# 框架

## 框架的认识

1. 简单

框架的使用非常简单。框架制定了一系列的规则，只要按照规则进行应用即可。

1. 困难

对于框架中的原理及思想是比较困难的。

## 学习方法

1. 阶段

* 初级：基本语法与面向对象
* 中级：使用面向对象思想
* 高级：理解业务逻辑

1. 学习方法

应用框架完成业务逻辑。

## 掌握程度

* 熟练运用框架
* 熟悉框架的底层原理

# 框架的由来

## Servlet开发

采用Model1模式进行开发。

Jsp

JavaBean

DB

Model1模式使用小脚本的方式进行开发。Model1模式下开发的项目运行效率较高。但开发效率较低。项目较大时不适用。

## MVC框架

MVC框架符合Model2模式。

Jsp（View）

Controller

Servlet

Model

JavaBean

DB

实现了代码分离。使得代码更加清晰，便于开发。使用Mvc模式提高了代码的服用性，提到了开发效率。使用Mvc也增加了代码量。

# 初始Struts2

## 框架

欢迎联想集团领导莅临我校参观

欢迎华为集团领导莅临我校参观

欢迎微软中国领导莅临我校参观

欢迎 领导莅临我校参观

框架就是一个模板，框架就是一个半成品。框架中包含一系列规则。

## 框架的好处

提高了开发效率(但可能会降低运行效率)。

## 常用MVC框架

常用MVC框架：Struts1、webwork、MVC、Struts2、Spring、Jsf、Nutz、Jersey

## Struts2的由来

Struts2 = Struts1 + webwork

Struts2是Apache公司的一个开源项目。

## 下载

https://struts.apache.org/

## 目录介绍

* apps ：示例代码
* docs ：文档
* lib ：jar包目录
* src ：源代码目录

## MVC作了什么

1. Servlet

* 将用户的请求映射到Java类
* 调用请求处理方法
* 处理响应数据
* 控制视图的显示
* 处理用户提交的数据

1. MVC

* 渲染视图
* 控制视图跳转
* 将用户请求映射到Java类
* 处理用户提交的数据

# 项目搭建

## 步骤

1. 第一步：创建Web项目
2. 第二步：导入相关jar包

|  |
| --- |
|  |

1. 第三步：配置核心过滤器

web app libraris ---> struts-core ---> org.apache.struts2.dispatcher

.ng.filter ---> StrutsPrepareAndExecuteFilter(Struts2的核心过滤器)

|  |
| --- |
| <!-- 过滤器 -->  <filter>  <filter-name>StrutsPrepareAndExecuteFilter</filter-name>  <filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>StrutsPrepareAndExecuteFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |

1. 第四步：添加Struts2的配置文件

src ---> struts.xml --->复制struts-default.xml内容(web app libraris >

> struts-core >> META-INF >> struts-default.xml) --->粘贴到struts2 配置文件 >> 去掉注释

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!DOCTYPE struts PUBLIC  "-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"  "http://struts.apache.org/dtds/struts-2.3.dtd">  <struts>  </struts> |

1. 第五步：创建Action类

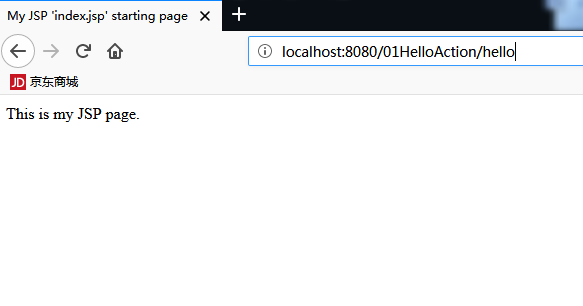
|  |
| --- |
| **public** **class** HelloAction {  /\*\*  \* 请求处理方法  \* 相当于Servlet中的Service  \* **@return**  \*/  **public** String execute(){  System.*out*.println("execute is do......");    **return** "success";  }  } |

1. 第六步：配置Action类

修改struts.xml中的内容。

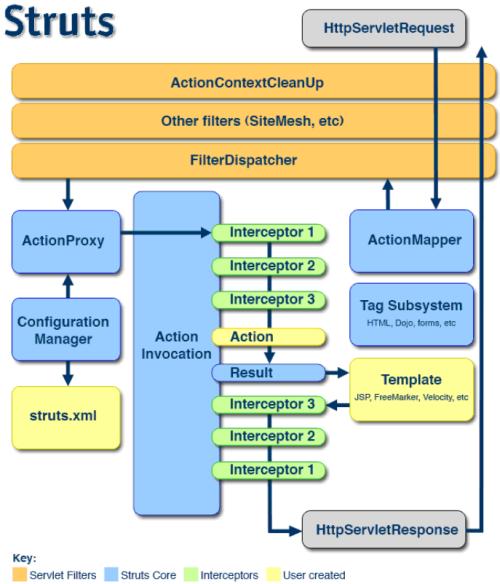
|  |
| --- |
| <struts>  <!--  name ：包名  extends ：表示继承于struts-default  -->  <package name=*"hello"* extends=*"struts-default"*>  <!--  name ：Action的名称，相当于servlet的映射名  class ：Action的完整名称(包名 + 类名)  -->  <action name=*"hello"* class=*"cn.hl.action.HelloAction"*>  <!--  控制跳转结果  -->  <result>index.jsp</result>  </action>  </package>  </struts> |

1. 第七步：布署项目
2. 第八步：访问action



# Struts2执行原理

## 原理图



## MVC



## 执行过程(\*\*\*\*\*)



1. 客户端浏览器发出请求时，被Tomcat服务器所接收。Tomcat容器将用户的请求封装为HttpServletRequest对象
2. 请求将通过一系列过滤器，当最后可以通过核心过滤器(StrutsPrepareAndExecuteFilter或FilterDispatcher（早期版本中使用）)时，将调用ActionMapper获取Action的映射信息(未得到Action对象)。
3. 请求继续向下流转，流转到ActionProxy。ActionProxy将通过ConfigManager类获取对应的Action类和相关方法。并创建一个Action代理类。
4. 代理类对象经过一系列拦截器栈(多个拦截器)，通过这些拦截器可以实现类型转换、数据绑定、国际化等操作
5. 调用Action方法处理客户的请求，请求处理结束后会形成一个Result。引擎将根据Result结果调用相应的模板进行渲染处理
6. 渲染结果将再次经过一系列拦截器对结果进行返回，返回后会形成一个HttpServletResponse对象
7. Tomcat会将Response对象的结果发送回客户端

## Struts2的线程安全性

### 什么是线程安全性

在多线程并发访问的情况下，如果一个对象中的变量的值不会随访问的线程而变化则是线程安全的。反之则称为非线程安全的。

### Servlet是线程安全的吗？

|  |
| --- |
| **[非线程安全的]**  **public** **class** HiServlet **extends** HttpServlet {  **int** i = 20;    @Override  **protected** **void** service(HttpServletRequest req, HttpServletResponse resp){  System.*out*.println("i = " + i);  i++;  }  } |
| **[线程安全的]**  **public** **class** HiServlet **extends** HttpServlet {  //int i = 20;    @Override  **protected** **void** service(HttpServletRequest req, HttpServletResponse resp){  **int** i = 20;    System.*out*.println("i = " + i);  i++;  }  } |

Servlet是使用单例模式进行实现的。请求过程中只会创建一个Servlet对象。所以Servlet是否是线程安全的与代码有关。

Struts1也是使用单例模式实现的。

### Struts2是线程安全的吗？

|  |
| --- |
| **public** **class** HiAction {  **private** **int** i = 20;  **public** HiAction() {  System.*out*.println("HiAction对象被创建了......");  }    **public** String execute(){  System.*out*.println("i = " + i);    i++;  **return** "success";  }  } |

