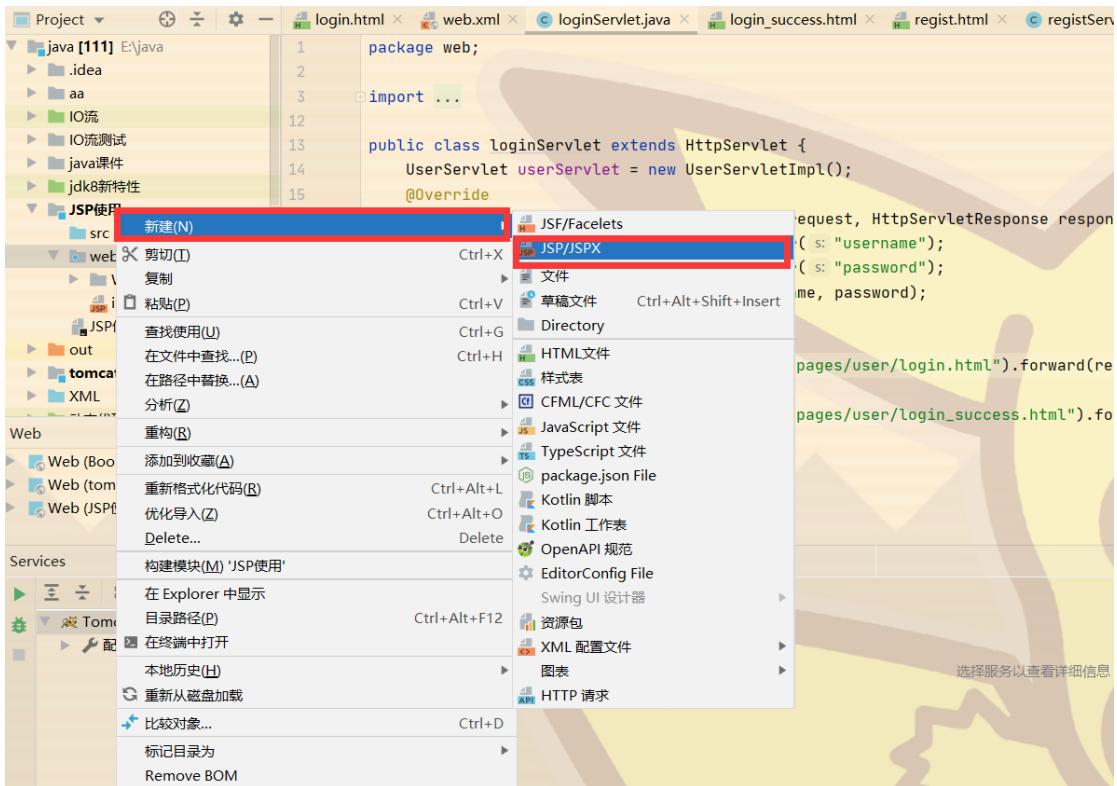
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
IOException {
        // 通过响应的回传流回传 html 页面数据

        resp.setContentType("text/html; charset=UTF-8");

        PrintWriter writer = resp.getWriter();

        writer.write("<!DOCTYPE html>\r\n");
        writer.write("  <html lang=\"en\">\r\n");
        writer.write("    <head>\r\n");
        writer.write("      <meta charset=\"UTF-8\">\r\n");
        writer.write("      <title>Title</title>\r\n");
        writer.write("    </head>\r\n");
        writer.write("    <body>\r\n");
        writer.write("      这是 html 页面数据 \r\n");
        writer.write("    </body>\r\n");
        writer.write("</html>\r\n");
        writer.write("\r\n");
    }
}
```

创建 jsp 文件



访问地址

a. jsp 页面 访问地址是 =====>>>> <http://ip:port/工程>

路径/a.jsp

JSP 标签头

1. language 属性 表示 jsp 翻译后是什么语言文件。暂时只支

持 java。

2. `contentType` 属性 表示 jsp 返回的数据类型是什么。也是源码中 `response.setContentType()`参数值

3. `encoding` 属性 表示当前 jsp 页面文件本身的字符集。

4 `import` 属性 跟 java 源代码中一样。用于导包，导类。

=====两个属性是给 `out` 输出流使用

=====

5. `autoFlush` 属性 设置当 `out` 输出流缓冲区满了之后，是否自动刷新缓冲区。默认值是 `true`。

6. `vi. buffer` 属性 设置 `out` 缓冲区的大小。默认是 8kb 缓冲区溢出错误：

=====两个属性是给 `out` 输出流使用

=====

7. `errorPage` 属性 设置当 jsp 页面运行时出错，自动跳转去的错误页面路径。

8. `isErrorPage` 属性 设置当前 jsp 页面是否是错误信息页面。默认是 `false`。如果是 `true` 可以 获取异常信息。

9. `session` 属性 设置访问当前 jsp 页面，是否会创建 `HttpSession` 对象。默认是 `true`。

10. `xextends` 属性 设置 jsp 翻译出来的 java 类默认继承谁

```
<%@ page import="java.util.Map" %>
<%@ page contentType="text/html;charset=utf-8" | language="java" %>
```

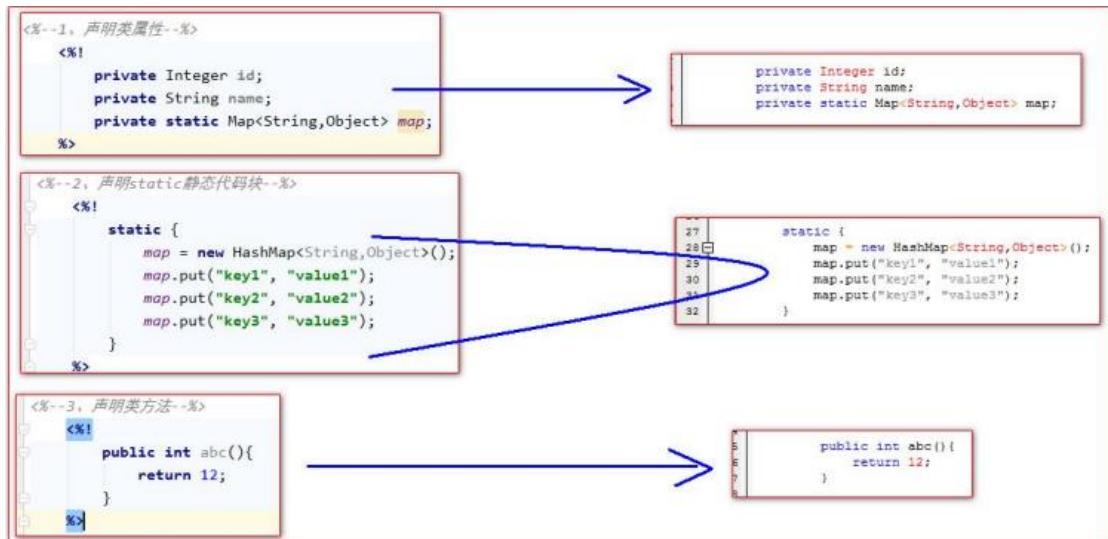
声明脚本

JSP 页面

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%--1、声明类属性--%>
<%!
    private Integer id;
    private String name;
    private static Map<String, Object> map;
%>
<%--2、声明 static 静态代码块--%>
<%!
    static {
        map = new HashMap<String, Object>();
        map.put("key1", "value1");
        map.put("key2", "value2");
        map.put("key3", "value3");
    }
%>
```

```
<%--3、声明类方法--%>
<%!
    public int abc() {
        return 12;
    }
%>
<%--4、声明内部类--%>
<%!
    public static class A {
        private Integer id = 12;
        private String abc = "abc";
    }
%>
```

对比



表达式脚本(常用)

