



# Struts2

讲师：佟刚

新浪微博：尚硅谷-佟刚

# 使用 Filter 作为控制器的 MVC

讲师：佟刚

新浪微博:@尚硅谷-佟刚

# MVC 设计模式概览

- 实现 MVC(Model、View、Controller) 模式的应用程序由 3 大部分构成：

- 模型：封装应用程序的数据和业务逻辑 **POJO(Plain Old Java Object)**
- 视图：实现应用程序的信息显示功能 **JSP**
- 控制器：接收来自用户的输入，调用模型层，响应对应的视图组件 **Servlet** **Filter**

- 需求:

`http://localhost:8989/struts2--1/Product_input.action`

显示表单

**Add a product**

Product Name:

Description:

Price:

保存表单数据到数据库

`http://localhost:8989/struts2--1/Product_save.action`

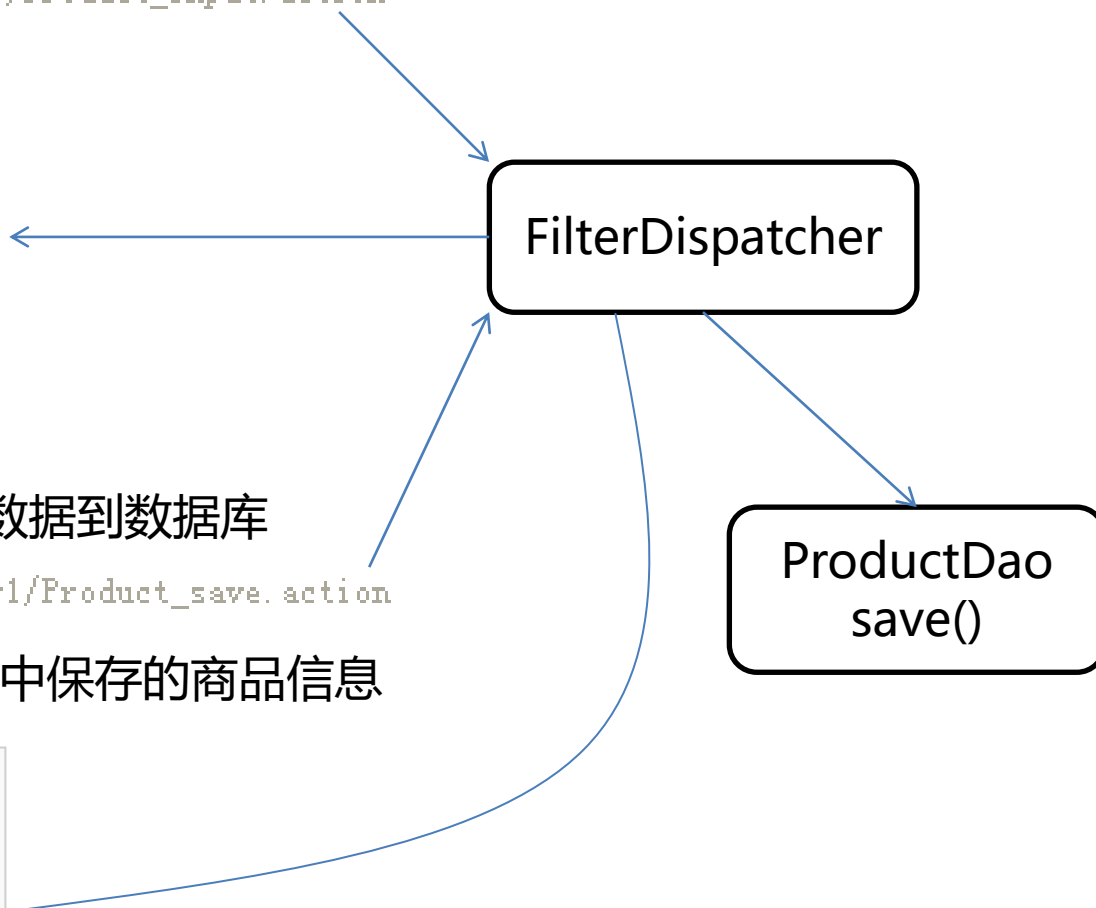
显示数据库中保存的商品信息

**The product has been saved.**

**Details:**

Product Name: Book  
Description: Struts2  
Price: \$50

- 流程:



# 使用 Filter 作为控制器的 MVC

- 目录结构
  - 📁 day\_48\_struts2
    - 📁 src
      - 📁 com.atguigu.app
        - 📄 FilterDispatcher.java
        - 📄 Product.java
      - 📖 JRE System Library [jdk1.7.0\_01]
      - 📖 Apache Tomcat v6.0 [Apache Tomcat v6.0]
      - 📁 build
      - 📁 WebContent
        - 📁 META-INF
        - 📁 WEB-INF
          - 📁 lib
          - 📁 pages
            - 📄 detail.jsp
            - 📄 input.jsp
          - 📄 web.xml
        - 📄 index.jsp

# 使用 Filter 作为控制器的 MVC

- 使用 Filter 作为控制器的好处
  - 使用一个过滤器来作为控制器, 可以方便地在应用程序里对**所有资源**(包括静态资源)进行控制访问.

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

## Servlet VS Filter

1. Servlet 能做的 Filter 是否都可以完成? 嗯。
2. Filter 能做的 Servlet 都可以完成吗? 拦截资源却不是 Servlet 所擅长的!  
Filter 中有一个 FilterChain, 这个 API 在 Servlet 中没有!

# Hello Strtus2

讲师：佟刚

新浪微博:@尚硅谷-佟刚

# Struts2 概述

- Struts2 是一个用来开发 MVC 应用程序的框架. 它**提供了 Web 应用程序开发过程中的一些常见问题的解决方案**:
  - 对来自用户的**输入数据进行合法性验证**
  - **统一的布局**
  - **可扩展性**
  - **国际化和本地化**
  - 支持 **Ajax**
  - **表单的重复提交**
  - **文件的上传下载**
  - .....



# Struts2 VS Struts1

- 在体系结构方面更优秀:
  - 类更少, 更高效: 在 **Struts2** 中无需使用 **“ActionForm”** 来封装请求参数.
  - 扩展更容易: Struts2 通过**拦截器**完成了框架的大部分工作. 在 Struts2 中**插入一个拦截器**对象相当简便易行.
- 更容易测试:
  - **即使不使用浏览器**也可以对基于 Struts2 的应用进行测试

# 从 Struts1 升级到 Struts2

- Struts2 从本质上讲已**不是从 Struts1 扩展而来的**, 说它是一个**换了品牌标签的 WebWork** 更合适
- 从 Struts1 升级到 Struts2:
  - Struts1 里使用 ActionServlet 作为控制器; **Struts2 使用了一个过滤器作为控制器**
  - Struts1 中每个 HTML 表单都对应一个 ActionForm 实例. **Struts2 中, HTML 表单将被直接映射到一个 POJO.**
  - Struts1 的验证逻辑编写在 ActionForm 中; **Struts2 中的验证逻辑编写在 Action 中.**
  - Struts1 中, Action 类必须继承 org.apache.struts.action.Action 类; **Struts2 中任何一个 POJO 都可以是一个 Action 类.**
  - **Struts2 在页面里使用 OGNL 来显示各种对象模型**, 可以不再使用 EL 和 JSTL

# 下载 Struts2

- 打开浏览器输入 <http://struts.apache.org/>
- 点击超链接 “[Struts 2.3.x](#)”，打开下载页面

12 May 2012 - Struts 2.3.4 General Availability Release

The latest production release of Struts 2 is [Struts 2.3.4](#), which was promote [version notes](#) are available online.

- 点击 “[struts-2.3.x-all.zip](#)” 下载

# Struts2 的 Hello World

- 需求:

`http://localhost:8989/struts2--1/Product_input.action`



### Add a product

Product Name:

Description:

Price:

`http://localhost:8989/struts2--1/Product_save.action`

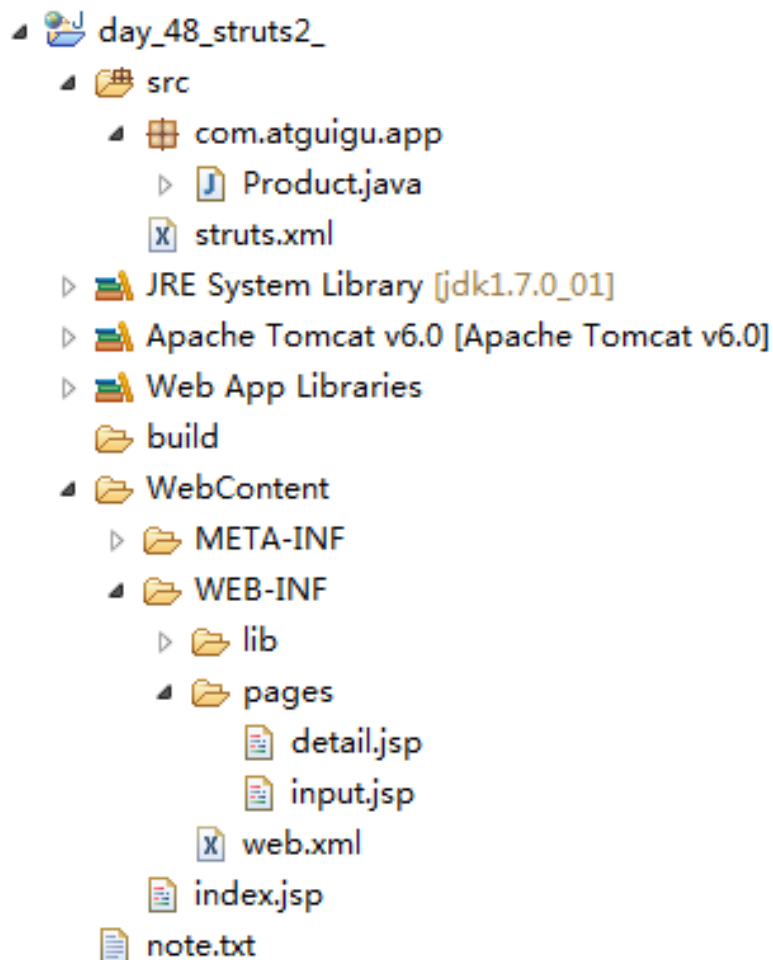


**The product has been saved.**

**Details:**

Product Name: Book  
Description: Struts2  
Price: \$50

- 目录结构:



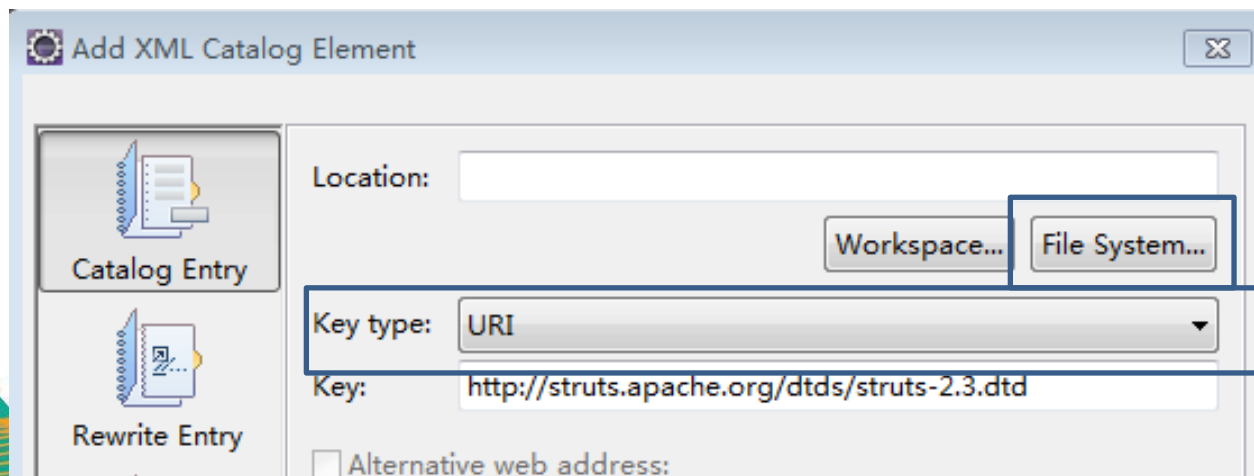
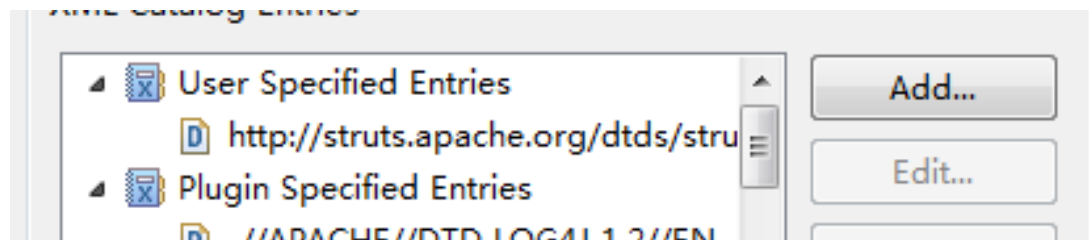
# Struts2 的 Hello World

- 搭建 Struts2 的环境:
  - **加入 jar 包**: 复制 struts\apps\struts2-blank\WEB-INF\lib 下的所有 jar 包到当前 web 应用的 lib 目录下.
  - **在 web.xml 文件中配置 struts2**: 复制 struts\apps\struts2-blank1\WEB-INF\web.xml 文件中的过滤器的配置到当前 web 应用的 web.xml 文件中
  - **在当前 web 应用的 classpath 下添加 struts2 的配置文件 struts.xml**: 复制 struts1\apps\struts2-blank\WEB-INF\classes 下的 struts.xml 文件到当前 web 应用的 src 目录下.

# 添加 DTD 约束

```
<!DOCTYPE struts PUBLIC  
"-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"  
"http://struts.apache.org/dtds/struts-2.3.dtd">
```

- XML
  - DTD Files
  - XML Catalog
  - XML Files
  - XML Schema Files
  - XPath
  - XSL



# 添加 DTD 约束

struts-2.3.4-all\struts-2.3.4\src\core\src\main\resources

名称

- META-INF
- org
- template
- FREEMARKER-LICENSE.txt
- LICENSE.txt
- NOTICE.txt
- OGNL-LICENSE.txt
- overview.html
- struts.vm
- struts-2.0.dtd
- struts-2.1.7.dtd
- struts-2.1.dtd
- struts-2.3.dtd
- struts-default.xml
- XWORK-LICENSE.txt

http://localhost:8080/struts2-2/Product-input.action

```
<action name="Product-input">  
  <result>/WEB-INF/pages/product-form.jsp</result>  
</action>
```

ProductName<sup>^</sup>:

Price<sup>^</sup>:

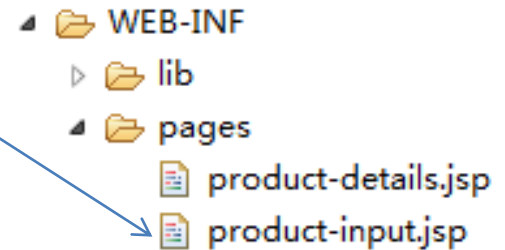
```
<action name="Product_save" class="org.simpleit.struts2.Product" method="execute">  
  <result name="success"/WEB-INF/pages/ProductDetails.jsp</result>  
</action>
```

```
public String execute(){  
  System.out.println("save: " + this);  
  return "success";  
}
```



http://localhost:8989/struts2-2/**product-input.action**

```
<action name="product-input">  
  <result>/WEB-INF/pages/product-input.jsp</result>  
</action>
```



http://localhost:8989/struts2-2/**product-save.action**

```
<action name="product-save" class="com.atguigu.struts2.app.Product" method="add">  
  <result name="details">/WEB-INF/pages/product-details.jsp</result>  
</action>
```

```
public String add(){  
  System.out.println("save: " + this);  
  return details;  
}
```

- 编辑 struts.xml 文件: struts.xml 文件是对 struts 应用程序里的 Action 进行配置的地方.
- 配置 package 元素

Struts2 把各种 action 分门别类地组织成不同的包. 可以把包想象为一个模块. 一个典型的 struts.xml 文件可以有一个或多个包

每个 package 元素都必须有一个 name 属性. 该 name 属性可被其它 package 引用

namespace 属性是可选的, 如果它没有给出, 则以 "/" 为默认值. 若 namespace 有一个非默认值, 则要想调用这个包里的 Action, 就必须把这个属性所定义的命名空间添加到有关的 URI 字符串里

```
<struts>
  <package name="struts-app2" namespace="/" extends="struts-default">
    <action name="Product_input">
      <result>/WEB-INF/jsp/ProductForm.jsp</result>
    </action>
    <action name="Product_save" class="org.simpleit.app.Product">
      <result>/WEB-INF/jsp/ProductDetails.jsp</result>
    </action>
  </package>
</struts>
```

package 元素通常要对 struts-default.xml 文件里定义的 struts-default 包进行扩展. 这么做了以后, 包里的所有动作就可以使用在 struts-default.xml 文件里的结果类型和拦截器了.

- 配置 action 元素

每个 action 都必须有一个 name 属性, 该属性和用户请求 servletPath 之间存在着一一对应关系

```
<struts>

  <package name="struts-app2" namespace="/" extends="struts-default">

    <action name="Product_input">
      <result>/WEB-INF/jsp/ProductForm.jsp</result>
    </action>

    <action name="Product_save" class="org.simpleit.app.Product">
      <result>/WEB-INF/jsp/ProductDetails.jsp</result>
    </action>

  </package>

</struts>
```

http://localhost:8989/struts2/Product\_input action

action 元素的 class 属性是可选的. 如果没有配置 class 属性, Struts 将把 com.opensymphony.xwork2.ActionSupport 作为其默认值. 如果配置了 class 属性, 还可以使用 method 属性配置该类的一个动作方法. method 属性的默认值为 execute

action 元素嵌套在 package 元素内部, 它表示一个 Struts 请求.

# Struts2 的 Hello World

## • 配置 result 元

result 元素 <action> 的一个子元素, 它告诉 struts 在完成动作后把控制权转交到哪里. result 元素(的 name 属性)对应着 Action 方法的返回值. 因为动作方法在不同情况下可能返回不同的值, 所以同一个 action 元素可能会有多个 result 元素

```
<struts>
```

```
<package name="struts-app2" namespace="/" extends="struts-default">
```

```
<action name="Product_input">
```

```
<result>/WEB-INF/jsp/ProductForm.jsp</result>
```

```
</action>
```

```
<action name="Product_save" class="org.simpleit.app.Product">
```

```
<result name="success" type="dispatcher"/WEB-INF/jsp/ProductDetails.jsp</result>
```

```
</action>
```

```
</package>
```

```
</struts>
```


result 元素的 name 属性建立 <result> 和 Action 方法返回值之间的映射关系。

name 属性的默认值为 “success”


result 元素的 type 属性负责指定结果类型. type 属性的值必须是在包含当前包或者是当前包的父包里注册过的结果类型. type 属性的默认值为 dispatcher

```
public String execute() {  
    return "success";  
}
```

```
<action name="Product_input">  
    <result>/jsp/ProductForm.jsp</result>  
</action>
```



```
<action name="Product_input"  
    class="com.opensymphony.xwork2.ActionSupport" method="execute">  
    <result name="success" type="dispatcher">/jsp/ProductForm.jsp</result>  
</action>
```



```
public String execute() throws Exception {  
    return SUCCESS;  
}
```

# Action

讲师：佟刚

新浪微博:@尚硅谷-佟刚

# Action 类

- action: 应用程序可以完成的每一个操作. 例如: 显示一个登陆表单; 把产品信息保存起来
- Action类: 普通的 Java 类, 可以有属性和方法, 同时必须遵守下面这些规则:
  - **属性的名字必须遵守与 JavaBeans 属性名相同的命名规则.** 属性的类型可以是任意类型. 从字符串到非字符串(基本数据库类型)之间的数据转换可以自动发生
  - **必须有一个不带参的构造器**
  - **至少有一个供 struts 在执行这个 action 时调用的方法**
  - **同一个 Action 类可以包含多个 action 方法.**
  - **Struts2 会为每一个 HTTP 请求创建一个新的 Action 实例**

# 访问 web 资源

- 在 Action 中, 可以通过以下方式访问 web 的 HttpSession, HttpServletRequest, HttpServletResponse 等资源

## – 与 Servlet API 解耦的访问方式

- 通过 `com.opensymphony.xwork2.ActionContext`
- 通过 Action 实现如下接口

```
org.apache.struts2.interceptor.ApplicationAware;  
org.apache.struts2.interceptor.RequestAware;  
org.apache.struts2.interceptor.SessionAware;
```

## – 与 Servlet API 耦合的访问方式

- 通过 `org.apache.struts2.ServletActionContext`
- 通过实现对应的 XxxAware 接口



# 与Servlet API解耦的访问方式

- 为了避免与 Servlet API 耦合在一起, 方便 Action 做**单元测试**, Struts2 对 HttpServletRequest, HttpSession 和 ServletContext 进行了封装, 构造了 3 个 Map 对象来替代这 3 个对象, 在 Action 中可以直接使用 HttpServletRequest, HttpSession, ServletContext 对应的 **Map 对象来保存和读取数据**.