Struts2中的Action是通过多例模式进行实现的，所以每次调用都会创建一个新的对象。所以它是线程安全的。

# 配置文件

## 配置文件加载顺序

struts-default.xml --> struts-plugin.xml --> struts.xml

## 常用配置

### 方式一

通过struts.xml配置文件直接修改常见配置

1. 配置编码格式

* 属性名

struts.i18n.encoding

* 取值

任意有效的字符编码

1. 配置后缀名

* 属性名

struts.action.extension

* 取值

任意有效的自定义名称

* 说明
* 设置默认值的同时，需要设置核心过滤器的映射
* ，，表示访问时可以省略后缀名

1. 设置错误提示模式

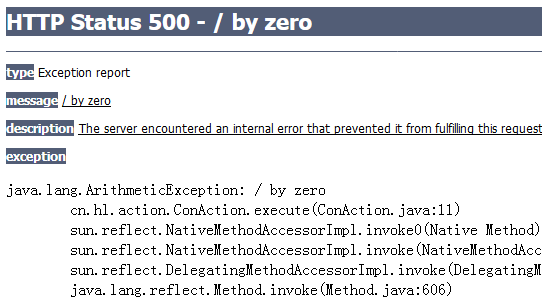
* 属性名

struts.devMode

* 值
* true ：开发模式，启用错误报告模式

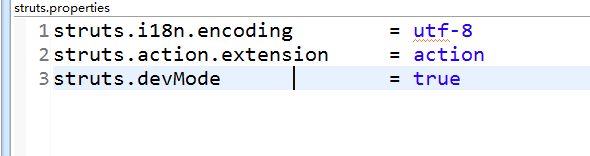


* false ：运行模式，不启用错误报告



### 方式二

除直接在struts.xml中修改配置外，也可以通过自定义struts.propertiesp修改常见配置。



## 团队协作

### 问题

团队开发环境下，当多人需要同时配置Action时，总会有人需要等待。造成了工作效率的下降。

### 解决

Struts2的配置文件允许将多个不同Action分别配置到多个配置文件中，然后通过include配置项引入其他配置文件。

### 示例

|  |
| --- |
| 【struts.xml】  <struts>  <!--  constant配置  配置编码格式（struts.i18n.encoding）  web app libraries >> struts2-core >> org.apache.struts >> default.properties  设置后缀名（struts.action.extension）  设置错误提示模式（struts.devMode）  -->  <!-- 编码格式设置 -->  <constant name=*""* value=*"UTF8"*></constant>  <!-- 设置后缀名 -->  <constant name=*"struts.action.extension"* value=*"terry,action,,"*></constant>  <!-- 设置错误提示模式 （true：开发模式；false：运行模式）-->  <constant name=*"struts.devMode"* value=*"false"*></constant>    <!--  引入其他配置文件  Struts框架允许将Action的配置定义其他文件中，然后引入其配置文件  在struts.xml配置文件中只定义公共的常量、变量、设置及文件的引入  -->  <include file=*"cn/conf/struts/Conf.xml"*></include>  </struts> |
| 【conf.xml】  <struts>  <package name=*"action"* extends=*"struts-default"*>  <action name=*"con"* class=*"cn.hl.action.ConAction"*>  <result>index.jsp</result>  </action>  </package>  </struts> |

### 说明

struts.xml配置文件只定义公共配置、常量、变量及文件引入

## Action的配置

1. 概述

Java是通过包管理类的，Struts2也是通过包来管理Action的。

1. 包(package)

* 示例

|  |
| --- |
| <!--  Java是通过包管理类的，Struts2也是通过包来管理Action的  通过包可以避免Action命名冲突的问题。    name ：包的名称  extends ：表示继承关系。一般继承于struts-default（顶级包，struts-default.xml中）.  namespace ：命名空间，默认值为"/"。    注意：设置命名空间后，访问action时需要附加命名空间作为前缀。如： conf/con.action  -->  <package name=*"action"* extends=*"struts-default"* namespace=*"/conf"*>  <action name=*"con"* class=*"cn.hl.action.ConAction"*>  <result>index.jsp</result>  </action>  </package> |

1. Action

配置请求处理方法。

|  |
| --- |
| <package name=*"action"* extends=*"struts-default"* namespace=*"/conf"*>  <!--  name ：配置Action的请求处理方法的名称，调用时通过该属性值进行调用  class ：配置请求处理方法所在类全名(包名 + 类名)  method ：配置请求处理方法的名称。默认值为execute  -->  <action name=*"con"* class=*"cn.hl.action.ConAction"*>  <result>index.jsp</result>  </action>    <action name=*"add"* class=*"cn.hl.action.ConAction"* method=*"add"*>  <result>index.jsp</result>  </action>  <action name=*"update"* class=*"cn.hl.action.ConAction"* method=*"update"*>  <result>index.jsp</result>  </action>  </package> |

1. result

配置请求处理的结果。

|  |
| --- |
| <package name=*"action"* extends=*"struts-default"* namespace=*"/conf"*>  <!--  name ：配置Action的请求处理方法的名称，调用时通过该属性值进行调用  class ：配置请求处理方法所在类全名(包名 + 类名)  method ：配置请求处理方法的名称。默认值为execute  -->  <action name=*"con"* class=*"cn.hl.action.ConAction"*>  <!--  跳转控制  name ：请求处理结果的名称，其值与对应请求处理方法中return的值相同  type ：配置结果的处理方式  chain ：表示请求链。用于从一个Action跳转到另一个Action（请求链）。如：ActionA -》ActionB  dispacher ：默认值，表示请求转发，相当于Servlet中的请求转发。一般用于将结果发送给jsp页面进行渲染时使用  redirect ：重定向，可以访问外部资源、jsp页面和Action。访问Action时需要加后缀名  redirectAction：跳转Action，访问过程中可以使用后缀名，也可以省略后缀名  -->  <result name=*"success"* type=*"chain"*>index.jsp</result>  </action>  </package> |

## 其他配置

|  |
| --- |
| <!-- 定义公共的结果集 -->  <global-results>  <result name=*"login"*>/login.jsp</result>  </global-results> |

# Action的实现方式

## Pojo方式

### 概述

Pojo(Plain Ordinary Java Object)称为简单Java类,其实就是一个JavaBean.

### 示例

|  |
| --- |
| /\*\*  \* Pojo类方式实现Action  \*  \* 优点：  \* 没有继承或实现任何类或接口，类不具有侵入性。便于代码的测试和迁移。  \* 缺点：  \* 没有约束，容易出现错误。  \* 不适用于新手使用  \* **@author** Terry  \*  \*/  **public** **class** PojoAction {  **public** String execute(){  System.*out*.println("Pojo.execute() is do......");    **return** "success";  }  } |
| <package name=*"pojo"* extends=*"struts-default"*>  <action name=*"pojo"* class=*"cn.hl.action.PojoAction"* >  <result>/index.jsp</result>  </action>  </package> |

### 优点

简单、方便。不具有侵入性（不具备侵入性：没有继承或实现其他类或接口），便于代码的迁移。

### 缺点

没有约束，容易出现错误，不适合新手使用。

## 实现接口

### 概述

通过实现Action接口的方式实现Action类。

### 示例

|  |
| --- |
| /\*\*  \* 实现方式  \* 创建类并实现Action接口  \*  \* 优点：  \* 具有一定的约束性，减少了错误的发生  \* 缺点：  \* 通过实现的方式实现的Action，使类具有了侵入性，降低了代码的迁移性  \*  \* **@author** Terry  \*  \*/  **public** **class** InheritAction **implements** Action{  @Override  **public** String execute() **throws** Exception {  System.*out*.println("InheritAction.execute() is do......");  **return** **null**;  }  } |

### 优点

具有一定的约束性，减少了错误的发生

### 缺点

通过实现的方式实现的Action，使类具有了侵入性，降低了代码的迁移性

## 继承类

### 概述

通过继承ActionSupport类的方式实现Action。

### 示例

|  |
| --- |
| /\*\*  \* 继承方式实现Action  \* 通过继承ActionSupport类进行实现  \*  \* 优点:  \* 可以直接继承父类的非私有成员,达到最少代码实现Action  \* 缺点:  \* 具有侵入性,同时会继承父类所有非私有成员(这些成员不一定都被使用)  \* 官方推荐方式  \* **@author** Terry  \*  \*/  **public** **class** ExtendsAction **extends** ActionSupport{  } |

### 优点

可以直接继承父类的非私有成员,达到最少代码实现Action

### 缺点

具有侵入性,同时会继承父类所有非私有成员(这些成员不一定都被使用)

## Action接口的成员

* success :表示Action执行成功，需要将结果展示给用户
* none :表示Action执行成功，但不需要象用户展示任何内容。如：下载功能
* error :表示Action执行失败，需要向用户展示一个错误视图
* input :表示需要跳转到输入表单时使用
* login :表示需要用户执行登录时使用