表达式脚本的格式是：

```
<%= 表达式 %>
```

表达式脚本的作用是：

在 jsp 页面上输出数据

表达式脚本的特点是

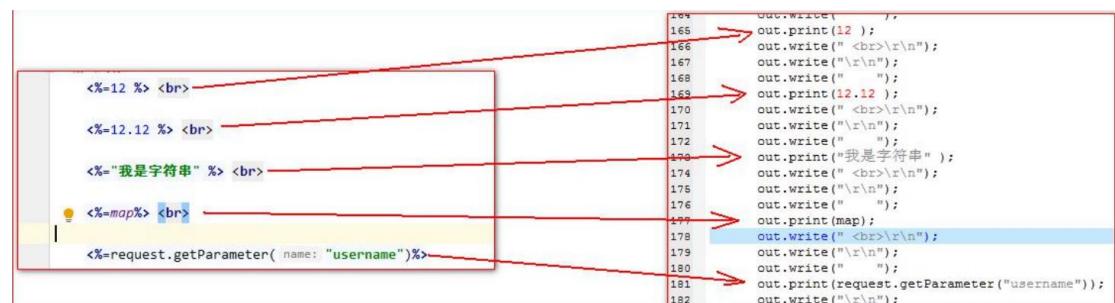
- 1、所有的表达式脚本都会被翻译到 `_jspService()` 方法中
- 2、表达式脚本都会被翻译成为 `out.print()` 输出到页面上
- 3、由于表达式脚本翻译的内容都在 `_jspService()` 方法中,所以 `_jspService()` 方法中的对象都可以直接使用。
- 4、表达式脚本中的表达式不能以分号结束

练习：

1. 输出整型
2. 输出浮点型
3. 输出字符串
4. 输出对象

```
<%=12 %> <br>
<%=12.12 %> <br>
<%="我是字符串" %> <br>
<%=map%> <br>
<%=request.getParameter("username")%>
```

底层代码对比



代码脚本

代码脚本的格式是：

<%

 Java 语句

%>

代码脚本的作用是：可以在 jsp 页面中，编写我们自己需要

的功能（写的是 java 语句）

代码脚本的特点是：

- 1、代码脚本翻译之后都在 `_jspService` 方法中
- 2、代码脚本由于翻译到 `_jspService()` 方法中，所以在 `_jspService()` 方法中的现有对象都可以直接使用。
- 3、还可以由多个代码脚本块组合完成一个完整的 java 语句。
- 4、代码脚本还可以和表达式脚本一起组合使用，在 `jsp` 页面上输出数

练习：

1. 代码脚本----if 语句

```
<%--练习：--%>
<%--1. 代码脚本----if 语句--%>
<%
    int i = 13 ;
    if (i == 12) {
%>
<h1>国哥好帅</h1>
<%
} else {
%>
<h1>国哥又骗人了！</h1>
<%
}
%>
<br>
```

2. 代码脚本----for 循环语句

```
<%--2.代码脚本----for 循环语句--%>
<table border="1" cellspacing="0">
    <%
        for (int j = 0; j < 10; j++) {
    %>
    <tr>
        <td>第 <%=j + 1%>行</td>
    </tr>
    <%
        }
    %>
</table>
```

3. 翻译后 java 文件中 _jspService 方法内的代码都可以

```
<%--3.翻译后 java 文件中_jspService 方法内的代码都可以写--%>
<%
    String username = request.getParameter("username");
    System.out.println("用户名的请求参数值是：" + username);
%>
```

三种注释

```
<!-- 这是html注释 -->
<%
    // 单行java注释
    /* 多行java注释 */
%>

<%-- 这是jsp注释 --%>
```

JSP 中的九大内置对象

```
public void _jspService(final javax.servlet.http.HttpServletRequest request, final javax.servlet.http.HttpServletResponse response)
throws java.io.IOException, javax.servlet.ServletException {
final javax.servlet.jsp.PageContext pageContext;
javax.servlet.http.HttpSession session = null;
java.lang.Throwable exception = org.apache.jasper.runtime.JspRuntimeLibrary.getThrowable(request);
if (exception != null) {
    response.setStatus(javax.servlet.http.HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
}
final javax.servlet.ServletContext application;
final javax.servlet.ServletConfig config;
javax.servlet.jsp.JspWriter out = null;
final java.lang.Object page = this;
```

jsp的九大内置对象

| | |
|-------------|------------------|
| request | 请求对象 |
| response | 响应对象 |
| pageContext | jsp的上下文对象 |
| session | 会话对象 |
| application | ServletContext对象 |
| config | ServletConfig对象 |
| out | jsp输出流对象 |
| page | 指向当前jsp的对象 |
| exception | 异常对象 |

jsp 四大域对象 四个域对象分别是:

说明:

域对象是可以像 Map 一样存取数据的对象。四个域对象功能一样。不同的是它们对数据的存取范围。虽然四个域对象都可以存取数据。

在使用上它们是有优先顺序的。 四个域在使用的时候，优先顺序分别是，他们从小到大的范围的顺序。

pageContext ==>>> request ==>>> session

pageContext (**PageContextImpl** 类) 当前 **jsp** 页面范围内有效

request (**HttpServletRequest** 类) 一次请求内有效

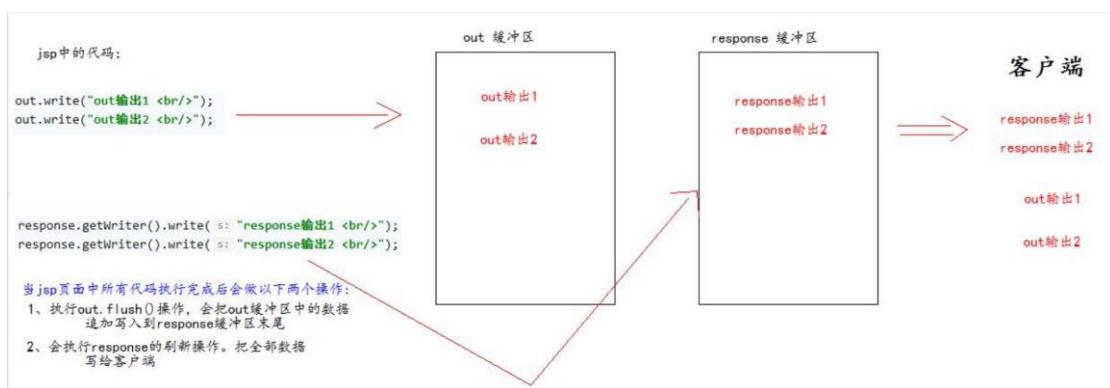
session (**HttpSession** 类) 一个会话范围内有效 (打开浏览器访问服务器, 直到关闭浏览器)

application (**ServletContext** 类) 整个 **web** 工程范围内都有效 (只要 **web** 工程不停止, 数据都在)

jsp 中的 **out** 输出和 **response.getWriter** 输出的区别

说明:

response 中表示响应, 我们经常用于设置返回给客户端的内容 (输出) **out** 也是给用户做输出使用的



由于 **jsp** 翻译之后, 底层源代码都是使用 **out** 来进行输出, 所以一般情况下。我们在 **jsp** 页面中统一使用 **out** 来进行输出。避免打乱页面输出内容的顺序。

out.write() 只能输出字符串

`out.print()` 所有类型都转换为字符串,什么都可以输出

jsp 的常用标签

jsp 静态包含

说明:

当一个网页跳转到另一个网页时,你第一个网页有些内容和第二个网页一样,就可以使用静态包含,一改全改。

使用:

```
<% include file="/全部要使用的统一网页">
```

jsp 动态包含

说明:

跟静态包含作用一样,只不过底层不一样,静态包含直接把网页 2 的源代码拿过来使用,而动态包含是把网页一的内置对象传递给网页二,这样网页二输出的内容就到了网页一的缓冲区,从而达到了拼接的作用

使用:

```
<jsp:include page="/全部使用的统一网页">
```

```
  <jsp:param name="name" value="bbj"> //传递给第二个
```

页面的信息

```
//第二个页面获取方式  
<%=requestgetParameter("name")>  
</jsp:include>
```

转发

```
<jsp:forward page="跳转的网页"></jsp:forward>
```

Servlet 和 jsp 之间的值传递

Servlet

```
req.setAttribute("stuList", studentList);
```

JSP

```
<%  
    List studentList = (List) request.getAttribute("stuList");  
%>
```