# 通过 ActionContext 访问 Web 资源

- **ActionContext** 是 Action 执行的上下文对象, 在 ActionContext 中保存了 Action 执行所需要的**所有对象**, 包括 parameters, request, session, application 等.
- 获取 HttpSession 对应的 Map 对象:
  - public Map getSession()
- 获取 ServletContext 对应的 Map 对象:
  - public Map getApplication()
- 获取请求参数对应的 Map 对象:
  - public Map getParameters()
- 获取 HttpServletRequest 对应的 Map 对象:
  - public Object get(Object key): ActionContext 类中没有提供类似 getRequest() 这样的方法来获取 HttpServletRequest 对应的 Map 对象. 要得到 HttpServletRequest 对应的 Map 对象, 可以**通过为 get() 方法传递 “request” 参数实现**

# 通过实现 Aware 接口访问 Web 资源

- Action 类通过可以实现某些特定的接口, 让 Struts2 框架在运行时向 Action 实例**注入** parameters, request, session 和 application 对应的 Map 对象:

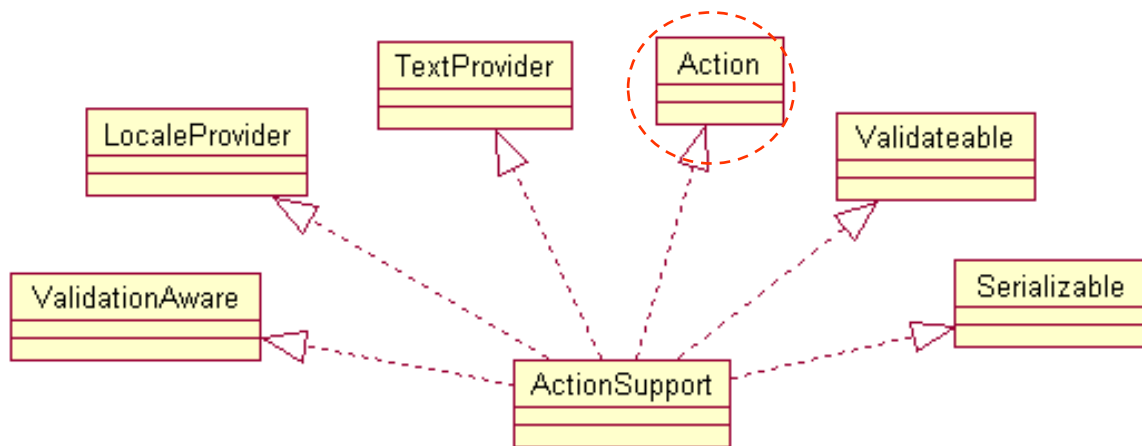
```
org.apache.struts2.interceptor.ApplicationAware;  
org.apache.struts2.interceptor.ParameterAware;  
org.apache.struts2.interceptor.RequestAware;  
org.apache.struts2.interceptor.SessionAware;
```

# 与 Servlet 耦合的访问方式

- 直接访问 Servlet API 将使 Action 与 Servlet 环境耦合在一起, 测试时需要有 Servlet 容器, 不便于对 Action 的单元测试.
- 直接获取 HttpServletRequest 对象:
  - **ServletActionContext**.getRequest()
- 直接获取 HttpSession 对象
  - ServletActionContext.getRequest().getSession()
- 直接获取 ServletContext 对象
  - ServletActionContext.getServletContext()
- 通过实现 ServletRequestAware, ServletContextAware 等接口的方式

# ActionSupport

- `com.opensymphony.xwork2.ActionSupport` 类是默认的 Action 类.
- 在编写 Action 类时, 通常会对这个类进行扩展



# 练习

`http://localhost:8989/struts2--2/Login_Ui`



**Login**

User Name:

Password:



`http://localhost:8989/struts23/User_login.action`



Welcome: Tom

Welcome. Number of users online: 1

[Log out](#)

把用户信息存入 Session 域中, 在线人数 + 1

显示当前在线人数

当前在线人数 - 1, Session 失效

`http://localhost:8989/struts23/User_logout.action`



result

讲师：佟刚

新浪微博:@尚硅谷-佟刚

# result

- 每个 action 方法都将返回一个 String 类型的值, Struts 将根据这个值来决定响应什么结果.
- 每个 action 声明都必须包含有数量足够多的 result 元素, 每个 result 元素分别对应着 action 方法的一个返回值.
- result 元素可以有下面两个属性
  - name: 结果的名字, 必须与 Action 方法的返回值相匹配, 默认值为 success
  - type: 响应结果的类型. 默认值为 dispatcher

```
<action name="Product_input"  
    class="com.opensymphony.xwork2.ActionSupport" method="execute">  
    <result name="success" type="dispatcher">/jsp/ProductForm.jsp</result>  
</action>
```

```
public String execute() throws Exception {  
    return SUCCESS;  
}
```



# 结果类型

```
<result-types>
  <result-type name="chain" class="com.op
  <result-type name="dispatcher" class="o
  <result-type name="freemarker" class="o
  <result-type name="httpheader" class="o
  <result-type name="redirect" class="org
  <result-type name="redirectAction" clas
  <result-type name="stream" class="org.a
  <result-type name="velocity" class="org
  <result-type name="xslt" class="org.apa
  <result-type name="plainText" class="or
</result-types>
```

# 结果类型: dispatcher

- dispatcher 结果类型是**最常用的结果类型**, 也是 struts 框架默认的结果类型
- 该结果类型有一个 location 参数, 它是一个默认参数

```
<result name="success">  
  <param name="location"/WEB-INF/jsp/Login.jsp</param>  
</result>
```

等同

```
<result name="success"/WEB-INF/jsp/Login.jsp</result>
```

- dispatcher 结果类型将把控制权**转发**给应用程序里的指定资源.
- dispatcher 结果类型不能把控制权**转发**给一个外部资源. 若需要把控制权重定向到一个外部资源, 应该使用 redirect 结果类型

# 结果类型: redirect

- redirect 结果类型将把响应**重定向**到另一个资源, 而不是转发给该资源.
- redirect 结果类型接受下面这些参数:
  - location: 用来给出重定向的目的地.它是默认属性
  - **parse**: 用来表明是否把 location 参数的值视为一个 **OGNL** 表达式来解释. 默认值为 true
- redirect 结果类型可以把响应重定向到一个外部资源
- 实例代码:

```
<action name="User_logout" class="org.simpleit.struts2.app1.User" method="logout">
  <result name="success" type="redirect" >
    <param name="location">/aware-test.jsp?name=${name}</param>
  </result>
</action>
```

# 结果类型: redirectAction

- redirectAction 结果类型把响应重定向到另一个 Action
- redirectAction 结果类型接受下面这些参数:
  - actionName: 指定 “目的地” action 的名字. 它是默认属性
  - namespace: 用来指定 “目的地” action 的命名空间. 如果没有配置该参数, Struts 会把当前 Action 所在的命名空间作为 “目的地” 的命名空间
- 示例代码:

```
<action name="User_logout" class="org.simpleit.struts2.app1.User" method="logout">
  <result type="redirectAction">
    <param name="actionName">Login_Ui</param>
    <param name="nameSpace"/></param>
  </result>
</action>
```

# 结果类型: chain

- chain 结果类型的基本用途是构成一个 action 链: 前一个 action 把控制权**转发**给后一个 action, 而前一个 action 的状态在后一个 action 中依然保持
- chain 结果类型接受下面这些参数:
  - actionName: 指定目标 action 的名字. 它是默认属性
  - namespace: 用来指定 “目的地” action 的命名空间. 如果没有配置该参数, Struts 会把当前 action 所在的命名空间作为 “目的地” 的命名空间
  - method: 指定目标 action 方法. 默认值为 execute

# 通配符映射

- 一个 Web 应用可能有成百上千个 action 声明. 可以利用 struts 提供的**通配符映射机制**把多个彼此相似的映射关系简化为一个映射关系
- 通配符映射规则
  - 若找到多个匹配, **没有通配符的那个将胜出**
  - **若指定的动作不存在**, Struts 将会尝试把这个 URI 与任何一个包含着通配符 **\*** 的动作名及进行匹配
  - **被通配符匹配到的 URI 字符串的子串可以用 {1}, {2} 来引用**. {1} 匹配第一个子串, {2} 匹配第二个子串...
  - **{0} 匹配整个 URI**
  - 若 Struts 找到的带有通配符的匹配不止一个, **则按先后顺序进行匹配**
  - \* 可以匹配零个或多个字符, 但不包括 / 字符. 如果想把 / 字符包括在内, 需要使用 \*\*. 如果需要对某个字符进行转义, 需要使用 \.