# 属性驱动与模型驱动

## 概述

通过属性(模型)驱动，可以使得Action获取到用户表单所提交的数据。

## 属性驱动

### 概述

所谓属性驱动是指使用类的属性（成员字段）来接收用户提交的数据。

### 实现步骤

* 第一步：创建Action类
* 第二步：添加对应的属性(成员字段)并生成get/set访问器
* 第三步：编写请求处理方法
* 第四步：配置Action

### 示例1：基本类型作为属性

|  |
| --- |
| [View]  <form action=*"user/add.action"* method=*"post"*>  <table>  <tr>  <th>姓名：</th>  <td>  <input type=*"text"* name=*"userName"* />  </td>  </tr>  <tr>  <th>帐号：</th>  <td>  <input type=*"text"* name=*"account"* />  </td>  </tr>  <tr>  <th>密码：</th>  <td>  <input type=*"text"* name=*"pwd"* />  </td>  </tr>  <tr>  <td>  <input type=*"submit"* value=*"保存"* />  </td>  </tr>  </table>  </form> |
| [Action类]  /\*\*  \* 属性驱动方式获取表单数据  \*  \* 将表单提交的数据以Action属性（成员字段）的形式进接收  \* **@author** Terry  \*  \*/  **public** **class** UserAction **implements** Action{  **private** String userName;  **private** String account;  **private** String pwd;    /\*\*  \* 请求处理方法  \*/  @Override  **public** String execute(){  System.*out*.println("userName = " + userName);  System.*out*.println("account = " + account);  System.*out*.println("pwd = " + pwd);  System.*out*.println("==========================================\n");  **return** "success";  }    //get/set访问器  } |
| [配置文件]  <package name=*"UserAction"* extends=*"struts-default"* namespace=*"/user"*>  <action name=*"add"* class=*"cn.hl.action.UserAction"*>  <result>/index.jsp</result>  </action>  </package> |
| [执行结果] |

### 示例2：引用类型作为属性

|  |
| --- |
| [View]  <form action=*"user/add.action"* method=*"post"*>  <table>  <tr>  <th>姓名：</th>  <td>  <input type=*"text"* name=*"user.userName"* />  </td>  </tr>  <tr>  <th>帐号：</th>  <td>  <input type=*"text"* name=*" user.account"* />  </td>  </tr>  <tr>  <th>密码：</th>  <td>  <input type=*"text"* name=*" user.pwd"* />  </td>  </tr>  <tr>  <td>  <input type=*"submit"* value=*"保存"* />  </td>  </tr>  </table>  </form> |
| [Javabean]  **public** **class** User {  **private** String userName;  **private** String account;  **private** String pwd;    **public** User() {  // **TODO** Auto-generated constructor stub  }  } |
| [Action类]  **public** **class** UserAction2 **implements** Action{  **private** User user = **new** User();    /\*\*  \* 请求处理方法  \*/  @Override  **public** String execute(){  System.*out*.println(user);  System.*out*.println("==========================================\n");  **return** "success";  }  **public** User getUser() {  **return** user;  }  **public** **void** setUser(User user) {  **this**.user = user;  }    } |
| [配置文件]  <action name=*"add2"* class=*"cn.hl.action.UserAction2"*>  <result>/update2.jsp</result>  </action> |
| [执行结果] |

### 注意事项

* 基本类型作为属性时,属性名（成员字段名）必须和表单中name值一致
* 属性必须提供get/set访问器。否则将无法获取到数据值
* 引用类型作为成员属性时，不需要进行实例化

## 模型驱动

### 概述

模型驱动又称为ModelDriven。在实现过程中，Action需要去实现ModelDriven接口。

### 实现步骤

* 第一步：创建类并实现ModelDriven接口
* 第二步：定义属性（成员字段）并编写get/set访问器
* 第三步：编写请求处理方法
* 第四步：配置Action

### 实例

|  |
| --- |
| [Action]  /\*\*  \* 模型驱动  \* 必须实现ModelDriven接口  \* 成员字段必须进行实例化  \* 必须完成get访问器  \* **@author** Terry  \*  \*/  **public** **class** UserAction3 **implements** ModelDriven<User>{  //必须进行实例化  **private** User user = **new** User();    //只有get（没有set）  @Override  **public** User getModel() {  **return** user;  }    **public** String execute(){  System.*out*.println(user);  **return** "success";  }  } |
| [配置Action]  <action name=*"add3"* class=*"cn.hl.action.UserAction3"*>  <result>/update3.jsp</result>  </action> |
| [Jsp]  <form action=*"user/add3.action"* method=*"post"*>  <table>  <caption>  用户注册(实体类方式--模型驱动)  </caption>  <tr>  <th>姓名：</th>  <td>  <input type=*"text"* name=*"userName"* />  </td>  </tr>  <tr>  <th>帐号：</th>  <td>  <input type=*"text"* name=*"account"* />  </td>  </tr>  <tr>  <th>密码：</th>  <td>  <input type=*"text"* name=*"pwd"* />  </td>  </tr>  <tr>  <td>  <input type=*"submit"* value=*"保存"* />  </td>  </tr>  </table>  </form> |

### 注意事项

* 成员字段必须进行实例化
* Jsp使用字段的属性时，不需要加字段名（前缀）
* 不需要定义set访问器

## 属性驱动和模型驱动[问]

* 属性驱动不需要继承或实现;模型驱动需要实现ModelDriven接口
* 属性驱动需要为成员字段提供get/set访问器；模型驱动只需要提供get访问器
* 属性驱动下，Action中的属性不需要进行初始化；模型驱动下Action中的属性必须进行初始化
* 使用属性驱动时，Jsp中必须通过对象名对其中成员进行调用；模型驱动时，Jsp中可以直接对对象中的成员进行访问

# Servlet API

## API

API（Application Programming Interface,应用程序编程接口）是一些预先定义的函数。直白的说API就是别人定义好的一组类和方法。

## 为什么要ServletAPI

Struts中无法直接使用Servlet中常用对象，如：cookie、session等。通过ServletAPI可以获取到这些对象。

## 解耦

### 概述

解耦就是尽量降低代码的耦合度。

### 方式一

|  |
| --- |
| /\*\*  \* 解耦方式一：  \* 通过ActionContext对象的getContext()获取Servlet对象  \* **@return**  \*/  **public** String api1(){  //1、设置Session  //Session中的数据是以键值对的形式存在的  Map<String,Object> session = ActionContext.*getContext*().getSession();  //将键值对存储到Map集合中  session.put("user", "Mike");  session.put("age", 20);  //将Session添加到ActionContext中  ActionContext.*getContext*().setSession(session);    //2、Application  Map<String,Object> application = ActionContext.*getContext*().getApplication();  //设置Application的值  application.put("count", "100");  ActionContext.*getContext*().setApplication(application);      //3、获取parameter(URL地址中的请求参数)  //http://localhost:8080/03ServletAPI/api1.action?name=Mike&age=20&gender=true  Map<String,Object> parameters = ActionContext.*getContext*().getParameters();  //获取Key的集合  Set<String> keys = parameters.keySet();  //获取set的迭代器  Iterator<String> iterator = keys.iterator();  //遍历迭代器  **while**(iterator.hasNext()){  //获取key的值  String key = iterator.next();  System.*out*.println(key + " = " + parameters.get(key));  }    //4、request  Map<String,Object> request = (Map<String,Object>)ActionContext.*getContext*().get("request");    **return** "success";  } |

### 方式二(推荐使用)

|  |
| --- |
| /\*\*  \* 解耦方式二：  \* StrutsStatics中定义了常用对象的字符串（字符串内容是类的完整名称）  \* **@return**  \* **@throws** IOException  \*/  **public** String api2() **throws** IOException{  //1、获取Request对象  HttpServletRequest req = (HttpServletRequest)ActionContext.*getContext*().get(StrutsStatics.*HTTP\_REQUEST*);    //通过Request对象获取Session对象  HttpSession session = req.getSession();    //通过Request对象获取application对象  ServletContext application = req.getServletContext();    //通过Reqeust对象获取请求参数  String name = req.getParameter("name");      //2、获取Response对象  HttpServletResponse resp = (HttpServletResponse)ActionContext.*getContext*().get(StrutsStatics.*HTTP\_RESPONSE*);  resp.getWriter().print("abvc");    **return** "success";  } |