ServletContextListener 监听器

说明:

ServletContextListener 它可以监听 ServletContext 对象的创建和销毁。

`ServletContext` 对象在 web 工程启动的时候创建，在 web 工程停止的时候销毁。

监听到创建和销毁之后都会分别调用 `ServletContextListener` 监听器的方法反馈

使用：

1、编写一个类去实现 `ServletContextListener`

2、实现其两个回调方法

3、到 `web.xml` 中去配置监听

监听器实现类：

```
public class MyServletContextListenerImpl implements ServletContextListener {  
  
    @Override  
    public void contextInitialized(ServletContextEvent sce) {  
        System.out.println("ServletContext 对象被创建了");  
    }  
  
    @Override  
    public void contextDestroyed(ServletContextEvent sce) {  
        System.out.println("ServletContext 对象被销毁了");  
    }  
}
```

`web.xml` 中的配置：

```
<!-- 配置监听器-->  
<listener>  
    <listener-class>com.atguigu.listener.MyServletContextListenerImpl</listener-class>  
</listener>
```

EL 表达式

什么是 EL 表达式，EL 表达式的作用：

EL 表达式的全称是：Expression Language。是表达式语言。

EL 表达式的什么作用: EL 表达式主要是代替 jsp 页面中的表达式脚本在 jsp 页面中进行数据的输出。

因为 EL 表达式在输出数据的时候,要比 jsp 的表达式脚本要简洁很多。

EL 表达式搜索域数据的顺序

pageContext request session application

EL 表达式输出

```
Person 类
public class Person {
    // i. 需求—输出Person类中普通属性，数组属性。List集合属性和map集合属性。
    private String name;
    private String[] phones;
    private List<String> cities;
    private Map<String, Object> map;

    public int getAge() {
        return 18;
    }
}
```

JSP 页面输出

```
<%  
Person person = new Person();  
person.setName("国哥好帅！");  
person.setPhones(new String[]{"18610541354", "18688886666", "18699998888"});  
  
List<String> cities = new ArrayList<String>();  
cities.add("北京");  
cities.add("上海");  
cities.add("深圳");  
person.setCities(cities);  
  
Map<String, Object> map = new HashMap<>();  
map.put("key1", "value1");  
map.put("key2", "value2");  
map.put("key3", "value3");  
person.setMap(map);
```

```
pageContext.setAttribute("p", person);  
%>  
  
输出 Person: ${ p }<br/>  
输出 Person 的 name 属性: ${p.name} <br>  
输出 Person 的 phones 数组属性值: ${p.phones[2]} <br>  
输出 Person 的 cities 集合中的元素值: ${p.cities} <br>  
输出 Person 的 List 集合中个别元素值: ${p.cities[2]} <br>  
输出 Person 的 Map 集合: ${p.map} <br>  
输出 Person 的 Map 集合中某个 key 的值: ${p.map.key3} <br>  
输出 Person 的 age 属性: ${p.age} <br>
```

Empty 运算

empty 运算可以判断一个数据是否为空，如果为空，则输出 true,不为空输出 false。

以下几种情况为空：

- 1、值为 null 值的时候，为空
- 2、值为空串的时候，为空
- 3、值是 Object 类型数组，长度为零的时候
- 4、list 集合，元素个数为零
- 5、map 集合，元素个数为零

```

<body>
    <%
        // 1、值为 null 值的时候，为空
        request.setAttribute("emptyNull", null);
        // 2、值为空串的时候，为空
        request.setAttribute("emptyStr", "");
        // 3、值是 Object 类型数组，长度为零的时候
        request.setAttribute("emptyArr", new Object[]{} );
        // 4、List 集合，元素个数为零
        List<String> list = new ArrayList<>();
        list.add("abc");
        request.setAttribute("emptyList", list);
        // 5、map 集合，元素个数为零
        Map<String, Object> map = new HashMap<String, Object>();
        map.put("key1", "value1");
        request.setAttribute("emptyMap", map);
    %>
    ${empty emptyNull } <br/>
    ${empty emptyStr } <br/>
    ${empty emptyArr } <br/>
    ${empty emptyList } <br/>
    ${empty emptyMap } <br/>
</body>

```

EL 表达式的 11 个隐含对象

| 变量 | 类型 | 作用 |
|-------------|-----------------|---|
| pageContext | PageContextImpl | 它可以获取 jsp 中的九大内置对象 pageScope |
| | | Map 它可以获取 pageContext 域中的数据 requestScope |
| | Map | 它可以获取 Request 域中的数据 sessionScope |
| | Map | 它 可 以 获 取 sessionScope 域中的数据 applicationScope |
| | Map | 它 可 以 获 取 applicationScope 域中的数据 param |
| | Map | 它 可 以 获 取 applicationScope 域中的数据 paramValues |
| paramValues | Map | 它 也 可 以 获 取 applicationScope 域中的数据 paramValues 的值，获取多个值的时候使用。 |
| Header | Map | 它 可 以 获 取 applicationScope 域中的数据 Header 的信息 |

| | | |
|--------------|-----|-----------------------------|
| headerValues | Map | 它可以获取请求头的信息， 它可以获取多个值的情况 |
| cookie | Map | 它可以获取当前请求的 Cookie 信息 |

initParam Map 它可以获取在 web.xml 中配置的上下文参数

EL 获取四个特定域中的属性

pageScope ===== pageContext 域
requestScope ===== Request 域
sessionScope ===== Session 域
applicationScope ===== ServletContext 域

```
<body>
<%
    pageContext.setAttribute("key1", "pageContext1");
    pageContext.setAttribute("key2", "pageContext2");
    request.setAttribute("key2", "request");
    session.setAttribute("key2", "session");
    application.setAttribute("key2", "application");
%>
${ applicationScope.key2 }
</body>
```

pageContext 对象的使用

1. 协议:
2. 服务器 ip:
3. 服务器端口:
4. 获取工程路径:
5. 获取请求方法:

6. 获取客户端 ip 地址:

7. 获取会话的 id 编号

```
<%  
    pageContext.setAttribute("req", request);  
%>  
<%=request.getScheme() %> <br>  
  
1. 协议: ${ req.scheme }<br>  
2. 服务器 ip: ${ pageContext.request.serverName }<br>  
3. 服务器端口: ${ pageContext.request.serverPort }<br>  
4. 获得工程路径: ${ pageContext.request.contextPath }<br>  
5. 获得请求方法: ${ pageContext.request.method }<br>  
6. 获得客户端 ip 地址: ${ pageContext.request.remoteHost }<br>  
7. 获得会话的 id 编号: ${ pageContext.session.id }<br>  
  
</body>
```

Param 和 paramValues

Param 获取单个的值 比如说账号、密码、性别

paramValues 获取多个值 比如说兴趣爱好

```
输出请求参数 username 的值: ${ param.username } <br>  
输出请求参数 password 的值: ${ param.password } <br>  
  
输出请求参数 username 的值: ${ paramValues.username[0] } <br>  
输出请求参数 hobby 的值: ${ paramValues.hobby[0] } <br>  
输出请求参数 hobby 的值: ${ paramValues.hobby[1] } <br>
```

Header 和 headerValues

Header 获取标签单个值

HeaderValues 获取标签多个值, 不常用

示例代码:

```
输出请求头【User-Agent】的值: ${ header['User-Agent'] } <br>  
输出请求头【Connection】的值: ${ header.Connection } <br>  
输出请求头【User-Agent】的值: ${ headerValues['User-Agent'][0] } <br>
```

Cookie

示例代码:

```
获取 Cookie 的名称: ${ cookie.JSESSIONID.name } <br>
获取 Cookie 的值: ${ cookie.JSESSIONID.value } <br>
```

initParam

获取 XML 中 context-param 的值

web.xml 中的配置:

```
<context-param>
    <param-name>username</param-name>
    <param-value>root</param-value>
</context-param>

<context-param>
    <param-name>url</param-name>
    <param-value>jdbc:mysql:///test</param-value>
</context-param>
```