# 通配符映射示例(1)

- 包声明:

```
<package name="struts-app3" namespace="/app3" extends="struts-default">  
  
    <action name="(*)_add" class="org.simpleit.app.Book" method="add">  
        <result>/WEB-INF/jsp/book.jsp</result>  
    </action>  
  
</package>
```

- 上面的包声明可以由正确的命名空间和\_add 组成的 URI 来调用, 包括:

```
/app3/Book_add.action  
/app3/Author_add.action  
/app3/_add.action  
/app3/Whatever_add.action
```

## 通配符映射示例(2)

- 包声明:

```
<package name="struts-app3" namespace="/app3" extends="struts-default">

    <action name="Book_add" class="org.simpleit.app.Book" method="add">
        <result>/WEB-INF/jsp/Book.jsp</result>
    </action>

    <action name="Author_add" class="org.simpleit.app.Author" method="add">
        <result>/WEB-INF/jsp/Author.jsp</result>
    </action>

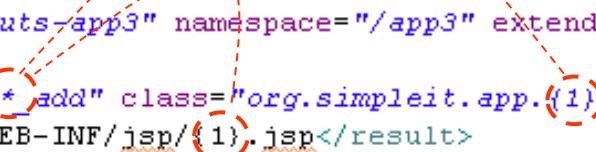
</package>
```

- 上面的包可改写为:

```
<package name="struts-app3" namespace="/app3" extends="struts-default">

    <action name="*add" class="org.simpleit.app.{1}" method="add">
        <result>/WEB-INF/jsp/{1}.jsp</result>
    </action>

</package>
```





- 包声明:

```
<package name="struts-app3" namespace="/app3" extends="struts-default">

    <action name="Book_add" class="org.simpleit.app.Book" method="add">
        <result>/WEB-INF/jsp/Book.jsp</result>
    </action>

    <action name="Book_delete" class="org.simpleit.app.Book" method="delete">
        <result>/WEB-INF/jsp/Book.jsp</result>
    </action>

    <action name="Author_add" class="org.simpleit.app.Author" method="add">
        <result>/WEB-INF/jsp/Author.jsp</result>
    </action>

    <action name="Author_delete" class="org.simpleit.app.Author" method="delete">
        <result>/WEB-INF/jsp/Author.jsp</result>
    </action>

</package>
```

- 上面的包可改写为:

```
<package name="struts-app3" namespace="/app3" extends="struts-default">

    <action name="*(*)" class="org.simpleit.app.{1}" method="{2}">
        <result>/WEB-INF/jsp/{1}.jsp</result>
    </action>

</package>
```

# 动态方法调用

- 动态方法调用: 通过 url 动态调用 Action 中的方法
- action 声明:

```
<package name="struts-app2" namespace="/" extends="struts-default">  
    <action name="Product" class="org.simpleit.app.Product">
```

- URI:
  - /struts-app2/Product.action: Struts 调用 Product 类的 execute
  - /struts-app2/Product!**save**.action: Struts 调用 Product 类的 save() 方法
- 默认情况下, Struts 的动态方法调用处于禁用状态

# OGNL

讲师：佟刚

新浪微博：@尚硅谷-佟刚

# 从页面显示说起

- Struts2 的 helloWorld 中



- 在 *ProductDetails.jsp* 页面中打印 request 隐含对象:

`org.apache.struts2.dispatcher.StrutsRequestWrapper@9bad5a`

# 值栈

- debug 跟踪 StrutsRequestWrapper 的 getAttribute() 方法, 当传入参数为 “productName” 时, ActionContext 对象的即时状态如下:

s	"productName" (id=31)
ctx	ActionContext (id=4)
context	OgnlContext (id=459)
_accessorStack	ArrayList<E> (id=46)
_classResolver	CompoundRootAccessor
_currentEvaluation	null
_currentNode	ASTConst (id=493)
_currentObject	CompoundRoot (id=46)
_keepLastEvaluation	false
_lastEvaluation	null
_localReferenceCounter	0
_localReferenceMap	null
_memberAccess	SecurityMemberAccess
_root	CompoundRoot (id=46)
_rootEvaluation	null
_traceEvaluations	false
_typeConverter	OgnlTypeConverterWra
_typeStack	ArrayList<E> (id=47)
_values	HashMap<K, V> (id=47)
entrySet	HashMap\$EntrySet (i
keySet	null
loadFactor	0.75
modCount	26
size	26
table	HashMap\$Entry<K, V>[6
threshold	48
values	null

[53]	HashMap\$Entry<K, V> (id=527)
hash	-1292118411
key	"com.opensymphony.xwork2.util.ValueStack.ValueSta
next	HashMap\$Entry<K, V> (id=540)
value	OgnlValueStack (id=455)
context	OgnlContext (id=459)
defaultType	null
devMode	false
logMissingProperties	false
ognlUtil	OgnlUtil (id=240)
overrides	null
root	CompoundRoot (id=463)
securityMemberAccess	SecurityMemberAccess (id=469)

•ValueStack(值栈): 贯穿整个 Action 的生命周期(每个 Action 类的对象实例都拥有一个 ValueStack 对象). 相当于一个数据的中转站. 在其中保存当前 Action 对象和其他相关对象.

•Struts 框架把 ValueStack 对象保存在名为 “struts.valueStack” 的请求属性中

# 值栈

- 在 ValueStack 对象的内部有两个逻辑部分:

[-] ▲ value	OgnlValueStack (id=455)
[-] ▲ context	OgnlContext (id=459)
▲ defaultType	null
■ devMode	false
■ logMissingProperties	false
[-] ▲ ognlUtil	OgnlUtil (id=240)
▲ overrides	null
[-] ▲ root	CompoundRoot (id=463)
[-] ▲ securityMemberAccess	SecurityMemberAccess (id=469)

```
public class CompoundRoot extends ArrayList
[Product [productName=Cpu, productPrice=1000], com.
```

**ObjectStack: Struts 把 Action 和相关对象压入 ObjectStack 中**

**ContextMap: Struts 把各种各样的映射关系(一些 Map 类型的对象) 压入 ContextMap 中. 实际上就是对 ActionContext 的一个引用**

**Struts 会把下面这些映射压入 ContextMap 中**

- parameters: 该 Map 中包含当前请求的请求参数
- request: 该 Map 中包含当前 request 对象中的所有属性
- session: 该 Map 中包含当前 session 对象中的所有属性
- application: 该 Map 中包含当前 application 对象中的所有属性
- attr: 该 Map 按如下顺序来检索某个属性: request, session, application

# 值 栈

- 在 ValueStack 对象的内部有两个逻辑部分:
  - ObjectStack: Struts 把 Action 和相关对象压入 ObjectStack 中
  - ContextMap: Struts 把各种各样的映射关系(一些 Map 类型的对象) 压入 ContextMap 中. 实际上就是对 ActionContext 的一个引用
- Struts 会把下面这些映射压入 ContextMap 中
  - parameters: 该 Map 中包含当前请求的请求参数
  - request: 该 Map 中包含当前 request 对象中的所有属性
  - session: 该 Map 中包含当前 session 对象中的所有属性
  - application: 该 Map 中包含当前 application 对象中的所有属性
  - attr: 该 Map 按如下顺序来检索某个属性: request, session, application

**ObjectStack(root 属性)**

Object0
Object1
Objectn

**ContextMap(context 属性)**

request
attr
parameters
session
application

# OGNL

- 在 JSP 页面上可以利用 OGNL(Object-Graph Navigation Language: 对象-图导航语言) 访问到值栈 (ValueStack) 里的对象属性.
- **若希望访问值栈中 ContextMap 中的数据, 需要给 OGNL 表达式加上一个前缀字符 #. 如果没有前缀字符 #, 搜索将在 ObjectStack 里进行.**



# property 标签

- Struts 的 property 标签用来输出值栈中的一个属性值.

名字	类型	默认值	说明
default	String		可选, 如果 value 值为 null 或没有给定, 将显示该属性值
escape	boolean	true	可选, 是否要对HTML特殊字符进行转义
value	String	<来自栈顶对象>	将要显示的值

# 读取 ObjectStack 里的对象的属性

- 若想访问 Object Stack 里的某个对象的属性. 可以使用以下几种形式之一:

```
object.propertyName
```

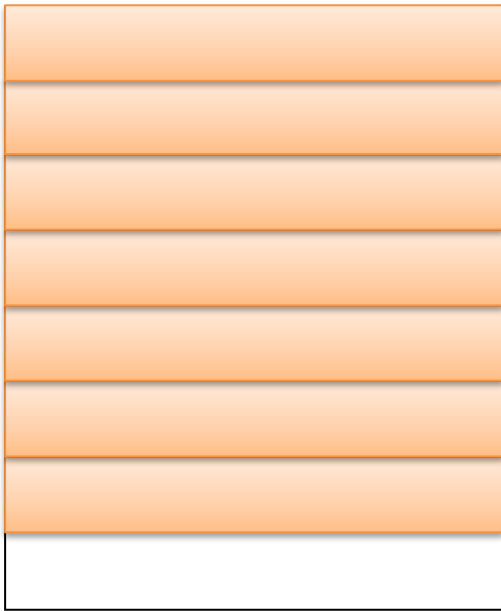
```
object['propertyName' ]
```

```
object["propertyName" ]
```

- **ObjectStack 里的对象可以通过一个从零开始的下标来引用.**  
ObjectStack 里的栈顶对象可以用 [0] 来引用, 它下面的那个对象可以用 [1] 引用. 若希望返回栈顶对象的 message 属性值: [0].message 或 [0][ "message" ] 或 [0][ 'message' ]
- **若在指定的对象里没有找到指定的属性, 则到指定对象的下一个对象里继续搜索. 即 [n] 的含义是从第 n 个开始搜索, 而不是只搜索第 n 个对象**
- **若从栈顶对象开始搜索, 则可以省略下标部分**

productName

<s:property value="[1].name">



# 读取 Context Map 里的对象的属性

- 若想访问 ContextMap 里的某个对象的属性, 可以使用以下几种形式之一:

```
#object.propertyName
```

```
#object['propertyName' ]
```

```
#object["propertyName" ]
```

- 示例:
  - 返回 session 中的 code 属性: #session.code
  - 返回 request 中的 customer 属性的 name 属性值:  
#request.customer.name
  - 返回域对象(按 request, session, application 的顺序)的 lastAccessDate 属性: #attr.lastAccessDate

# 调用字段和方法

- 可以利用 OGNL 调用
  - 任何一个 Java 类里的静态字段或方法.
  - 被压入到 ValueStack 栈的对象上的公共字段和方法.
- 默认情况下, Struts2 不允许调用任意 Java 类静态方法, 需要重新设置 **struts.ognl.allowStaticMethodAccess** 标记变量的值为 true.
- 调用静态字段或方法需要使用如下所示的语法:
  - **@fullyQualifiedClassName@fieldName:**  
@java.util.Calendar@DECEMBER
  - **@fullyQualifiedClassName@methodName(argumentList):**  
@app4.Util@now()
- 调用一个实例字段或方法的语法, 其中 object 是 Object Stack 栈里的某个对象的引用:
  - **.object.fieldName:** [0].datePattern
  - **.object.methodName(argumentList):** [0].repeat(3, "Hello" );