## 耦合

### 概述

耦合是指模块和模块之间的依赖关系，如：A调用了B中的方法就称为A依赖B。同时，他们之间也就建立了耦合关系。

代码中模块间的依赖关系越多，模块的独立性就越差。所以一般设计时要求：高内聚，低耦合

### 方式一

|  |
| --- |
| /\*\*  \* 耦合的方式  \* 需要实现ServletRequestAware(或ServletContextAware)  \* 步骤：  \* 1、创建类并实现于ServletRequestWare接口  \* 2、创建私有成员对象req  \* 3、实现接口方法  \* 4、创建请求处理方法  \*  \* **@author** Terry  \*  \*/  **public** **class** ApiAction2 **implements** ServletRequestAware{  //HttpServletRequest类型的成员对象  **private** HttpServletRequest req;    //为成员对象赋值  @Override  **public** **void** setServletRequest(HttpServletRequest request) {  **this**.req = request;  }  //请求处理方法  **public** String api3(){  //1、获取Session  HttpSession session = req.getSession();    //2、获取Cookie  Cookie[] cookies = req.getCookies();    //3、获取Application  ServletContext application = req.getServletContext();    //4、获取请求参数  String name = req.getParameter("name");    **return** "success";  }  } |

## 解耦和耦合

* 解耦：便于测试
* 耦合：代码简单

# 验证框架

## 概述

Struts2提供了一套验证框架。如果需要使用Struts2的验证框架，Action必须继承于ActionSupport。

## 手动验证

1. 步骤

* 创建Action并继承ActionSupport
* 重写父类的验证方法并编写验证代码
* 配置Action
* 应用验证

1. 示例

|  |
| --- |
| 【Jsp页面】  <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"utf-8"*%>  <%@ taglib prefix=*"s"* uri=*"/struts-tags"* %>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <body>  <form action=*"user/reg.action"* method=*"post"*>  <!--  如果需要使用Struts2的验证框架。  必须使用对应的Struts2标签才能对错误消息进行显示  -->  <s:actionerror/>  <table>  <tr>  <th>姓名：</th>  <td><input type=*"text"* name=*"userName"* /></td>  </tr>  <tr>  <th>账号：</th>  <td><input type=*"text"* name=*"account"* /></td>  </tr>  <tr>  <th>密码：</th>  <td><input type=*"text"* name=*"pwd"* /></td>  </tr>  <tr>  <td colspan=*"2"*>  <input type=*"submit"* value=*"提交"* />  </td>  </tr>  </table>  </form>  </body>  </html> |
| 【Action】  /\*\*  \* Struts2验证框架  \* 必须继承于ActionSupport并重写Validate()方法  \* **@author** Terry  \*  \*/  **public** **class** UserAction **extends** ActionSupport{  **private** String userName;  **private** String account;  **private** String pwd;    /\*\*  \* 输入验证  \* 定义验证代码  \*/  @Override  **public** **void** validate() {  //编写数据验证代码  //用户名：不能为空且长度为6~20  **if**(userName == **null** || userName.length()<6 || userName.length() >20)  **this**.addActionError("用户名不合法");    //账号：不能为空且长度为6~20  **if**(account == **null** || account.length()<6 || account.length()>20)  **this**.addActionError("账号名不合法");    //密码：6个长度  **if**(pwd.length()!= 6)  **this**.addActionError("密码应为6个长度内容");    System.*out*.println("validate is do......");  }  /\*\*  \* 请求处理方法  \* **@return**  \*/  **public** String reg(){  System.*out*.println("userName = " + userName);  System.*out*.println("account = " + account);  System.*out*.println("pwd = " + pwd);    **return** "success";  }  } |
| 【配置文件】  <package name=*"UserAction"* extends=*"struts-default"* namespace=*"/user"*>  <action name=*"reg"* class=*"cn.hl.action.UserAction"* method=*"reg"*>  <result>/index.jsp</result>  <!--  未通过验证应该跳转回reg.jsp页面  跳转时name必须设置为input  -->  <result name=*"input"*>/reg.jsp</result>  </action>  </package> |

1. 执行流程

validate

reg

View视图

Y

Y

N

1. 扩展

validate()方法在Action中是公用的验证方法，当出现不同验证内容时，Struts的验证框架支持为方法定义对应的验证代码。

1. 实现

|  |
| --- |
| /\*\*  \* 为reg()方法定义的验证代码  \* 命名方式：validate+方法名  \*/  **public** **void** validateReg(){  //编写数据验证代码  // 用户名：不能为空且长度为6~20  **if** (userName == **null** || userName.length() < 6 || userName.length() > 20)  **this**.addActionError("用户名不合法");  // 账号：不能为空且长度为6~20  **if** (account == **null** || account.length() < 6 || account.length() > 20)  **this**.addActionError("账号名不合法");  // 密码：6个长度  **if** (pwd.length() != 6)  **this**.addActionError("密码应为6个长度内容");  } |

1. 执行流程

validateXxxx()

validate ()

xxxx()

View视图

Y

Y

Y

N

1. 注意事项

* 使用验证框架必须继承ActionSupport类
* result的name默认是input
* 如果需要显示错误消息，需要使用Struts标签

## 自动验证

1. 概述

Struts2框架同时支持自动验证。所谓自动验证是指通过编写验证配置文件来实现验证功能。

1. 步骤

* 创建Action类并继承ActionSupport
* 编写验证文件
* 编写请求处理方法
* 应用验证

1. 示例

|  |
| --- |
| 【Jsp页面】  <form action=*"emp/add"* method=*"post"*>  <!--  如果需要使用Struts2的验证框架。  必须使用对应的Struts2标签才能对错误消息进行显示  -->  <table>  <tr>  <th>姓名：</th>  <td>  <input type=*"text"* name=*"userName"* />  <s:fielderror fieldName=*"userName"*></s:fielderror>  </td>  </tr>  <tr>  <th>账号：</th>  <td><input type=*"text"* name=*"account"* /></td>  </tr>  <tr>  <th>密码：</th>  <td><input type=*"text"* name=*"pwd"* /></td>  </tr>  <tr>  <th>年龄：</th>  <td>  <input type=*"text"* name=*"age"* />  <s:fielderror fieldName=*"age"*></s:fielderror>  </td>  </tr>  <tr>  <td colspan=*"2"*>  <input type=*"submit"* value=*"提交"* />  </td>  </tr>  </table>  </form> |
| 【Action】  **public** **class** EpmAction **extends** ActionSupport{  **private** String userName;  **private** String account;  **private** String pwd;  **private** **int** age;    /\*\*  \* 请求处理方法  \* **@return**  \*/  **public** String add(){  System.*out*.println("userName = " + userName);  System.*out*.println("account = " + account);  System.*out*.println("pwd = " + pwd);  System.*out*.println("age = " + age);    **return** *SUCCESS*;  }  } |
| 【配置Action】  <package name=*"EmpAction"* extends =*"struts-default"* namespace=*"/emp"*>  <action name=*"add"* class=*"cn.hl.action.EpmAction"* method=*"add"*>  <result>/index.jsp</result>  <result name=*"input"*>/add.jsp</result>  </action>  </package> |
| 【验证文件】  <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!--  1、帮助文档  2、web app libraries 》》 xwork-core-xxx.jar 》》 xwork-validator-1.0.x.dtd  -->  <!DOCTYPE validators PUBLIC  "-//Apache Struts//XWork Validator 1.0.2//EN"  "http://struts.apache.org/dtds/xwork-validator-1.0.2.dtd">    <validators>  <field name=*"userName"*>  <!--  设置验证器的类型  web app libraries 》》 xwork-core-xxx.jar 》》 com.opensymphony.xwork2.validator.validators 》》 default.xml  -->  <field-validator type=*"requiredstring"*>  <param name=*"trim"*>true</param>  <message>用户名不能为空</message>  </field-validator>  </field>    <field name=*"age"*>  <field-validator type=*"int"*>  <param name=*"min"*>0</param>  <param name=*"max"*>180</param>  <message>年龄信息不合法</message>  </field-validator>  </field>  </validators> |

1. 注意事项

* 验证配置文件必须和Action在同一个包下
* 文件命名格式为：Action文件名 + validation.xml
* 消息必须通过对应的Struts标签进行显示