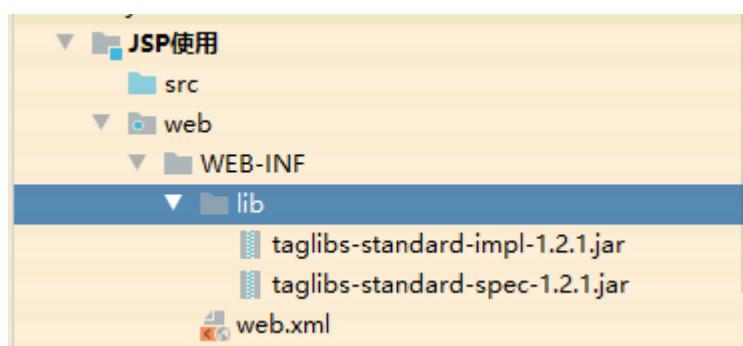
示例代码:

```
输出<Context-param>username 的值: ${ initParam.username } <br>
输出<Context-param>url 的值: ${ initParam.url } <br>
```

JSTL 标签库

使用步骤:

1. idea 导入标签库包



2. 在 jsp 页面中使用 taglib 指令引入标签库

core 核心库使用

<c:set />:可以往域中存储数据

Scope 为往哪个域 var 为 key value 为 value

```
<c:set scope="session" var="abc" value="abcValue"/>
```

```
$(sessionScope.abc)                //打印这个值
```

<c:if>:if 判断只能判断一次

```
<c:if test="$(12 == 12)">  
    <h1>等于</h1>  
</c:if>  
  
<c:if test="$(12 != 12)">  
    <h1>不等于</h1>  
</c:if>
```

<c:choose> <c:when> <c:otherwise>标签

相当于 java 的 switch case default

choose 标签开始选择判断

when 标签表示每一种判断情况

test 属性表示当前这种判断情况的值

otherwise 标签表示剩下的情况

JSP 代码

```
<%
    request.setAttribute("height", 180);
%>

<c:choose>
    <c:when test="${requestScope.height > 190}">
        <h2>小巨人</h2>
    </c:when>
    <c:when test="${requestScope.height > 180}">
        <h2>很高</h2>
    </c:when>
    <c:when test="${requestScope.height > 170}">
        <h2>还可以</h2>
    </c:when>
</c:choose>
```

注意：

- 1、标签里不能使用 html 注释，要使用 jsp 注释
- 2、when 标签的父标签一定要是 choose 标签

<c:forEach />:for 循环

1. 循环 1 到 10

Item 循环的集合、数组

begin 属性设置开始的索引

end 属性设置结束的索引

step 设置步长值(var 一次加几)

var 属性表示循环的变量(也是当前正在遍历到的数据)

varStatus="status" 获 取 获 取 到 的 数 据 信 息

```
public interface LoopTagStatus {  
    public Object getCurrent(); → 表示获取当前遍历到的数据  
    public int getIndex(); → 表示获取遍历的索引  
    public int getCount(); → 表示遍历的个数  
    public boolean isFirst(); → 表示当前遍历的数据是否是第一条，  
    public boolean isLast(); → 或最后一条  
    public Integer getBegin();  
    public Integer getEnd();  
    public Integer getStep(); → [获取begin, end, step属性值]  
}
```

```
<c:forEach begin="1" end="10" var="i">
```

```
    ${i}
```

2. 遍历集合、数组

```
<%  
request.setAttribute("arr", newString []  
{"18610541354","18688886666","18699998888"});
```

```
%>

<c:forEach item="${requestScope.arr}" var="item">
    ${item}

```

3. 遍历 map 集合

```
<%
Map<String, Object> map = new HashMap<String, Object>();
map.put("key1", "value1");
map.put("key2", "value2");
map.put("key3", "value3");
request.setAttribute("map", map);
%>

<c:forEach items="${requestScope.map}" var="entry">
    ${entry.key} = ${entry.value}
</c:forEach>
```

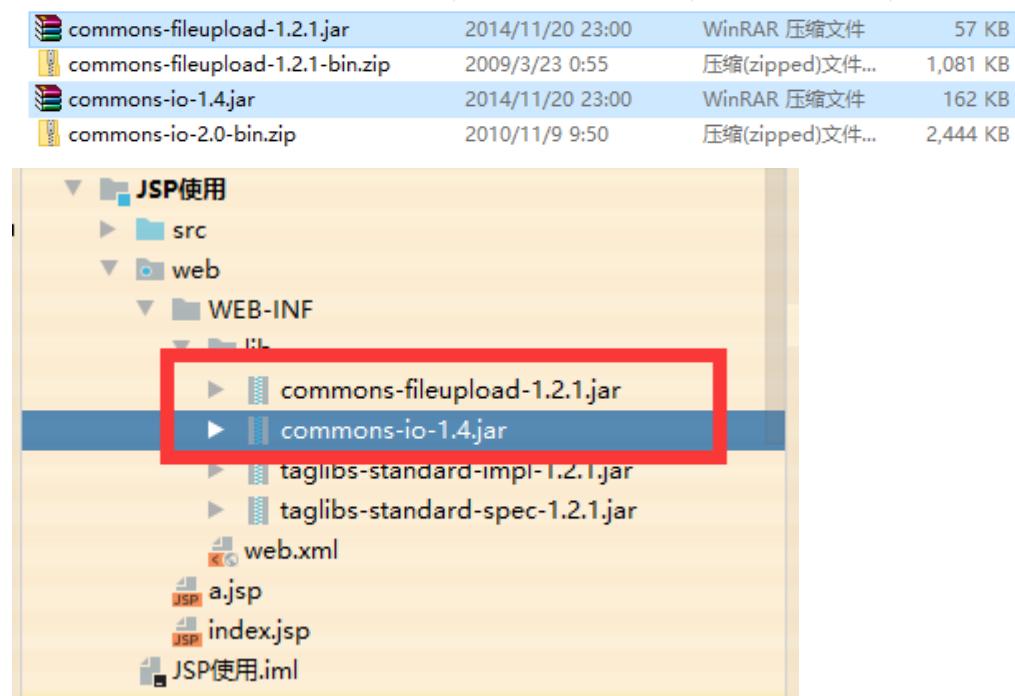
文件上传

要求:

- 1、要有一个 form 标签， method=post 请求
- 2、form 标签的 enctype 属性值必须为 multipart/form-data 值
- 3、在 form 标签中使用 input type=file 添加上传的文件

4、编写服务器代码（Servlet 程序）接收，处理上传的数据。
encType=multipart/form-data 表示提交的数据，以多段（每一个表单项一个数据段）的形式进行拼接，然后以二进制流的形式发送给服务器

IDEA 上传 jar 包



IDEA 上传 JAR 包中的方法

FileItem 类，表示每一个表单项。

ServletFileUpload 类，用于解析上传的数据。

Boolean

```
ServletFileUpload.isMultipartContent(HttpServletRequest  
request);
```

判断当前上传的数据格式是否是多段的格式。

public List parseRequest(HttpServletRequest request)

解析上传的数据

boolean FileItem.isFormField()

判断当前这个表单项，是否是普通的表单项。还是上传的文件类型。

true 表示普通类型的表单项 false 表示上传的文件类型

String FileItem.getFieldName()

获取表单项的 name 属性值

String FileItem.getString()

获取当前表单项的值。

String FileItem.getName();

获取上传的文件名

void FileItem.write(file);