# 访问数组类型的属性

- 有些属性将返回一个对象数组而不是单个对象, 可以像读取任何其他对象属性那样读取它们. 这种**数组型属性的各个元素以逗号分隔, 并且不带方括号**
- 可以使用**下标**访问数组中指定的元素: `colors[0]`
- 可以通过调用其 `length` 字段查出给定数组中有多少个元素:  
**`colors.length`**

# 访问 List 类型的属性

- 有些属性将返回的类型是 `java.util.List`, 可以像读取任何其他属性那样读取它们. 这种 List 的各个元素是字符串, **以逗号分隔, 并且带方括号**
- 可以使用下标访问 List 中指定的元素: `colors[0]`
- 可以通过调用其 `size` 方法或专用关键字 `size` 的方法查出给定 List 的长度: **`colors.size` 或 `colors.size()`**
- 可以通过使用 `isEmpty()` 方法或专用关键字 `isEmpty` 来得知给定的 List 是不是空. **`colors.isEmpty` 或 `colors.isEmpty()`**
- 还可以使用 OGNL 表达式来**创建 List**, 创建一个 List 与声明一个 Java 数组是相同的: **`{ "Red" , "Black" , "Green" }`**

# 访问 Map 类型的属性

- 读取一个 Map 类型的属性将以如下所示的格式返回它所有的**键值对**:

```
{key-1=value-1, key-2=value-2, ... , key-n=value-n}
```

- 若希望检索出某个 Map 的值, 需要使用如下格式: **map[key]**
- 可以使用 size 或 size() 得出某个给定的 Map 的键值对的个数
- 可以使用 isEmpty 或 isEmpty() 检查某给定 Map 是不是空.
- 可以使用如下语法来创建一个 Map:

```
#{ key-1:value-1, key-2:value-2, ... key-n:value-n }
```



# 使用 EL 访问值栈中对象的属性

- `<s:property value= "fieldName" >` 也**可以通过 JSP EL 来达到目的**: `${fieldName}`
- 原理: Struts2 将包装 `HttpServletRequest` 对象后的 `org.apache.struts2.dispatcher.StrutsRequestWrapper` 对象传到页面上, 而这个类重写了 `getAttribute()` 方法.

# 异常处理: exception-mapping 元素

- exception-mapping 元素: 配置当前 action 的**声明式异常处理**
- exception-mapping 元素中有 2 个属性
  - exception: 指定需要捕获的异常类型。异常的全类名
  - result: 指定一个响应结果, 该结果将在捕获到指定异常时被执行, 既可以来自当前 action 的声明, 也可以来自 **global-results** 声明。
- 可以通过 **global-exception-mappings** 元素为应用程序提供一个全局性的异常捕获映射. 但在 global-exception-mappings 元素下声明的任何 exception-mapping 元素只能引用在 **global-results** 元素下声明的某个 result 元素
- 声明式异常处理机制由 ExceptionMappingInterceptor 拦截器负责处理, 当某个 exception-mapping 元素声明的异常被捕获到时, ExceptionMappingInterceptor 拦截器就会向 ValueStack 中添加**两个对象**:
  - **exception**: 表示被捕获异常的 Exception 对象
  - **exceptionStack**: 包含着被捕获异常的栈可以在视图上通过 `<s:property>` 标签显示异常消息

# 通用标签

讲师：佟刚

新浪微博：@尚硅谷-佟刚

# \*property 标签

- property 标签用来输出一个值栈属性的值

名字	类型	默认值	说明
default	String		可选, 如果 value 值为 null 或没有给定, 将显示该属性值
escape	boolean	true	可选, 是否要对HTML特殊字符进行转义
value	String	<来自栈顶对象>	将要显示的值

- 示例:

- 输出 Action 属性 customerId 的值: `<s:property value= "customerId" />`
- 输出 session 属性 userName 的值:  
`<s:property value= "#session.userName" />`

- 如果 value 属性没有给出, ValueStack 值栈栈顶对象的值被输出**
- 在许多情况下, JSP EL 可以提供更简洁的语法**

- url 标签用来动态地创建一个 URL

名字	类型	默认值	说明
• action	String		可选, 指定生成的 url 为哪个 action
anchor	String		可选, 指定被创建的 url 的连接锚点
encode	Boolean	true	可选, 是否要对参数进行编码
escapeAmp	Boolean	true	可选, 是否要对“&”字符进行转义
includeContext	Boolean	true	可选, 是否要把当前的上下文包括进来
• includeParams	String	get	可选, 指定是否包含请求参数, 可以取 3 个值之一: one、get、all
• method	String		可选, 指定 action 的方法. 当用 action 属性来生成 url 时, 如果指定该属性, url 将链接到指定的 action 的方法上
• namespace	String		可选, 指定 url 的命名空间
portletMode	String		可选, 指定结果页面的 portlet 模式
portletUrlType	String		可选, 指定将被创建的 URL 是一个 portlet 例程, 还是 action URL
scheme	String		可选, 指定使用什么协议: http, https
• value	String		可选, 指定将生成的 url 值 (如果新建 URL 的不是一个 action 的话)
• var	String		可选, 指定用来被压入 ContextMap 中的键值
windowState	String		可选, 当用在一个 portlet 环境里时, 用来指定 portlet 的窗口状态

# \*param 标签

- param 标签用来把一个参数传递给包含着它的那个标签

名字	类型	默认值	说明
name	String		将传递给外层标签的参数的名字
value	String		将传递给外层标签的参数的值

- 无论在给出 value 值时有没有使用 %{}, **Struts 都会对它进行 ognl 求值**
- 如果想传递一个 String 类型的字符串作为参数值, 必须把它用**单引号**括起来.
- 可以把 value 属性的值写在开始标签和结束标签之间. **利用这种方式来传递一个 EL 表达式的值**

# \*set 标签

- set 标签用来在以下 Map 对象里创建一个键值对：
  - ValueStack 值栈的 ContextMap 值栈
  - Map 类型的 session 对象
  - Map 类型的 application 对象
  - Map 类型的 request 对象
  - Map 类型的 page 对象

名字	类型	默认值	说明
• name	String		将被创建的属性的键
• value	String		该键所引用的对象
• scope	String	default	目标变量的作用范围。可取值是application、session、request、page和default

# \*push 标签

- push 标签的功能和 set 标签类似.
- push 标签将把一个对象**压入 ValueStack** 而不是压入 ContextMap.
- push 标签**在标签起始时把一个对象压入栈, 标签结束时将对对象弹出栈.**

名字	类型	默认值	说明
• Value*	String		将被压入Value Stack栈的值



# \*if, else 和 elseif 标签

- 这三个标签用来进行条件测试, 它们的用途和用法类似于 if, else 和 elseif 关键字. 其中 if 和 elseif 必须有 test 属性

名字	类型	默认值	说明
test*	Boolean		测试条件

# \*iterator 标签

- iterator 标签用来遍历一个数组, Collection 或一个 Map, **并把这个可遍历对象里的每一个元素依次压入和弹出 ValueStack 栈**

名字	类型	默认值	说明
• value	String		将被遍历的可遍历对象
• status	org.apache.struts2.views.jsp.IteratorStatus		
• var	String		用来引用这个可遍历对象中的当前元素的变量

- 在开始执行时, iterator 标签会先把 IteratorStatus 类的一个实例压入 ContextMap, 并在每次遍历循环时**更新它**. 可以将一个指向 IteratorStatus 对象的变量赋给 status 属性.

# \*iterator 标签

- iterator 标签的 status 属性的属性值

名字	类型	默认值	说明
index	integer		各次遍历的下标值（从零开始）
count	integer		当前遍历的下标值或“index+1”
first	boolean		如果当前元素是可遍历对象里的第一个元素，这个值将为true
last	boolean		如果当前元素是可遍历对象里的最后一个元素，这个值将为true
even	boolean		如果count属性的值是一个偶数，这个值将为true
odd	boolean		如果count属性的值是一个奇数，这个值将为true
modulus	int		这个属性需要一个输入参数，它的返回值是count属性值除以那个输入参数的余数

# \*sort 标签

- sort 标签用来对一个可遍历对象里的元素进行排序.

名字	类型	默认值	说明
• comparator*	java.util.Comparator		在排序过程中使用的比较器
• source	String		将对之进行排序的可遍历对象
• var	String		用来引用因排序而新生成的可遍历对象的变量

点击 price，使其可以按 price 进行排序，  
且可以升序，降序切换

Title	Author	Price	Remark

# \*date 标签

- date 标签用来对 Date 对象进行排版

名字	类型	默认值	说明
format	String		可选, 日期的格式
name*	String		将被排版的日期值
nice	boolean	false	可选, 指定是否输出指定日期和当前日期之间的时间差.
var	String		可选, 用来引用被压入 ValueStack 栈的日期值的变量

- format 属性的值必须是 java.text.SimpleDateFormat 类里定义的日期/时间格式之一.

## \*a 标签

- a 标签将呈现为一个 HTML 连接. 这个标签可以接受 HTML 语言中的 a 元素所能接受的所有属性.

# action 标签

- action 标签用在页面上来执行一个 action.
- action 标签还会把当前 Action 对象压入 ValueStack 值栈的 ContextMap 子栈.

名字	类型	默认值	说明
executeResult	boolean	false	是否执行/呈现该动作的结果
flush	boolean	true	是否要在动作结束后对“写”缓冲区进行“冲洗”
ignoreContextParams	boolean	false	是否要在触发该动作时把请求参数也包括进来
name*	String		将要触发的动作的名字，不需要“.action”后缀
namespace	boolean	与当前标签所使用的相同	将要触发的动作的命名空间
var	String		用来引用被压入Context Map栈的动作对象的变量



# bean 标签

- bean 标签将创建一个 JavaBean, 并把它压入 ValueStack 值栈的 ContextMap 子栈. 这个标签的功能与 JSP 中的 useBean 动作元素很相似

名字	类型	默认值	说明
name*	String		将创建的JavaBean的完全限定类名
var	String		用来引用被压入Context Map栈的JavaBean的变量

# include 标签

- include 标签用来把一个 Servlet 或 JSP 页面的输出包含到当前页面里来.