## 说明

Struts2的验证框架是服务器端验证框架，一般基本不被使用。除非是一些安全性要求较高的项目。

# ActionContext

## 概述

Struts1中,对象经容器封装后直接传递给Action。

请求

容器

request

response

Action

Struts1中数据的访问无法脱离Action及容器，所以导致耦合度过高。Struts2中为了解耦创建了ActionContext对象。

## 什么是ActionContext

ActionContext实际就是一个Map集合。ActionContex中存储了Action的相关数据。

请求

request

response

ActionContext

Action

每次请求会创建一个Action对象。同时也会创建一个ActionContext对象。

ActionContext是线程安全的。ActionContext通过ThreadLocal管理数据。

## ActionContext的作用

ActionContext中存储了Application、Session、Request、Local等数据。

## ActionContext的获取方式

ActionContext提供了一个名为getContext()的静态方法，通过方法可以获取到ActionContext对象。

## 执行过程

ActionMapper--> req

StrutsPreparedAndExecuteFilter ---> ActionContext

ActionProxy

Inteceptor

## 注意事项

ActionContext只能在当前请求线程内使用，无法脱离当前请求线程。如：Filter的init()方法

|  |  |
| --- | --- |
| Key | Value |
| Request | request |
| Session |  |
| Application |  |
| Attr |  |

# ThreadLocal

## 概述

ThreadLocal是一个泛型类，通过ThreadLocal可以为每个线程管理自己的变量。

## 常用方法

* get() ：获取线程局部变量的值
* set() ：设置线程局部变量的值
* remove() ：移出线程局部变量的值

## 示例

|  |
| --- |
| /\*\*  \* ThreadLocal  \* 为每条线程管理各自独立的数据  \* **@author** Terry  \*  \*/  **public** **class** TestThread {  //ThreadLocal类型静态对象  **private** **static** ThreadLocal<String> *tl* = **new** ThreadLocal<String>();  //静态变量  **private** **static** String *str* = "str";    **public** **static** **void** main(String[] args) {  //设置线程局部变量str的值  *tl*.set("def");  System.*out*.println("main.Id :" + Thread.*currentThread*().getId());    //获取线程局部变量str的值。输出结果为：def  System.*out*.println("main.str :" + *tl*.get());    //创建了一个新的线程（第二个线程）  **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  System.*out*.println("thread.run.id：" + Thread.*currentThread*().getId());  //输出结果为：str  System.*out*.println("thread.run.str：" + *str*);  //数据读取失败，输出结果为：null  System.*out*.println("thread.run.get()：" + *tl*.get());  }  }).start();  }  } |

# ActionContext的内容

ActionContext中的内容比较多，重点只需要关注如下几个对象即可：

ActionContext

request

parameter

application

attr(request 》 session 》 application 》null)

session

valustack

# ValueStack

## 概述

创建Action时,同时会创建ValueStack。ValueStack中存储了Action的相关信息。

ValueStack是一个接口，其实现类（OGNLValueStack）主要由root和context组成。

OgnlValueStack(ActionContext)

page

application

session

request

......

JSP

在action中通过ActionContext进行获取

ValueStack

Action

Action的相关属性（这些属性在调用请求处理方法之前，被set()访问器赋值）

OGNL表达式可以直接进行访问

StackContext

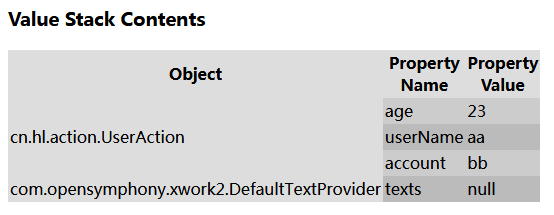
parameters

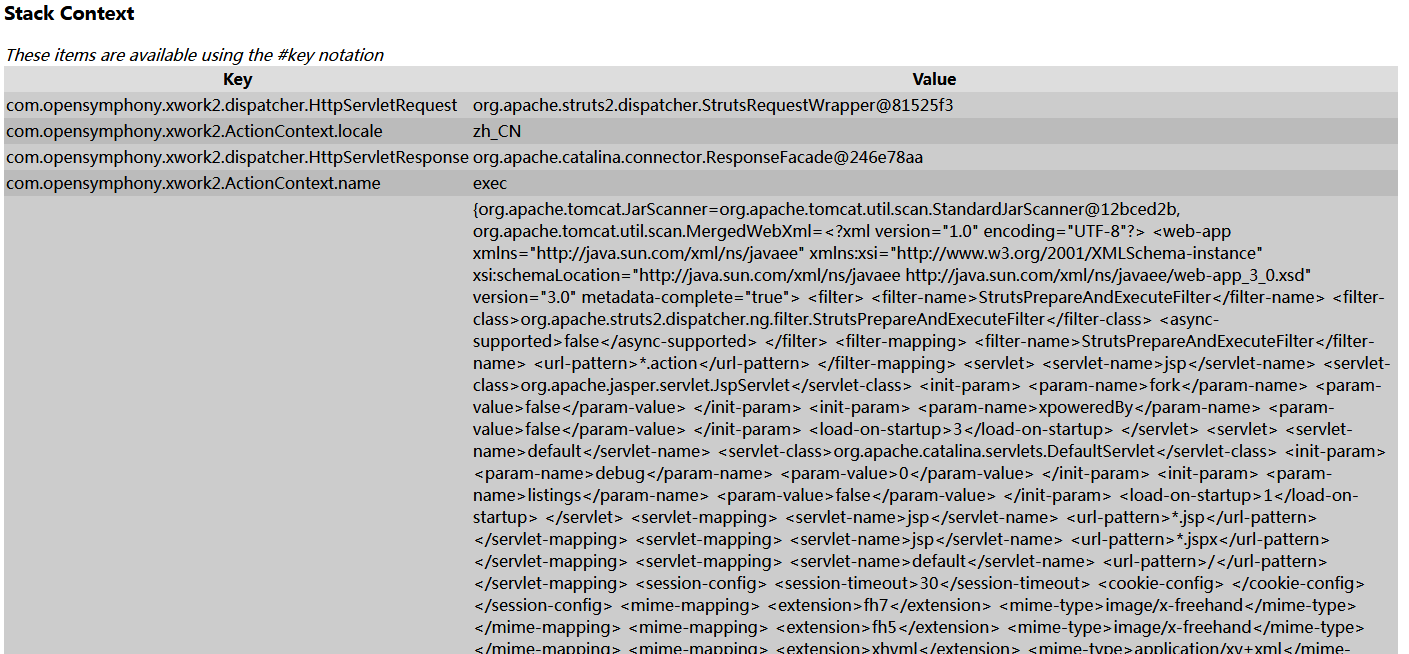
application

session

request

OGNL表达式通过#进行访问





# Ongl表达式

## 表达式

EL表达式、Reg

## 作用

用最简单的代码实现最复杂的功能。

## Ognl表达式

### 概述

OGNL是Object-Graph Navigation Language的缩写（对象图像导航语言）。通过OGNL表达式可以存取对象的属性，调用对象的方法。

### 作用

* 读取对象的静态属性，调用对象的方法
* 访问ActionContext

### 应用

* 根数据(常用数据)：直接进行访问
* 非根数据(非常用数据)：通过“#”进行访问

### 示例

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** OgnlException {  //1、创建OgnlContext对象  OgnlContext c1 = **new** OgnlContext();  //1)、存储数据简单数据  //a、存储数据  c1.put("name", "Mike");    //b、读取数据  //通过get方法直接进行读取  String name = (String)c1.get("name");  System.*out*.println("name = " + name);    //2)、存储自定义(复杂)数据  User u = **new** User("Jack","admin",20);  //a、存储数据  c1.put("user", u);    //b、读取数据  User temp = (User)c1.get("user");  System.*out*.println(temp);    //c、读取对象的userName属性  //读取非根数据  String userName = (String)c1.get("user.userName");  //读取失败，值为null  System.*out*.println("userName = " + userName);    //d、传统读取方式  String userName2 = ((User)c1.get("user")).getUserName();  System.*out*.println("userNam2 =" + userName2);    //e、Ognl读取  //非根数据#访问  Object obj1 = Ognl.*getValue*("#user.userName", c1, c1.getRoot());  System.*out*.println("obj1 = " + obj1);      //2、存储根数据  OgnlContext c2 = **new** OgnlContext();  //1)、将u设置为根  c2.setRoot(u);    //2)、读取根数据  //根数据直接访问  Object obj2 = Ognl.*getValue*("userName", c2,c2.getRoot());  System.*out*.println("obj2 = " + obj2);    } |