将上传的文件写到 参数 file 所指向抽硬盘位置 。

小演示

JSP

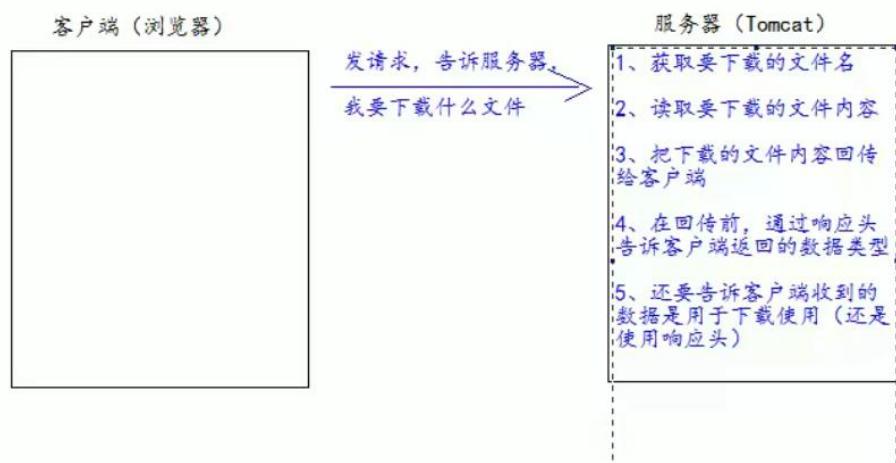
```
<body>
    <form action="http://localhost:8080/JSP_war_exploded/upFileServlet" enctype="multipart/form-data" method="post">
        账户:<input type="text"><br>
        头像:<input type="file"><br>
        <input type="submit" value="上传">
    </form>
</body>
```

Servlet

```
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    System.out.println("post");
    //1.判断是否传入的是多段数据,如果不是不进行解析
    if (ServletFileUpload.isMultipartContent(request)) {
        //2.创建一个Fileitem 流工厂
        FileItemFactory fileItemFactory = new DiskFileItemFactory();
        //3.创建一个解析上传文件的工具类
        ServletFileUpload servletFileUpload = new ServletFileUpload(fileItemFactory);
        //4.request 获取的数据解析放入 list 集合中
        List<FileItem> list = null;
        try {
            list = servletFileUpload.parseRequest(request);
            for (FileItem fileItem : list) {
                //5.判断是否为普通类型
                if (fileItem.isFormField()) {
                    //6.处理普通文件
                    //获取表单 name 值
                    System.out.println("表单 name 的值为:" + fileItem.getFieldName());
                    //获取 value 值
                    System.out.println("表单 value 值为:" + fileItem.getString("UTF-8"));
                } else {
                    //7.处理文件类型
                    //获取表单文件 name 值
                    System.out.println("表单文件 name 为:" + fileItem.getFieldName());
                    System.out.println("上传文件 name 为:" + fileItem.getName());
                    try {
                        fileItem.write(new File("e:\\\" + fileItem.getName()));
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

文件下载

服务器客户端流程



Servlet

```
@Override  
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException,  
IOException {  
    //设置下载到客户端的文件名  
    String filename = "b.jpg";  
    //声明 Context 对象读取文件  
    ServletContext context = getServletContext();  
    //1. 设置响应头告诉客户端返回的什么类型  
    resp.setContentType(context.getMimeType("/file/" + filename));  
    //2. 告诉客户端去下载,第一个是设置响应头,第二个为文件名,使用 utf-8 编码  
    resp.setHeader("Content-Disposition", "attachment; filename=" +  
        URLEncoder.encode(filename, "UTF-8"));  
    //3. 获取下载的内容  
    InputStream resourceAsStream = context.getResourceAsStream("/file/" + filename);  
    //4. 获取输出流  
    ServletOutputStream outputStream = resp.getOutputStream();  
    //5. 发送客户端让客户端下载文件  
    IOUtils.copy(resourceAsStream, outputStream);  
}
```

Cookie

Cookie 的创建

客户端和服务器的交互

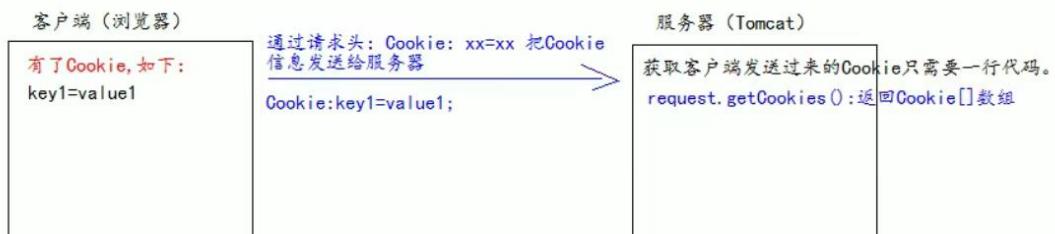


Servlet 程序

```
protected void addCookie(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
    //创建Cookie对象
    Cookie cookie = new Cookie("name", "key1");
    //发送给客户端
    resp.addCookie(cookie);
    //回写
    resp.getWriter().write("Cookie创建成功");
}
```

获取 Cookie 的值

客户端服务器交互



Servlet

使用到了自己写的 utils

```
//获取  
protected void getCookie(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  
    Cookie[] cookies = req.getCookies();  
    Cookie key1 = cookieUtils.getNameCookie(cookName: "key1", cookies);  
    System.out.println(key1.getValue());  
    resp.getWriter().write(key1.getValue());  
}
```

Utils

```
public static Cookie getNameCookie(String cookName,Cookie[] cookies){  
    if(cookName==null||cookies==null||cookies.length==0){  
        return null;  
    }  
    for (Cookie cookie : cookies) {  
        if(cookName.equals(cookie.getName())){  
            return cookie;  
        }  
    }  
    return null;  
}
```

修改 Cookie 的值

```
//修改  
protected void updateCookie(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  
    //方式一：创建一个新的cookie对象，构造器中key放想修改cookie中key的值,value为新值  
    //Cookie cookie = new Cookie("key1", "newKey1");  
    //resp.addCookie(cookie);  
    //方式二：通过客户端穿过来的cookie值获取cook对象,然后使用setValue设置新的value值  
    Cookie cookie = cookieUtils.getNameCookie(cookName: "key1", req.getCookies());  
    if(cookie!=null){  
        cookie.setValue("newKey1");  
    }  
    resp.addCookie(cookie);  
    resp.getWriter().write(s: "修改成功");  
}
```

Cookie 的生命周期

正数为存活多长时间 3600 一小时 秒为单位

负数为浏览器一关就全部删除

零为立即删除

Servlet

```
// 设置声明周期  
protected void CookieAge(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  
    Cookie cookie = new Cookie( name: "key", value: "key" );  
    cookie.setMaxAge(3600); //存活一个小时  
    resp.addCookie(cookie);  
}
```

创建带有 Path 的 Cookie

```
//设置带Path路径Cookie  
protected void testPath(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  
    Cookie cookie = new Cookie( name: "path1", value: "path1" );  
    // getContextPath() ==>>> 得到工程路径  
    cookie.setPath( req.getContextPath() + "/abc" ); // ==>>> /工程路径/abc  
    resp.addCookie(cookie);  
    resp.getWriter().write( s: "创建了一个带有 Path 路径的 Cookie" );  
}
```

Session 会话

什么是 Session 会话?

- 1、Session 就一个接口（ HttpSession ）。
- 2、Session 就是会话。它是用来维护一个客户端和服务器之间关联的一种技术。
- 3、每个客户端都有自己的一个 Session 会话。
- 4、Session 会话中，我们经常用来保存用户登录之后的信息

如何创建 Session 和获取(id 号

- 1、request.getSession() 方法:第一次是创建,以后每一次都是获取
- 2、isNew(); 判断到底是不是刚创建出来的（新的） true 为刚创建 false 为获取

3、getId() 得到 Session 的身份证号

每个会话都有一个身份证号。也就是 ID 值。而且这个 ID 是唯一的。

Session 存储数据

```
protected void setAttribute(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
    req.getSession().setAttribute("key1", "value1");
    resp.getWriter().write("已经往 Session 中保存了数据");
}
```

Session 获取数据