名字	类型	默认值	说明
Value*	String		将被包括的servlet/JSP页面的名字

# append, merge 标签

- append 标签用来合并可遍历对象.
- merge 标签用来交替合并可遍历对象.

名字	类型	默认值	说明
var	String		用来引用新合并而成的可遍历对象的变量

- 示例:

```
<s:merge id="cars">
  <s:param value="{americanCars}" />
  <s:param value="{europeanCars}" />
  <s:param value="{japaneseCars}" />
</s:merge>
<s:append id="cars2">
  <s:param value="americanCars"></s:param>
  <s:param value="europeanCars"></s:param>
  <s:param value="japaneseCars"></s:param>
</s:append>
mergeCars:
<ul>
<s:iterator value="{#cars}">
  <li><s:property/></li>
</s:iterator>
</ul>
appendCars:
<ul>
<s:iterator value="{#cars2}">
  <li><s:property/></li>
</s:iterator>
</ul>
```

# generator 标签

- generator 标签用来生成一个可遍历对象并把它压入 ValueStack 栈.
- generator 标签结束标记将弹出遍历对象

名字	类型	默认值	说明
converter	Converter		用来把val属性传递来的String值转换为一个对象的转换器
count	Integer		可遍历对象最多能够容纳的元素个数
separator*	String		各元素之间的分隔符
val*	String		可遍历对象中的各个元素的来源
var	String		用来引用新生成的可遍历对象的变量

- 如果在一个 generator 标签里给出了 converter 属性, 新生成的可遍历对象里的每一个元素都会传递到该属性所指定的方法进行必要的转换.

# generator 标签

- 示例.

```
<s:generator val="'Honda,Toyota,Ford'"
    separator=", ">
    <ul>
    <s:iterator>
        <li><s:property/></li>
    </s:iterator>
    </ul>
</s:generator>

<s:generator id="cameras"
    count="3"
    val="%{'Canon,Nikon,Pentax,FujiFilm'}"
    separator=", ">

</s:generator>
<s:iterator value="#attr.cameras">
    <s:property/>
</s:iterator>
```

# subset 标签

- subset 标签用来创建一个可遍历集合的子集。
- subset 标签通过 decider 属性来创建一个可遍历集合的子集。

名字	类型	默认值	说明
count	Integer		新建子集里的元素个数
decider	Decider		SubsetUteratorFilter.Decider接口的一种实现，它决定着什么样的元素能成为新建子集里的一员
source	String		用来充当新建子集的元素来源的可遍历对象
start	Integer		一个起始下标值，新建子集将从这个下标开始从source属性给定的那个可遍历对象里筛选各个元素
var	String		用来引用这个新建子集的变量

- 示例:

```
<s:generator id="computers"
    val="%{'HP,Dell,Asus,Fujitsu,Toshiba'}"
    separator=",">
</s:generator>

<s:subset source="#attr.computers" decider="myDecider">
    <s:iterator>
        <s:property/>
    </s:iterator>
</s:subset>
```

# 表单标签

讲师：佟刚

新浪微博：@尚硅谷-佟刚

# 概述

- 表单标签将在 HTML 文档里被呈现为一个表单元素
- 使用表单标签的优点:
  - **表单回显**
  - 对页面进行布局和排版
- 标签的属性可以被赋值为一个静态的值或**一个 OGNL 表达式**. 如果在赋值时使用了一个 OGNL 表达式并把它用 **%{} 括起来**, 这个表达式将会被求值.



# 表单标签的共同属性

名字	数据类型	说明
cssClass	String	用来呈现这个元素的CSS类
cssStyle	String	用来呈现这个元素的CSS样式
title	String	指定HTML title属性
disabled	String	指定HTML disabled属性
• label*	String	指定一个表单元素在xhtml和ajax主题里的行标
labelPosition	String	指定一个表单元素在xhtml和ajax主题里的行标位置。可取值是top和left（默认值）
key	String	这个输入字段所代表的属性的名字。它是name和label属性的一个快捷方式
requiredpositi	String	指定一个表单元素在xhtml和ajax主题里的标签必须出现的位置。可取值是left和right（默认值）
• name	String	指定HTML name属性。一个输入元素的name属性将被映射到一个 Action 属性
required*	boolean	在xhtml主题里，这个属性表明是否要给当前行标加上一个星号*
tabIndex	String	指定HTML tabIndex属性
• value	String	指定一个表单元素的值

- \* 该属性只在没有使用 simple 主题时才可以使用。

- form 标签用来呈现 HTML 语言中的表单元素

名字	数据类型	默认值	说明
acceptcharset	String		这个表单接受的字符集。如果有多个字符集，它们的名字要用逗号或空格隔开
action	String	当前action	提交这个表单将触发的action
enctype	String		表单的enctype属性
method	String	post	表单方法
namespace	String	当前命名空间	提交这个表单将触发的action所在的命名空间
onsubmit	String		Javascript onsubmit属性
openTemplate	String		用来打开这个表单的模板
portletMode	String		在用户提交这个表单后将显示的portlet模式
target	String		表单的target属性
validate	Boolean	false	在xhtml/ajax主题下，是否进行客户端输入验证
windowState	String		在用户提交这个表单后将显示的窗口状态

- 默认情况下, form 标签将被呈现为一个**表格**形式的 HTML 表单. 嵌套在 form 标签里的输入字段将被呈现为一个表格行. 每个表格行由两个字段组成, 一个对应着行标, 一个对应着输入元素. 提交按钮将被呈现为一个横跨两列单元格的行

# textfield, password, hidden 标签

- textfield 标签将被呈现为一个输入文本字段, password 标签将被呈现为一个口令字段, hidden 标签将被呈现为一个不可见字段.

名字	数据类型	默认值	说明
maxlength	integer		由这个标签呈现出来的HTML元素所能容纳的字符最大个数
readonly	boolean	false	用来表明该输入元素是不是只读的
size	integer		尺寸属性

- password 标签扩展自 textfield 标签, 多了一个 showPassword 属性. 该属性是布尔型. 默认值为 false, 它决定着在表单回显时是否显示输入的密码.

# submit 标签

- submit 标签将呈现为一个提交按钮. 根据其 type 属性的值. 这个标签可以提供 3 种呈现效果:
  - input: `<input type= "submim" .../>`
  - button: `<input type= "button" .../>`
  - image: `<input type= "image" />`

名字	数据类型	默认值	说明
action	String		HTML action属性
align	String		HTML align属性
method	String		method属性
type	String	input	这个属性的值决定着提交按钮的屏显效果类型, 它的可取值是input、button或image

# textarea 标签

- textarea 标签将呈现为一个 HTML 文本域元素

名字	数据类型	默认值	说明
cols	integer		HTML cols属性
readonly	boolean	false	用来表明该textarea元素是不是只读的
rows	integer		HTML rows属性
wrap	boolean		HTML wrap属性

# \*checkbox 标签

- checkbox 标签将呈现为一个 HTML 复选框元素. 该复选框元素通常用于提交一个布尔值
- 当包含着一个复选框的表单被提交时, 如果某个复选框被选中了, 它的值将为 **true**, 这个复选框在 HTTP 请求里增加一个请求参数. 但如果该复选框未被选中, 在请求中就不会增加一个请求参数.
- checkbox 标签解决了这个局限性, 它采取的办法是为单个复选框元素创建一个配对的不可见字段

```
<s:checkbox name="daily" label="Daily news alert"/>
```

```
<input type="checkbox" name="daily" value="true" id="CheckBox_daily"/>
```

```
<input type="hidden" id="__checkbox_CheckBox_daily" name="__checkbox_daily" value="true" />
```

Name:

Agree:

Submit

没有选中复选框, 提交表单

参数 application

name ABC

Action

```
String name = null;  
boolean agree = true;
```

# \*checkbox 标签

- checkbox 标签有一个 **fieldValue** 属性, 该属性指定的值将在用户提交表单时作为被选中的单选框的实际值发送到服务器. 如果没有使用 fieldValue 属性, 单选框的值将为 true 或 false.

名字	数据类型	默认值	说明
fieldvalue	String	true	该单选框的实际值

- 示例:

```
<s:iterator value="magazineList">

    <s:checkbox name="magazines"
        label="%{name}"
        fieldValue="%{code}"/>

</s:iterator>
```

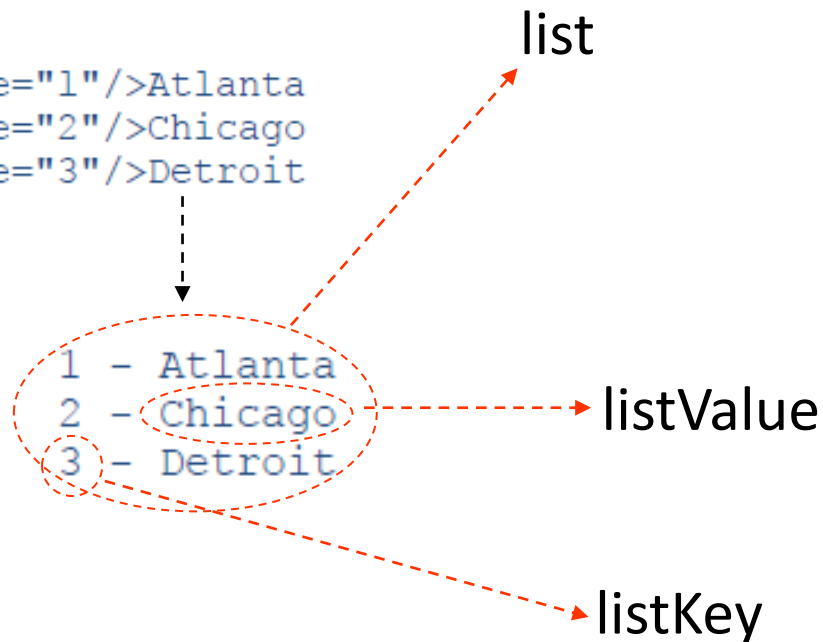


# list, listKey 和 listValue 属性

- list, listKey, listValue 这 3 个属性对 **radio, select, checkboxlist** 等标签非常重要

```
<input type="radio" name="city" value="1"/>Atlanta  
<input type="radio" name="city" value="2"/>Chicago  
<input type="radio" name="city" value="3"/>Detroit
```

Atlanta  Chicago  Detroit



- 可以把一个 String, 一个数组, 一个 Enumeration, Iterator, Map 或 Collection 赋给 list 属性.