### Ongl和ValueStack

1. 概述

ValueStack中的数据可以通过Ognl表达式进行读取，读取过程中需要使用Struts2的标签进行显示。

1. 示例

|  |
| --- |
| [JavaBean]  **public** **class** Grade {  **private** String gradeId;  **private** String gradeName;    **public** Grade() {  // **TODO** Auto-generated constructor stub  }  **public** Grade(String gradeId, String gradeName) {  **super**();  **this**.gradeId = gradeId;  **this**.gradeName = gradeName;  }  } |
| [Action类]  **public** String ognl(){  //1、数据存储到ValueStack中  //1)、获取ValueStack对象  ValueStack stack = ActionContext.*getContext*().getValueStack();    stack.set("str","Hello Ognl");    //2)、创建grade对象  Grade g = **new** Grade("1","精英101");  stack.set("g", g);    //2、数据存储到Context中  //1)、数据存储到ContextMap中  ActionContext context = ActionContext.*getContext*();  context.getContextMap().put("g1", **new** Grade("2","精英102(ContextMap)"));    //2)、数据存储到Session中  context.getSession().put("g2", **new** Grade("3","精英103(session)"));    //3)、数据存储到application中  context.getApplication().put("g3", **new** Grade("4","精英104(application)"));    **return** "success";  } |
| [配置Action]  <package name=*"GradeAction"* extends=*"struts-default"*>  <action name=*"ognl"* class=*"cn.hl.action.GradeAction"* method=*"ognl"*>  <result>/index.jsp</result>  </action>  </package> |
| [Jsp页面]  <body>  <!-- 1、读取str的数据 -->  <!-- 直接读取:读取成功 -->  str： <s:property value=*"str"*/><br />    <!-- 使用#进行读取：读取失败 -->  str：<s:property value=*"#str"* /><br />    <!-- 使用EL表达式进行读取：读取成功 -->  str： ${str }<br />    <!-- 2、读取g的数据 -->  <!-- 直接读取：读取成功 -->  g.gradeName：<s:property value=*"g.gradeName"* /><br />  <!-- 使用#读取：读取失败 -->  g.gradeName：<s:property value=*"#g.gradeName"* /><br />  <!-- 使用EL表达式进行读取：读取成功 -->  g.gradeName：${g.gradeName }<br />    <!-- 3、读取作用域中的数据 -->  g1：<s:property value=*"#g1.gradeName"* /><br />  g2：<s:property value=*"#session.g2.gradeName"* /><br />  g3：<s:property value=*"#application.g3.gradeName"* /><br />  </body> |

1. 注意事项

* Ognl表达式读取非根数据需要通过“#”进行读取
* Ognl表达式读取到的数据需要通过Struts2标签进行显示
* attr添加数据的顺序是page >> request >> session >> application

1. 说明

因为很多框架都有自己的表达式，且通用性较差，所以一般较少用。实际应用时一般都应用EL表达式。

# 拦截器

## 概述

1. 过滤器

过滤器对所有请求和响应进行拦截(过滤)，过滤器可以过滤所有请求，如：Jsp、servlet、jpg、js等。过滤器属于容器。

1. 拦截器

拦截器在Action的请求前后被执行。拦截器可以在请求处理方法执行前进行控制。拦截器是AOP的一种体现。

拦截器是Struts2的基石，struts2的很多功能都是通过拦截器进行实现的。如：类型转换，参数映射等。

## 名词

1. 拦截器

拦截器(interceptor)可以动态的拦截Action调用的对象，拦截器在Action的执行前后进行拦截，拦截器可以阻止Action的执行，也可以对Action请求的资源进行处理。

1. 拦截器链

拦截器链(interceptor chain)也称为拦截器栈(interceptor stack)，他是有一系列拦截器组成的。其中各拦截器的执行与配置直接相关。

## 实现方式

### 继承方式

1. 实现步骤

* 创建类并继承自AbstractInterceptor
* 编写请求处理代码
* 配置拦截器
* 在Action中引用拦截器

1. 示例

|  |
| --- |
| 【Interceptor】  **public** **class** TimeInterceptor **extends** AbstractInterceptor{    /\*\*  \* 拦截器的处理方法  \* invoke将调用下一个拦截器的请求处理方法。如果后续没有拦截器，则将调用Action的请求处理方法进行执行。  \*/  @Override  **public** String intercept(ActionInvocation invocation) **throws** Exception {  //请求处理方法执行前调用  //System.out.println("before:TimeInterceptor is do......");  Long begin = System.*currentTimeMillis*();    //调用请求处理方法  String result = invocation.invoke();    //请求处理方法执行后调用  System.*out*.println("after:TimeInterceptor is do......");  Long end = System.*currentTimeMillis*();    String name = invocation.getProxy().getActionName();  System.*out*.println(name + "执行用时：" + (end - begin));    **return** result;  }  } |
| 【Interceptor配置】  <struts>  <package name=*"UserAction"* extends=*"struts-default"*>  <!-- 配置Interceptor -->  <interceptors>  <interceptor name=*"timer"* class=*"cn.hl.interceptor.TimeInterceptor"*></interceptor>  </interceptors>  </package>  </struts> |
| 【action】  /\*\*  \* 请求处理方法  \* **@return**  \*/  **public** String add(){  System.*out*.println("add() method is do......");    **return** "success";  } |
| 【Action的配置】  <struts>  <package name=*"UserAction"* extends=*"struts-default"*>  <!-- 配置Interceptor -->  <interceptors>  <interceptor name=*"timer"* class=*"cn.hl.interceptor.TimeInterceptor"*></interceptor>  </interceptors>    <!-- 配置UserAction -->  <action name=*"add"* class=*"cn.hl.action.UserAction"* method=*"add"*>  <interceptor-ref name=*"timer"*></interceptor-ref>  <result>/index.jsp</result>  </action>  </package>  </struts> |
| 【执行结果】 |

### 实现的方式

1. 步骤

* 创建类并实现Interceptor接口
* 实现接口的init()、destory和intercept()方法
* 配置Interceptor
* 引用自定义Interceptor

1. 示例

|  |
| --- |
| **public** **class** StringInterceptor **implements** Interceptor {  @Override  **public** **void** destroy() {  // **TODO** Auto-generated method stub    }  @Override  **public** **void** init() {  // **TODO** Auto-generated method stub    }  @Override  **public** String intercept(ActionInvocation invocation) **throws** Exception {  System.*out*.println("StringInterceptor is do......");  **return** invocation.invoke();  }  } |

## 配置规则

1. 多个拦截器可以组成拦截器栈

|  |
| --- |
| <!-- 配置Interceptor -->  <interceptors>  <interceptor name=*"timer"* class=*"cn.hl.interceptor.TimeInterceptor"*></interceptor>  <interceptor name=*"str"* class=*"cn.hl.interceptor.StringInterceptor"*></interceptor>    <!-- 配置拦截器链 -->  <interceptor-stack name=*"myStack"*>  <interceptor-ref name=*"timer"*></interceptor-ref>  <interceptor-ref name=*"str"*></interceptor-ref>  </interceptor-stack>  </interceptors> |
| <!-- 配置UserAction -->  <action name=*"add"* class=*"cn.hl.action.UserAction"* method=*"add"*>  <interceptor-ref name=*"myStack"*></interceptor-ref>  <result>/index.jsp</result>  </action> |

1. 如果引用了自定义的拦截器（栈）则框架默认的拦截器栈（defaultStack）将不再生效，如果需要默认拦截器栈生效，则需要进行手动引入

|  |
| --- |
| <!-- 配置UserAction -->  <action name=*"add"* class=*"cn.hl.action.UserAction"* method=*"add"*>  <interceptor-ref name=*"myStack"*></interceptor-ref>  <interceptor-ref name=*"defaultStack"*></interceptor-ref>  <result>/index.jsp</result>  </action> |

1. 拦截器(栈)之间可以相互进行引用

|  |
| --- |
| <!-- 配置拦截器链 -->  <interceptor-stack name=*"myStack"*>  <interceptor-ref name=*"timer"*></interceptor-ref>  <interceptor-ref name=*"str"*></interceptor-ref>    <!-- 引入框架默认的拦截器栈（defaultStack） -->  <interceptor-ref name=*"defaultStack"*></interceptor-ref>  </interceptor-stack>  </interceptors> |