```
protected void getAttribute(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
    Object attribute = req.getSession().getAttribute("key1");
    resp.getWriter().write("从 Session 中获取出 key1 的数据是: " + attribute);
}
```

Session 生命周期控制

`public void setMaxInactiveInterval(int interval)` 设置 Session 的超时时间（以秒为单位），超过指定的时长，Session 就会被销毁。

值为正数的时候，设定 Session 的超时时长。

负数表示永不超时（极少使用）

`public int getMaxInactiveInterval()` 获取 Session 的超时时间

`public void invalidate()` 让当前 Session 会话马上超时无效

Session 时长

Session 默认的超时时间为 30 分钟。在底层有配置文件设置默认值

```
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
```

修改项目默认时长

如果想修改自己项目的时长,在 Session 中添加相同的配置

```
<session-config>
    <session-timeout>20</session-timeout>
</session-config>
```

修改个别的 Session 的时长

session.setMaxInactiveInterval(int interval)单独设置超时时长

设置 Session 马上超时

```
Session.invalidate()
```

Filter 过滤器

作用:

1、Filter 过滤器它是 JavaWeb 的三大组件之一。三大组件分

别是：Servlet 程序、Listener 监听器、Filter 过滤器

2、Filter 过滤器它是 JavaEE 的规范。也就是接口

3、Filter 过滤器它的作用是：拦截请求，过滤响应。

应用：

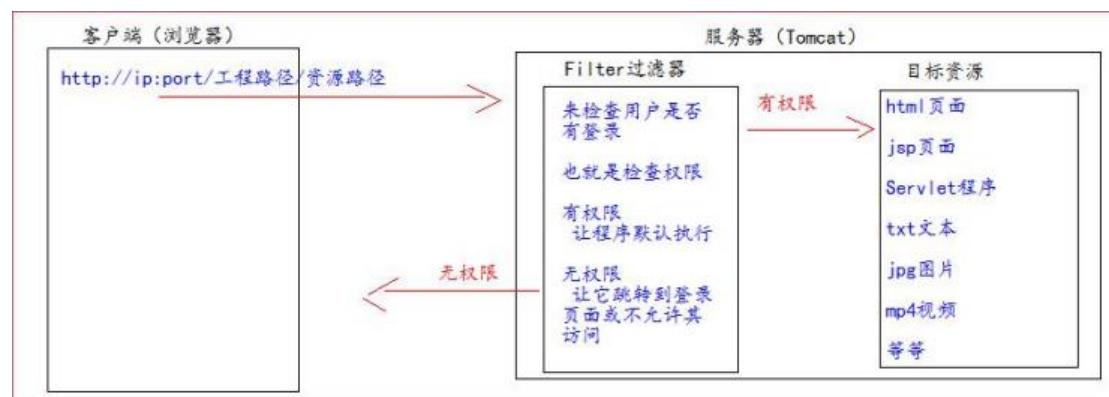
拦截请求常见的应用场景有：

1、权限检查

2、日记操作

3、事务管理等等

流程图



简单使用：

要求：

在你的 web 工程下，有一个 admin 目录。这个 admin 目录下的所有资源（html 页面、jpg 图片、jsp 文件、等等）都必须是用户登录之后才允许访问

根据之前我们学过内容。我们知道，用户登录之后都会把用户登录的信息保存到 **Session** 域中。所以要检查用户是否登录，可以判断 **Session** 中否包含有用户登录的信息即可!!

Filter 过滤器的使用步骤：

- 1、编写一个类去实现 Filter 接口
- 2、实现过滤方法 doFilter()
- 3、到 web.xml 中去配置 Filter

Filter 的代码：

```
public class AdminFilter implements Filter {  
    /**  
     * doFilter 方法，专门用于拦截请求。可以做权限检查  
     */  
    @Override  
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) throws IOException, ServletException {  
        HttpServletRequest httpServletRequest = (HttpServletRequest) servletRequest;  
  
        HttpSession session = httpServletRequest.getSession();  
        Object user = session.getAttribute("user");  
        // 如果等于null，说明还没有登录  
        if (user == null) {  
  
            servletRequest.getRequestDispatcher("/login.jsp").forward(servletRequest,servletResponse);  
            return;  
        } else {  
            // 让程序继续往下访问用户的目标资源  
            filterChain.doFilter(servletRequest,servletResponse);  
        }  
    }  
}
```

web.xml 中的配置：

```
<!--filter 标签用于配置一个Filter 过滤器-->
<filter>
    <!--给 filter 起一个别名-->
    <filter-name>AdminFilter</filter-name>
    <!--配置 filter 的全类名-->
    <filter-class>com.atguigu.filter.AdminFilter</filter-class>
</filter>
```

```
<!--filter-mapping 配置 Filter 过滤器的拦截路径-->
<filter-mapping>
    <!--filter-name 表示当前的拦截路径给哪个filter 使用-->
    <filter-name>AdminFilter</filter-name>
    <!--url-pattern 配置拦截路径
        / 表示请求地址为: http://ip:port/工程路径/ 映射到IDEA 的 web 目录
        /admin/* 表示请求地址为: http://ip:port/工程路径/admin/*
    -->
    <url-pattern>/admin/*</url-pattern>
</filter-mapping>
```

Filter 的生命周期

Filter 的生命周期包含几个方法

1、构造器方法

2、init 初始化方法

第 1, 2 步，在 web 工程启动的时候执行（Filter 已经创建）

3、doFilter 过滤方法

第 3 步，每次拦截到请求，就会执行

4、destroy 销毁

第 4 步，停止 web 工程的时候，就会执行（停止 web 工程，也会销毁 Filter 过滤

FilterConfig 类

FilterConfig 类见名知义，它是 Filter 过滤器的配置文件类。Tomcat 每次创建 Filter 的时候，也会同时创建一个 FilterConfig 类，这里包含了 Filter 配置文件的配置信息。

FilterConfig 类的作用是获取 filter 过滤器的配置内容

- 1、获取 Filter 的名称 filter-name 的内容
- 2、获取在 Filter 中配置的 init-param 初始化参数
- 3、获取 ServletContext

java 代码：

```
@Override  
public void init(FilterConfig filterConfig) throws ServletException {  
    System.out.println("2.Filter 的 init(FilterConfig filterConfig) 初始化");  
  
    // 1. 获取Filter 的名称 filter-name 的内容  
    System.out.println("filter-name 的值是: " + filterConfig.getFilterName());  
    // 2. 获取在web.xml 中配置的init-param 初始化参数  
    System.out.println("初始化参数 username 的值是: " + filterConfig.getInitParameter("username"));  
    System.out.println("初始化参数 url 的值是: " + filterConfig.getInitParameter("url"));  
    // 3. 获取ServletContext 对象  
    System.out.println(filterConfig.getServletContext());  
}
```

web.xml 配置：

```
<!--filter 标签用于配置一个Filter 过滤器-->
<filter>
    <!--给 filter 起一个别名-->
    <filter-name>AdminFilter</filter-name>
    <!--配置 filter 的全类名-->
```

```
<filter-class>com.atguigu.filter.AdminFilter</filter-class>

<init-param>
    <param-name>username</param-name>
    <param-value>root</param-value>
</init-param>

<init-param>
    <param-name>url</param-name>
    <param-value>jdbc:mysql://localhost3306/test</param-value>
</init-param>