# list, listKey 和 listValue 属性

- 示例:

```
<s:select list="{ 'Atlanta', 'Chicago', 'Detroit' }"/>
```



```
<select>  
  <option value="Atlanta">Atlanta</option>  
  <option value="Chicago">Chicago</option>  
  <option value="Detroit">Detroit</option>  
</select>
```

```
<s:select list="#{ '1': 'Atlanta', '2': 'Chicago', '3': 'Detroit' }"/>
```



```
<select>  
  <option value="1">Atlanta</option>  
  <option value="2">Chicago</option>  
  <option value="3">Detroit</option>  
</select>
```

# list, listKey 和 listValue 属性

- 赋值为一个Map:

```
Map<Integer, String> cities = new HashMap<Integer, String>();  
cities.put(1, "Atlanta");  
cities.put(2, "Chicago");  
cities.put(3, "Detroit");
```

```
<s:select list="cities"/>
```

- 赋值为一个 Collection(或一个对象数组): 把数组或 Collection 赋值给 list 属性, 把用来提供选项值的对象属性赋给 listKey 属性, 把用来提供选项行标的对象属性赋给 listValue 属性

```
<s:select list="cities" listKey="id" listValue="name" />
```

```
List<City> cities1 = new ArrayList<City>();  
cities1.add(new City(1, "AA"));  
cities1.add(new City(2, "BB"));  
cities1.add(new City(3, "CC"));  
cities1.add(new City(4, "DD"));  
  
requestMap.put("cities1", cities1);
```

City
- cityId : Integer
- cityName : String

```
<s:select name="city" label="Cities"  
  list="#request.cities1"  
  listKey="cityId"  
  listValue="cityName" >
```

```
</s:select>
```

↓

```
<select name="city">  
  <option value="1">AA</option>  
  <option value="2">BB</option>  
  <option value="3">CC</option>  
  <option value="4">DD</option>  
</select>
```



发送请求, Struts2 框架调用 execute() 方法

```
StrutsFormTagTestAction
  username : String
  getUsername() : String
  setUsername(String) : void
  age : Integer
  getAge() : Integer
  setAge(Integer) : void
  page : Page
  getPage() : Page
  execute() : String
```

Struts2 转发

```
public List<Person> getPersons() {
    persons = new ArrayList<Person>(5);
    persons.add(new Person(1, "AA", "AA"));
    persons.add(new Person(2, "BB", "BB"));
    persons.add(new Person(3, "CC", "CC"));
    persons.add(new Person(4, "DD", "DD"));
    persons.add(new Person(5, "EE", "EE"));

    return persons;
}
```

```
<s:checkboxlist label="Teacher" name="teacher"
```

```
list="page.persons" listKey="id" listValue="name"></s:checkboxlist>
```

```
<input type="checkbox" name="teacher" value="1" id="teacher-1" />
```

```
<input type="checkbox" name="teacher" value="2" id="teacher-2" />
```

```
<input type="checkbox" name="teacher" value="3" id="teacher-3" />
```

```
<input type="checkbox" name="teacher" value="4" id="teacher-4" />
```

```
<input type="checkbox" name="teacher" value="5" id="teacher-5" />
```

# radio 标签

- radio 标签将呈现为一组单选按钮, 单选按钮的个数与程序员通过该标签的 list 属性提供的选项的个数相同.
- 一般地, **使用 radio 标签实现 “多选一”, 对于 “真/假” 则该使用 checkbox 标签.**

名字	数据类型	默认值	说明
• list*	String		用来充当选项来源的可遍历对象
• listKey	String		用来提供选项值的对象属性
• listValue	String		用来提供选项行标的对象属性

- 示例:

```
<s:radio name="userType" label="User Type"
         list="#({'1':'Individual', '2':'Organization'})"
/>

<s:radio name="incomeLevel" label="Income Level"
         list="incomeLevels"
/>
<s:submit/>
```

# select 标签

- select 标签将呈现一个 select 元素.

名字	数据类型	默认值	说明
emptyOption	boolean	false	指明是否要在标题下面插入一个空白选项
headerKey	String		选项列表中的第一个选项的键
headerValue	String		选项列表中的第一个选项的值
list*	String		用来充当选项来源的可遍历对象
listKey	String		用来提供选项值的对象属性
listValue	String		用来提供选项行标的对象属性
multiple	boolean	false	指明是否允许多重选择（多选多）
size	integer		同时显示在页面里的选项的个数

- 示例:

```
<s:select name="country" label="Country"
          list="#application.countries"
          onchange="this.form.submit()"
/>

<s:select name="city" label="City"
          list="cities" listKey="id" listValue="name"
/>
```

# optgroup 标签

- optgroup 标签对 select 元素所提供的选项进行分组。每个选项有它自己的来源。

名字	数据类型	默认值	说明
list*	String		用来充当选项来源的可遍历对象
listKey	String		用来提供选项值的对象属性
listValue	String		用来提供选项行标的对象属性

- 示例:

```
<s:select name="city" label="City"
  list="usCities">
  <s:optgroup label="Canada" list="canadaCities"/>
  <s:optgroup label="Mexico" list="mexicoCities"/>
</s:select>
```



# checkboxlist 标签

- checkboxlist 标签将呈现一组多选框。

名字	数据类型	默认值	说明
list*	String		用来充当选项来源的可遍历对象
listKey	String		用来提供选项值的对象属性
listValue	String		用来提供选项行标的对象属性

- checkbox 标签被映射到一个字符串数组或是一个基本类型的数组. 若它提供的多选框一个也没有被选中, 相应的属性将被赋值为一个空数组而不是空值.**
- 示例:

```
<s:checkboxlist name="interests" label="Interests"  
    list="interestOptions"  
    listKey="id" listValue="description"  
>
```

兴趣标签： 新闻  娱乐  文化  体育  IT  
 财经  时尚  汽车  房产  生活

checkboxlist

同意[搜狐网络服务使用协议](#) [用户注册协议](#) · checkbox

- 默认情况下, form 标签将呈现为一个 HTML form 元素和一个 table 元素.

```
<s:form></s:form>
```



```
<form id="..." name="..." onsubmit="return true;" action="..."  
  method="post">  
<table class="wwFormTable">  
  
</table>  
</form>
```

- 每一种输入标签都将呈现为一个带标号的输入元素, 而这个输入元素将被包含在一个 tr 元素和 td 元素的内部

```
s:textfield label="My Label" -----> <tr>  
  <td class="tdLabel">  
    <label for="..." class="label">My Label:</label>  
  </td>  
  <td>  
    <input type="text" name="..." id="..." />  
  </td>  
</tr>
```

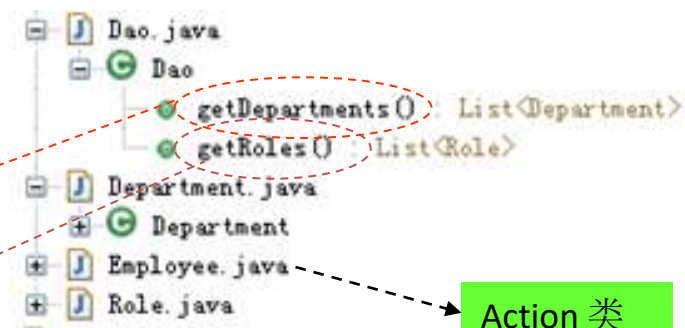
# 主题

- 主题: 为了让所有的 UI 标签能够产生同样的视觉效果而归集到一起的一组模板. 即**风格相近的模板被打包为一个主题**
  - **simple**: 把 UI 标签翻译成最简单的 HTML 对应元素, 而且会忽视行标属性
  - **xhtml**: xhtml 是**默认的主题**. 这个主题的模板通过使用一个布局表格提供了一种自动化的排版机制.
  - **css xhtml**: 这个主题里的模板与 xhtml 主题里的模板很相似, 但它们将使用 css 来进行布局 and 排版
  - **ajax**: 这个主题里的模板以 xhtml 主题里德模板为基础, 但增加了一些 Ajax 功能.
- 修改主题:
  - 通过 UI 标签的 theme 属性
  - 在一个表单里, 若没有给出某个 UI 标签的 theme 属性, 它将使用这个表单的主题
  - 在 page, request, session 或 application 中添加一个 theme 属性
  - 修改 struts.properties 文件中的 struts.ui.theme 属性.

# 示例代码

- 利用 Struts2 完成一个用户注册模块。

deptId  
deptName



http://localhost:8989/struts2--2/emp-input-ui

Name:

Password:

Gender:  Male  Female

Department:

Role:  Employee  Admin  Teacher

Submit

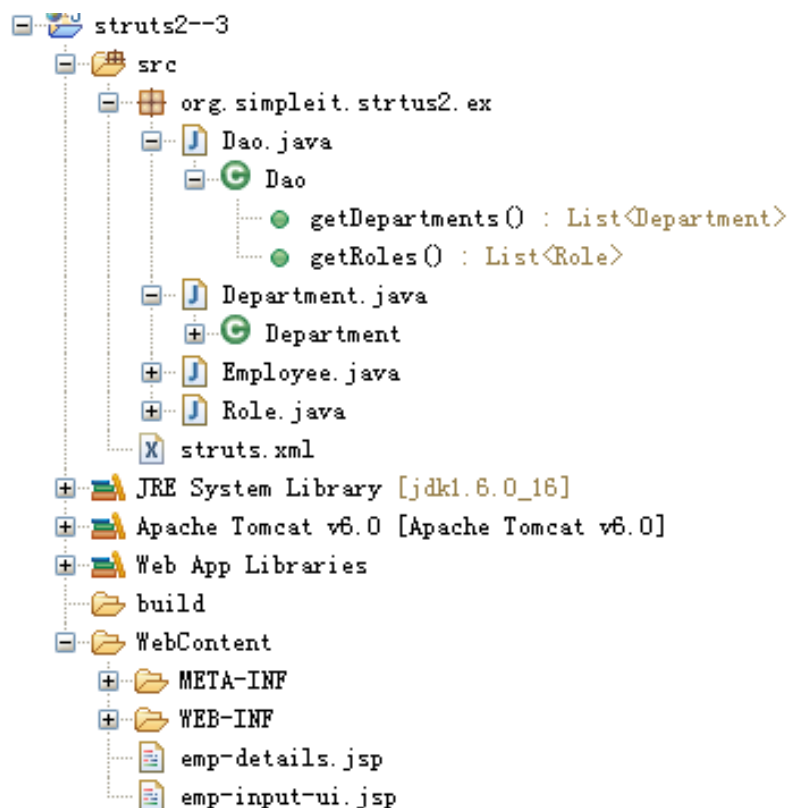
roleId  
roleName

http://localhost:8989/struts2--2/emp-input

Name: Tom  
Password: 123  
Gender:  
Dept: 2  
Role: 2, 3  
Description: I am Tom!