1. 如果所有请求处理方法都应用自定义拦截器栈，可以将其设置为包内默认拦截器(栈)

|  |
| --- |
| <!-- 配置成默认的拦截器栈，包内所有Action均有效 -->  <default-interceptor-ref name=*"myStack"*></default-interceptor-ref>    <!-- 配置UserAction -->  <action name=*"add"* class=*"cn.hl.action.UserAction"* method=*"add"*>  <!--  只在当前Action中有效  <interceptor-ref name="myStack"></interceptor-ref>  -->  <result>/index.jsp</result>  </action> |

## 方法拦截器

### 概述

所谓访问拦截器就是对特定的方法进行拦截的拦截器。

### 实现

|  |
| --- |
| 【Interceptor】  **public** **class** MtdInterceptor **extends** MethodFilterInterceptor{  @Override  **protected** String doIntercept(ActionInvocation invocation) **throws** Exception {  System.*out*.println("Method Inteceptor is do......");  **return** invocation.invoke();  }  } |
| 【配置Interceptor】  <!-- 配置拦截器 -->  <interceptors>  <!-- 配置方法拦截器 -->  <interceptor name=*"shop"* class=*"cn.hl.interceptor.MtdInterceptor"*>  <!-- 包含的方法（需要进行拦截的） -->  <param name=*"includeMethods"*>add,update</param>  <!-- 排出的方法（不需要进行拦截的） -->  <param name=*"excludeMethods"*>getAll,getById</param>  </interceptor>  </interceptors> |
| 【Action】  **public** **class** ShopAction {  **public** String add(){  System.*out*.println("add method is do......");  **return** "success";  }    **public** String update(){  System.*out*.println("update method is do......");  **return** "success";  }    **public** String getAll(){  System.*out*.println("getAll method is do......");  **return** "success";  }    **public** String getById(){  System.*out*.println("getById method is do......");  **return** "success";  }  } |

### 注意

* 方法拦截器需要继承于MethodFilterInterceptor类
* 配置过程中，包含和排除部分使用Action中的方法名

### 说明

方法拦截器一般使用较少，比较适合于在电商项目中使用。

## 过滤器和拦截器

* 过滤器是通过回调函数的方式进行实现的；拦截器是通过Aop方式进行实现
* 过滤器是属于容器的；而拦截器属于Struts2框架
* 过滤器可以对所有的请求和响应进行拦截；而拦截器只能对Action进行拦截

# 上传下载

## Jsp

* + Commons-FileUpload
  + SmartFileUpload

## Struts2

* 以Commons-FileUpload为基础进行实现
* 通过拦截器进行实现

## 单文件上传

### 实现步骤

* 创建Web项目并搭建Struts2框架
* 创建文件上传表单
  + 修改method为post方式
  + 修改enctype为multipart/form-data
* 创建Action并编写请求处理方法
* 添加File类型的私有成员属性
* 添加String类型的FileName及ContentType属性
* 编写请求处理方法
* 配置Action

### 示例

|  |
| --- |
| [Jsp]  <form action=*"file/upload1.action"* method=*"post"* enctype=*"multipart/form-data"*>  <table>  <tr>  <td>  文件：<input type=*"file"* name=*"file"* />  <input type=*"submit"* value=*"上传"* />  </td>  </tr>    </table>  </form> |
| [Action]  /\*\*  \* file ：实际进行上传的文件对象。命名与表单中file元素的名称相同  \* fileName ：上传文件的名称。命名方式采用file元素名称+ "FileName"  \* contentType ：上传文件的Mime类型。命名方式采用file元素名称+"ContentType"  \*  \* String contentTypeName = inputName + "ContentType";  String fileNameName = inputName + "FileName";  \* **@author** Terry  \*  \*/  **public** **class** UploadAction {  **private** File file;  **private** String fileFileName;  **private** String fileContentType;      /\*\*  \* Struts2实现文件上传  \* **@return**  \* **@throws** IOException  \*/  **public** String upload() **throws** IOException{  //获取HttpServletRequest对象  HttpServletRequest req = (HttpServletRequest)ActionContext.*getContext*().get(StrutsStatics.*HTTP\_REQUEST*);  //通过HttpServletRequest对象获取上传路径  String path = req.~~getRealPath~~("/files");    //拼接文件名  String fileName = path + File.*separatorChar* + fileFileName;    //创建输入流对象  InputStream is = **new** FileInputStream(file);  //创建输出流对象  OutputStream os = **new** FileOutputStream(fileName);    //读取输出流中的数据并且写入到本地服务器  **byte**[] b = **new** **byte**[1024];  **int** len = 0;    //循环读取输入流中的数据  **while**((len=is.read(b))!=-1){  os.write(b, 0, len);  }    //关闭流对象  os.close();  is.close();    **return** "success";  }  } |
| [配置Action]  <action name=*"upload1"* class=*"cn.hl.action.UploadAction"* method=*"upload"*>  <result>/index.jsp</result>  </action> |

### 配置选项

* struts.multipart.maxSize ：允许上传文件的最大尺寸
* struts.multipart.saveDir ：上传的临时目录

### 说明

* File属性的名称与表单中File的name属性值相同，FileName和ContentType的命名以表单中file元素的name名称为前缀
* 临时目录在文件上传成功后会自动删除，可以不进行设置
* 如果允许上传大文件，需要设置maxSize配置选项

## 多文件上传

### 示例

|  |
| --- |
| [Jsp]  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <script type=*"text/javascript"* src=*"js/jquery-3.3.1.js"*></script>    <script type=*"text/javascript"*>  $(**function**(){  $("#btn").click(**function**(){  //拼接添加的内容（File元素、button元素、文本对象（文件））  **var** str = "<p>文件：<input type='file' name='files' />";  str += "<input type='button' onclick='del(this)' value='删除'/></p>";  //将拼接的结果添加到外层容器（名为content的div）  $("#content").append(str);  });    });    /\*\*  \*移除当前上传项  \*/  **function** del(obj){  //获取当前元素(button)的父元素然后删除  $(obj).parent().remove();  }  </script>    </head>    <body>  <form action=*"file/upload2.action"* method=*"post"* enctype=*"multipart/form-data"*>  <div id=*"content"*>  文件：<input type=*"file"* name=*"files"* />    </div>  <input type=*"button"* value=*"添加"* id=*"btn"*/>  <input type=*"submit"* value=*"上传"* />  </form>  </body>  </html> |
| [Action]  **public** **class** UploadAction2 {  **private** File[] files;  **private** String[] filesFileName;  **private** String[] filesContentType;    **public** String upload() **throws** IOException{  HttpServletRequest req = (HttpServletRequest)ActionContext.*getContext*().get(StrutsStatics.*HTTP\_REQUEST*);  //获取上传路径  String path = req.~~getRealPath~~("/files");    //遍历上传文件的集合  **for**(**int** i = 0;i<files.length;i++){  String fileName = path + File.*separatorChar* + filesFileName[i];  //获取输入流对象  InputStream is = **new** FileInputStream(files[i]);  //获取输出流对象  FileOutputStream os = **new** FileOutputStream(fileName);    //将文件写入到服务器  **byte**[] b = **new** **byte**[1024];  **int** len = 0;    //循环读取文件  **while**((len = is.read(b))!=-1){  os.write(b, 0, len);  }    os.close();  is.close();    }    **return** "success";  } |

### 注意

多文件上传时,属性必须是数组.

## Response下载

### 概述

Struts2支持使用Response对象实现文件下载.