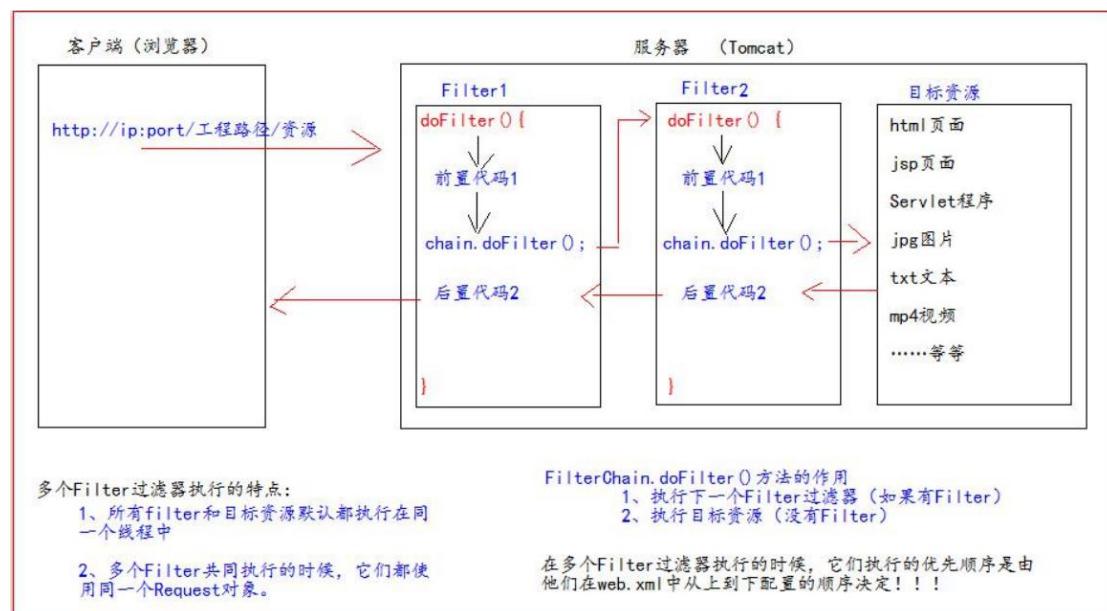
</filter>
```

FilterChain 过滤器链

Filter 过滤器

Chain 链，链条

FilterChain 就是过滤器链（多个过滤器如何一起工作）



说明:

一个文件绑定了多个过滤器,过滤器就会按照 web.xml 中配置的先后顺序来执行过滤器,当过滤器执行成功调到资源路径,失败更改程序

Filter 的拦截路

--精确匹配

<url-pattern>/target.jsp</url-pattern>

以上配置的路径, 表示请求地址必须为: http://ip:port/工程路径/target.jsp

--目录匹配

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

以上配置的路径, 表示请求地址必须为: http://ip:port/工程路径/admin/*

--后缀名匹配

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

以上配置的路径, 表示请求地址必须以.html 结尾才会拦截到

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

以上配置的路径，表示请求地址必须以`.do` 结尾才会拦截到

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

以上配置的路径，表示请求地址必须以`.action` 结尾才会拦截到

`Filter` 过滤器它只关心请求的地址是否匹配，不关心请求的资源是否存在!!!

JSON

什么是 JSON

`JSON` (`JavaScript Object Notation`) 是一种轻量级的数据交换格式。易于人阅读和编写。同时也易于机器解析和生成。`JSON` 采用完全独立于语言的文本格式，而且很多语言都提供了对 `json` 的支持（包括 `C, C++, C#, Java, JavaScript, Perl, Python` 等）。这样就使得 `JSON` 成为理想的数据交换格式。数据交换指的是客户端和服务器之间业务数据的传递格式。

JSON 在 JavaScript 中的使用

json 的定义<String, Object>

```
// json的定义
var jsonObj = {
    "key1":12,                                <!--int类型-->
    "key2":"abc",                               <!--字符串-->
    "key3":true,                               <!--布尔-->
    "key4":[11, "arr", false],                <!--数组-->
    "key5":{                                    <!--JSON类型-->
        "key5_1" : 551,
        "key5_2" : "key5_2_value"
    },
    "key6":[{                                   <!--数组+JSON类型-->
        "key6_1_1":6611,
        "key6_1_2":"key6_1_2_value"
    },{
        "key6_2_1":6621,                      <!--数组+JSON类型-->
        "key6_2_2":"key6_2_2_value"
    }]
};
```

JSON 的访问

json 本身是一个对象。

json 中的 key 我们可以理解为是对象中的一个属性。

json 中的 key 访问就跟访问对象的属性一样： json 对象.key

json 访问示例

```
alert(typeof(jsonObj)); // object json 就是一个对象
```

```
alert(jsonObj.key1); //12  
alert(jsonObj.key2); // abc  
alert(jsonObj.key3); // true  
alert(jsonObj.key4);//得到数组[11,"arr",false]  
// json 中数组值的遍历  
for(var i = 0; i < jsonObj.key4.length; i++) {  
    alert(jsonObj.key4[i]);  
}  
alert(jsonObj.key5.key5_1);//551  
alert(jsonObj.key5.key5_2);//key5_2_value  
alert( jsonObj.key6 );// 得到 json 数组  
//取出来每一个元素都是 json 对象  
var jsonItem = jsonObj.key6[0];  
alert( jsonItem.key6_1_1 ); //6611  
alert( jsonItem.key6_1_2 ); //key6_1_2_value
```

json 的两个常用方法

json 的存在有两种形式。

一种是：对象的形式存在，我们叫它 json 对象。

一种是：字符串的形式存在，我们叫它 json 字符串。

一般我们要操作 json 中的数据的时候，需要 json 对象的格式。

一般我们要在客户端和服务器之间进行数据交换的时候，使用 json 字符串

JSON.stringify() 把 json 对象转换成为 json 字符串

JSON.parse() 把 json 字符串转换成为 json 对象

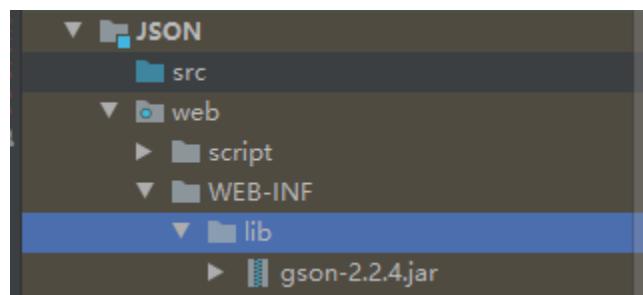
```
// 把json 对象转换成为 json 字符串
var jsonObjString = JSON.stringify(jsonObj); // 特别像 Java 中对象的toString
alert(jsonObjString)
// 把json 字符串。转换成为 json 对象
var jsonObj2 = JSON.parse(jsonObjString);
alert(jsonObj2.key1); // 12
alert(jsonObj2.key2); // abc
```

JSON 的转换

首先导入谷歌的 gson 包

 gson-2.2.4.jar 2014/12/8 1:27 WinRAR 压缩文件 186 KB

导入到 idea 并添加到库



JavaBean 和 JSON 的转换

```
public void test1(){
    Person person = new Person(1,"国哥好帅!");
    // 创建 Gson 对象实例
    Gson gson = new Gson();
    // toJson 方法可以把 java 对象转换成为 json 字符串
```

```
String personJsonString = gson.toJson(person);

System.out.println(personJsonString);

// fromJson 把 json 字符串转换回 Java 对象

// 第一个参数是 json 字符串, 第二个参数是转换回去的 Java 对象类型

Person person1 = gson.fromJson(personJsonString,
Person.class);

System.out.println(person1);

}
```

JSON 转换为 List 集合

测试类

```
@Test
public void test1(){
    List<Person> listPerson = new ArrayList<>();
    Person person1 = new Person( id: 1, name: "abc");
    Person person2 = new Person( id: 2, name: "abc");
    Gson g = new Gson();
    listPerson.add(person1);
    listPerson.add(person2);
    //json转换为字符串
    String s = g.toJson(listPerson);
    System.out.println(s);
    //字符串转换为json,第一个为转换的字符串,第二个为类型,但是不能直接放,需要实现jar包下的TypeToken接口实现类
    List<Person> list = g.fromJson(s, new importShixian().getType());
    System.out.println(list);
    Person person3 = list.get(0);
    System.out.println(person3);
}
```

实现类

```
package pojo;

import com.google.gson.reflect.TypeToken;

import java.util.List;

public class importShixian extends TypeToken<List<Person>> {
}
```

JSON 和 MAP 之间的互转