### 示例

|  |
| --- |
| [jsp]  <body>  <a href=*"file/down1.action"*>下载</a>  </body> |
| [Action]  /\*\*  \* Response实现文件下载  \* 1、request ：用于获取文件的所在路径  \* 2、response ：用于向客户端进行输出（获取输出流对象）  \* **@return**  \* **@throws** IOException  \*/  **public** String down() **throws** IOException{  //1、获取request对象(为获取上传文件的所在路径)  HttpServletRequest req = ServletActionContext.*getRequest*();    //2、获取response对象  HttpServletResponse resp = ServletActionContext.*getResponse*();    //3、获取文件路径  String path = req.~~getRealPath~~("/files");    //4、获取下载的文件对象  File file = **new** File(path,"1.pdf");    //5、设置response对象  //1）、设置编码格式  resp.setCharacterEncoding("utf-8");    //2）、设置下载文件的长度(大小)  resp.setContentLength((**int**)file.length());    //3）、设置文件的内容(表示内容可以是任意类型)  resp.setContentType("application/octet-stream");    //4）、设置应答头(设置内容的方式-附件方式)  //fileName用于设置附件的名称  resp.setHeader("content-disposition", "attachment;filename=1.pdf");    //6、获取输出流对象  //向客户端发送数据需要通过Response完成，所以需要获取Response的输出流对象  OutputStream os = resp.getOutputStream();    //7、获取输入流对象  InputStream is = **new** FileInputStream(file);    //8、定义byte数组存储读取到的文件内容  **byte**[] b = **new** **byte**[1024];  **int** len = 0;    //9、循环读取文件内容并输出到客户端  **while**((len = is.read(b))!=-1){  os.write(b, 0, len);  }    //10、关闭对象  os.close();  is.close();    //11、返回result结果  //null表示没有返回值  **return** **null**;  } |
| [配置]  <package name=*"down"* extends=*"struts-default"*>  <action name=*"down1"* class=*"cn.hl.action.ResponseDownAction"* method=*"down"*>  <!-- result结果为null，则不需要配置result -->  </action>  </package> |

### 说明

* 文件下载中,Action的返回值类型需要配置为null
* 配置Action时,null值表示不需要配置result

## struts2下载

### 步骤

* 创建Action类
* 定义fileName属性并声称get/set访问器
* 定义请求处理方法(只需要result结果即可)
* 定义名为getInputStream的访问器
* 配置Action

### 示例

|  |
| --- |
| [jsp页面]  <body>  <a href=*"file/down2.action?fileName=1.pdf"*>下载</a>  </body> |
| [Action]  /\*\*  \* Struts2实现文件下载  \* 1、定义名为fileName的属性并提供get/set访问器  \* 2、定义请求处理方法  \* **@author** Terry  \*  \*/  **public** **class** StrutsDownAction {  //用于接收和提供待下载文件的名称  **private** String fileName;  /\*\*  \* 请求处理方法  \* **@return**  \*/  **public** String execute(){  **return** "success";  }    /\*\*  \* 获取下载文件的输入流  \* **@return**  \* **@throws** FileNotFoundException  \*/  **public** InputStream getInputStream() **throws** FileNotFoundException{  //获取Request对象（获取文件的物理路径）  HttpServletRequest req = ServletActionContext.*getRequest*();  //获取物理路径  String path = req.~~getRealPath~~("/files");    **return** **new** FileInputStream(**new** File(path,fileName));  }      **public** String getFileName() {  **return** fileName;  }  **public** **void** setFileName(String fileName) {  **this**.fileName = fileName;  }  } |
| [配置Action]  <action name=*"down2"* class=*"cn.hl.action.StrutsDownAction"*>  <!-- result的type必须配置为stream -->  <result type=*"stream"*>  <param name=*"contentDisposition"*>  attachment;filename=${fileName}  </param>  </result>  </action> |

### StreamResult

|  |
| --- |
| resolveParamsFromStack(invocation.getStack(), invocation); |

从栈中读取流对象

### 说明

* Action中必须提供名为getInputStream()的访问器
* Action中的result的类型必须设置为stream

# Struts2 异步请求

## ServletApi实现Ajax

1. 示例

|  |
| --- |
| [jsp]  <head>    <script type=*"text/javascript"* src=*"js/jquery-3.3.1.js"*></script>  <script type=*"text/javascript"*>  $(**function**(){  $("#btn").click(**function**(){  $.get("ajax/add.action",{"id":"1"},**function**(data){  alert(data);  },"text");  });  });  </script>  </head>    <body>  <input type=*"button"* value=*"Ajax"* id=*"btn"*/>  </body> |
| [Action]  /\*\*  \* ServletApi实现Ajax  \* **@author** Terry  \*  \*/  **public** **class** AjaxAction {  **private** **int** id;    /\*\*  \* 请求处理方法  \* **@return**  \* **@throws** IOException  \*/  **public** String execute() **throws** IOException{  //通过ServletApi获取Response对象  HttpServletResponse resp = ServletActionContext.*getResponse*();    //通过Response对象进行响应  **if**(id>0){  resp.getWriter().print("Yes");  }  **else**{  resp.getWriter().print("No");  }    //对于Ajax的响应只能通过Response对象进行  //所以返回值设置为null  **return** **null**;  }    } |
| [配置Action]  <struts>  <package name=*"AjaxCtionb"* extends=*"struts-default"* namespace=*"/ajax"*>  <action name=*"add"* class=*"cn.hl.action.AjaxAction"*>  <!-- Ajax只能通过Response对象进行响应，不能进行页面跳转 -->  </action>  </package>  </struts> |

1. 注意事项

* 请求处理方法中，只能通过Response对象对Ajax请求进行响应，不能具有任何返回值。只能是null
* 配置过程中，null表示无返回值，不需要配置任何result

## Json数据

### 概述

在Servlet中我们是通过第三方jar包实现Json数据转换，如：Gson、Json等。在Struts中直接封装了相关的工具类。

### jar包

|  |
| --- |
|  |

### 说明

* struts2-json-plugin ：将对象转为Json数据

### 步骤

* 创建Action类
* 定义JSONArray类型的属性并生成get/set访问器
* 编写请求处理方法并将数据转为JSONArray对象
* 配置Action

### 示例

|  |
| --- |
| [Jsp]  <head>  <script type=*"text/javascript"* src=*"js/jquery-3.3.1.js"*></script>  <script type=*"text/javascript"*>  $(**function**(){  $("#btn").click(**function**(){  $.get("json/list.action",**function**(data){  alert(data);  },"text");  });  });  </script>  </head>    <body>  <input type=*"button"* value=*"Json"* id=*"btn"*/>  </body> |
| [User Bean]  **public** **class** User {  **private** **int** userId;  **private** String userName;  **private** **int** age;  } |
| [Action]  **public** **class** JsonAction {  //定义私有的JSONArray类型对象  **private** JSONArray root;    /\*\*  \* 请求处理方法  \* **@return**  \*/  **public** String list(){  //创建集合对象  List<User> list = **new** ArrayList<User>();  list.add(**new** User(1,"Mike",20));  list.add(**new** User(2,"Jack",24));  list.add(**new** User(3,"Tom",22));  list.add(**new** User(4,"Jerry",20));    //将对象转为JSONArray类型的数据  root = JSONArray.*fromObject*(list);    **return** "success";  }  **public** JSONArray getRoot() {  **return** root;  }  **public** **void** setRoot(JSONArray root) {  **this**.root = root;  }  } |
| [配置Action]  <!--  需要继承于json-default  web app libraries 》》 struts2-json-plugin >> org.apache.struts.json 》》 struts-plugin  -->  <package name=*"JsonAction"* extends=*"json-default"* namespace=*"/json"*>  <action name=*"list"* class=*"cn.hl.action.JsonAction"* method=*"list"*>  <!-- result的返回值类型必须为json -->  <result type=*"json"*>  <!-- 元素的值与Action中JSONArray类型对象的名称相同 -->  <param name=*"root"*>root</param>  </result>  </action>  </package> |

### 部分源代码(struts2-json-plugin)

1. JSONInterceptor

|  |
| --- |
| @SuppressWarnings("unchecked")  **public** String intercept(ActionInvocation invocation) **throws** Exception {  HttpServletRequest request = ServletActionContext.*getRequest*();  HttpServletResponse response = ServletActionContext.*getResponse*();  ........  Object rootObject = **null**;  //判断root是否是null  //如果不为空则读取ValueStack中的JSONArray数据  **if** (**this**.root != **null**) {  //获取当前Action的ValueStack（值栈）  ValueStack stack = invocation.getStack();  //读取值栈中root所对应JSONArray类型的数据  rootObject = stack.findValue(**this**.root);  **if** (rootObject == **null**) {  **throw** **new** RuntimeException("Invalid root expression: '" + **this**.root + "'.");  }  }  ..........  } |
| //通过读取配置文件为变量root赋值  **public** **void** setRoot(String root) {  **this**.root = root;  } |

1. struts-plugin.xml

|  |
| --- |
| <struts>  <package name=*"json-default"* extends=*"struts-default"*>  <result-types>  <result-type name=*"json"* class=*"org.apache.struts2.json.JSONResult"*/>  </result-types>  .........  </package>  </struts> |

### 说明

* Action类中必须声明JSONArray类型的对象
* 请求处理方法中必须调用JSONArray对象的fromObject()方法转换bean(list)对象
* Action所属的包必须继承于json-default。result的类型必须是json
* result中必须提供名为root的参数，参数的值与Action类中JSONArray类型对象的名称相同