```
@Test
public void test3(){
    Map<Integer,Person> personMap = new HashMap<>();
    personMap.put(1, new Person( id: 1, name: "国哥好帅"));
    personMap.put(2, new Person( id: 2, name: "康师傅也好帅"));
    Gson gson = new Gson();
    // 把 map 集合转换成为 json 字符串
    String personMapJsonString = gson.toJson(personMap);
    System.out.println(personMapJsonString);
    // Map<Integer,Person> personMap2 = gson.fromJson(personMapJsonString, new PersonMapType().getType());
    Map<Integer,Person> personMap2 = gson.fromJson(personMapJsonString,
                                                    new TypeToken<HashMap<Integer,Person>>().getType());
    System.out.println(personMap2);
    Person p = personMap2.get(1);
    System.out.println(p);
}
```

AJAX

什么是 AJAX 请求

AJAX 即“*Asynchronous Javascript And XML*”（异步 JavaScript 和 XML），是指一种创建交互式网页应用的网页开发 技术。AJAX 是一种浏览器通过 js 异步发起请求，局部更新页面的技术。

AJAX 请求的局部更新，浏览器地址栏不会发生变化 局部更新不会舍弃原来页面的内容

原生 AJAX 示例:

JavaScript

```
<script type="text/javascript">
    // 在这里使用 JavaScript 语言发起 Ajax 请求，访问服务器 AjaxServlet 中 javaScriptAjax
    function ajaxRequest() {
        // 1、我们首先要创建 XMLHttpRequest
        var xmlhttprequest = new XMLHttpRequest();
        // 2、调用 open 方法设置请求参数,1.什么请求 2.请求地址 3.是否异步 异步true 同步false
        xmlhttprequest.open("GET", "http://localhost:8080/16_json_ajax_i18n/ajaxServlet?action=javaScriptAjax", true);
        // 5、在 send 方法前绑定 onreadystatechange 事件，处理请求完成后的操作。
        xmlhttprequest.onreadystatechange = function() {
            // 判断服务器是否就绪或是是否发生错误
            if (xmlhttprequest.readyState == 4 && xmlhttprequest.status == 200) {
                var jsonObj = JSON.parse(xmlhttprequest.responseText);
                // 把响应的数据显示在页面上
                document.getElementById("div01").innerHTML = "编号：" + jsonObj.id + "， 姓名：" +
                    jsonObj.name;
            }
        }
        // 4、调用 send 方法发送请求
        xmlhttprequest.send();
    }
</script>
```

Servlet

```
protected void javaScriptAjax(HttpServletRequest req,
HttpServletResponse resp) throws ServletException, IOException
{
    Person person = new Person(1, "嘿嘿");

    try {
        Thread.sleep(3000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    // json 格式的字符串
    Gson gson = new Gson();
```

```
String personJsonString = gson.toJson(person);
```

```
resp.getWriter().write(personJsonString);
```

```
}
```

jQery 的 AJAX

`$.ajax` 方法

`url` 表示请求的地址

`type` 表示请求的类型 GET 或 POST 请求

`data` 表示发送给服务器的数据

格式有两种：

一： `name=value&name=value`

二： `{key:value}`

`success` 请求成功， 响应的回调函数

`dataType` 响应的数据类型

常用的数据类型有：

`text` 表示纯文本

`xml` 表示 xml 数据

`json` 表示 json 对

```
$("#ajaxBtn").click(function(){
```

```
    $.ajax({
```

```
        url:"http://localhost:8080/16_json_ajax_i18n/aj
```

```
axServlet", //请求地址  
          // data:"action=jQueryAjax",  
          data:{action:"jQueryAjax"}, //请求参数  
          type:"GET", //请求方式  
          success:function (data) { //获取服务器发  
            来的数据  
              $("#msg").html("编号: " + data.id + " , 姓  
名: " + data.name);  
            },  
            dataType : "json" //接受服务器数据的  
类型,一般都是 json  
        );  
    );
```

\$get 和\$post

url 请求的 url 地址

data 发送的数据

callback 成功的回调函数

type 返回的数据类型

ajax--get 请求

```
$("#getBtn").click(function(){
```

```
$.get("http://localhost:8080/16_json_ajax_i18n/ajaxServlet","action=jQueryGet",function (data) {
    $("#msg").html(" get 编号: " + data.id + " , 姓名: " + data.name);
}, "json");
});
```

ajax--post 请求

```
$("#postBtn").click(function(){
    $.post("http://localhost:8080/16_json_ajax_i18n/ajaxServlet","action=jQueryPost",function (data)
{
    $("#msg").html(" post 编号: " + data.id + " , 姓名: " + data.name);
}
});
```

`$.getJSON` 方法 只 get 请求,只接受 JSON 数据

`url` 请求的 `url` 地址

`data` 发送给服务器的数据

`callback` 成功的回调函数

```
// ajax--getJSON 请求
$("#getJSONBtn").click(function(){
```

```
$.getJSON("http://localhost:8080/16_json_ajax_i18n/
ajaxServlet","action=jQueryGetJSON",function
(data) {
    $("#msg").html(" getJSON 编号: " + data.id + " , 姓
名: " + data.name);
});
});
```

表单序列化 serialize()

serialize()可以把表单中所有表单项的内容都获取到，并以 name=value&name=value 的形

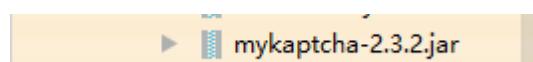
```
// ajax 请求
$("#submit").click(function(){
    // 把参数序列化
    $.getJSON("http://localhost:8080/16_json_ajax_i18n/
ajaxServlet","action=jQuerySerialize&" +
$("#form01").serialize(),function (data) {
    $("#msg").html(" Serialize 编号: " + data.id + " , 姓
名: " + data.name);
});
});
```

谷歌验证码 JAR 包使用

复制到 java 项目中



添加到库



在 web.xml 中添加

```
<servlet>
    <servlet-name>KaptchaServlet</servlet-name>
    <servlet-class>com.google.code.kaptcha.servlet.KaptchaServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>KaptchaServlet</servlet-name>
    <url-pattern>/kaptcha.jpg</url-pattern>
</servlet-mapping>
```

然后再 jsp 网页中添加 img 标签指向这个位置

```

```

Servlet

```
//获取谷歌jar包下生成的验证码
String rightCode = (String) request.getSession().getAttribute(KAPTCHA_SESSION_KEY);
```

通过 JSTL 标签库实现国际化

<%--1 使用标签设置 Locale 信息--%>

```
<fmt:setLocale value="" />
```

<%--2 使用标签设置 basename--%>

```
<fmt:setBundle basename="" />
```

<%--3 输出指定 key 的国际化信息--%>

```
<fmt:message key="" />
```

```
<%@ taglib prefix="fmt"
uri="http://java.sun.com/jsp/jstl/fmt" %> //第一步加入标签库
```

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

```
pageEncoding="UTF-8"%>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
```

```
Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

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

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

<meta http-equiv="Expires" content="0" />

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

<title>Insert title here</title>

</head>

<body>

<%--1 使用标签设置 Locale 信息--%>

<fmt:setLocale value="${param.locale}" />          //动态获取
语言

<%--2 使用标签设置 basename--%>

<fmt:setBundle basename="i18n"/>                  //获取配置文
件

<a href="i18n_fmt.jsp?locale=zh_CN">中文</a>|    //点击更
改语言发送参数

<a href="i18n_fmt.jsp?locale=en_US">english</a>

<center>

<h1><fmt:message key="regist" /></h1>
```

```
<table>

<form>

<tr>

<td><fmt:message key="username" /></td>

<td><input name="username" type="text" /></td>

</tr>

<tr>

<td><fmt:message key="password" /></td>      //根据 key 获得配置文件的 value

<td><input type="password" /></td>

</tr>

<tr>

<td><fmt:message key="sex" /></td>

<td>

<input type="radio" /><fmt:message key="boy" />

<input type="radio" /><fmt:message key="girl" />

</td>

</tr>

<tr>

<td><fmt:message key="email" /></td>

<td><input type="text" /></td>

</tr>
```

```
<tr>

<td colspan="2" align="center">
<input type="reset" value=<fmt:message key="reset" />>&nbsp;&nbsp;
<input type="submit" value=<fmt:message key="submit" />>
/></td>

</tr>
</form>
</table>
<br /><br /><br /><br />
</center>
</body>
</html>
```