# 示例代码

- 目录结构: 页面上显示的 Department 和 Role 信息来自于 Dao 的 getXxx 方法



# ModelDriven 和 Preparable 拦截器

讲师：佟刚

新浪微博：@尚硅谷-佟刚

# 示例代码

- 完成员工的增, 删, 改, 查操作

`http://localhost:8989/struts2--5/Employee_list`

## Add New Employee

First Name:

Last Name:

Employee Id	First Name	Last Name	Edit Delete
2	Monica	Geller	<a href="#">Edit</a> <a href="#">Delete</a>
3	Phoebe	Buffay	<a href="#">Edit</a> <a href="#">Delete</a>
4	Joey	Tribbiani	<a href="#">Edit</a> <a href="#">Delete</a>
5	Chandler	Bing	<a href="#">Edit</a> <a href="#">Delete</a>
6	Ross	Geller	<a href="#">Edit</a> <a href="#">Delete</a>

## Add New Employee

First Name:

Last Name:

Employee Id	First Name	Last Name	Edit Delete
3	Phoebe	Buffay	<a href="#">Edit</a> <a href="#">Delete</a>
4	Joey	Tribbiani	<a href="#">Edit</a> <a href="#">Delete</a>
5	Chandler	Bing	<a href="#">Edit</a> <a href="#">Delete</a>
6	Ross	Geller	<a href="#">Edit</a> <a href="#">Delete</a>

`<a href="Employee_delete.action?employeeId=2">Delete</a>`



action="/struts2--5/Employee\_create"

## Add New Employee

First Name:

Last Name:

Employee Id	First Name	Last Name	Edit	Delete
3	Phoebe	Buffay	<a href="#">Edit</a>	<a href="#">Delete</a>
4	Joey	Tribbiani	<a href="#">Edit</a>	<a href="#">Delete</a>
5	Chandler	Bing	<a href="#">Edit</a>	<a href="#">Delete</a>
6	Ross	Geller	<a href="#">Edit</a>	<a href="#">Delete</a>

`<a href="Employee_edit.action?employeeId=2">Edit</a>`

## Edit Employee

First Name:

Last Name:

/struts2--5/Employee\_update.action

## Add New Employee

First Name:

Last Name:

Employee Id	First Name	Last Name	Edit	Delete
3	Phoebe	Buffay	<a href="#">Edit</a>	<a href="#">Delete</a>
4	Joey	Tribbiani	<a href="#">Edit</a>	<a href="#">Delete</a>
5	Chandler	Bing	<a href="#">Edit</a>	<a href="#">Delete</a>
6	Ross	Geller	<a href="#">Edit</a>	<a href="#">Delete</a>
9	Simple	it	<a href="#">Edit</a>	<a href="#">Delete</a>

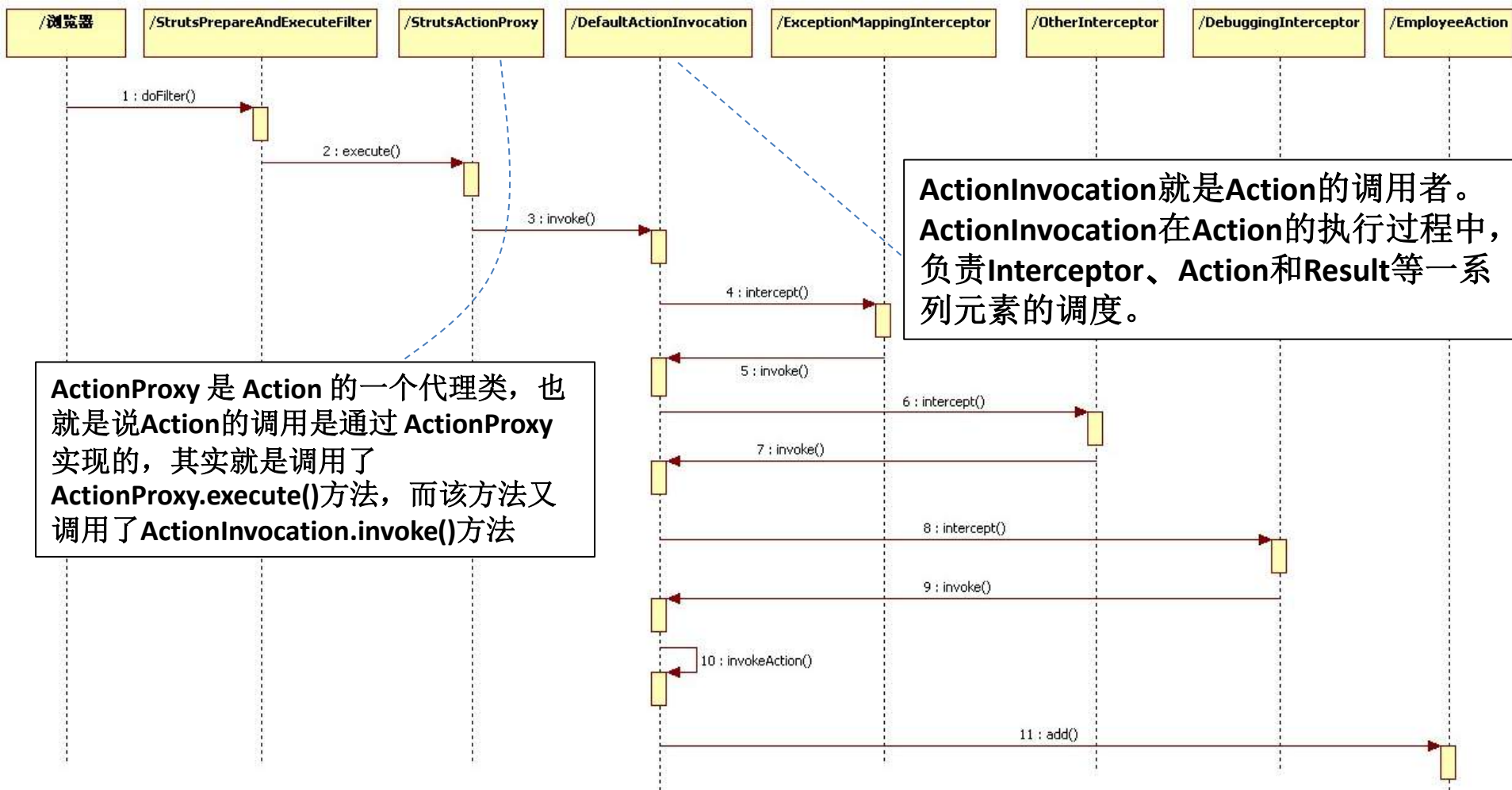
## Add New Employee

First Name:

Last Name:

Employee Id	First Name	Last Name	Edit	Delete
3	Phoebe	Buffay	<a href="#">Edit</a>	<a href="#">Delete</a>
4	Joey	Tribbiani	<a href="#">Edit</a>	<a href="#">Delete</a>
5	Chandler	Bing	<a href="#">Edit</a>	<a href="#">Delete</a>
6	Ross**	Geller**	<a href="#">Edit</a>	<a href="#">Delete</a>

# Struts2 运行流程图-1



# Params 拦截器

- **Parameters 拦截器**将把表单字段映射到 **ValueStack 栈的栈顶对象的各个属性中**. 如果某个字段在模型里没有匹配的屬性, Param 拦截器将尝试 ValueStack 栈中的下一个对象

```
<interceptor-stack name="defaultStack">
  <interceptor-ref name="exception"/>
  <interceptor-ref name="alias"/>
  <interceptor-ref name="servletConfig"/>
  <interceptor-ref name="i18n"/>
  <interceptor-ref name="prepare"/>
  <interceptor-ref name="chain"/>
  <interceptor-ref name="debugging"/>
  <interceptor-ref name="scopedModelDriven"/>
  <interceptor-ref name="modelDriven"/>
  <interceptor-ref name="fileUpload"/>
  <interceptor-ref name="checkbox"/>
  <interceptor-ref name="multiselect"/>
  <interceptor-ref name="staticParams"/>
  <interceptor-ref name="actionMappingParams"/>
  <interceptor-ref name="params">
```

把表单的值赋给栈顶对象的属性  
此时栈顶对象即为 Action

# 把 Action 和 Model 分开

- 在使用 Struts 作为前端的企业级应用程序时把 Action 和 Model 清晰地隔离开是有必要的: 有些 Action 类不代表任何 Model 对象, 它们的功能仅限于提供显示服务

```
<interceptor-stack name="defaultStack">
  <interceptor-ref name="exception"/>
  <interceptor-ref name="alias"/>
  <interceptor-ref name="servletConfig"/>
  <interceptor-ref name="i18n"/>
  <interceptor-ref name="prepare"/>
  <interceptor-ref name="chain"/>
  <interceptor-ref name="debugging"/>
  <interceptor-ref name="scopedModelDriven"/>
  <interceptor-ref name="modelDriven"/>
  <interceptor-ref name="fileUpload"/>
  <interceptor-ref name="checkbox"/>
  <interceptor-ref name="multiselect"/>
  <interceptor-ref name="staticParams"/>
  <interceptor-ref name="actionMappingParams"/>
  <interceptor-ref name="params"/>
</interceptor-stack>
```

如果 Action 类实现了 ModelDriven 接口，该拦截器将把 ModelDriven 接口的 getModel() 方法返回的对象置于栈顶

把表单的值赋给栈顶对象的属性

# ModelDriven 拦截器

- 当用户触发 add 请求时, ModelDriven 拦截器将调用 EmployeeAction 对象的 getModel() 方法, 并把返回的模型 (Employee实例)压入到 ValueStack 栈.
- 接下来 Parameters 拦截器将把表单字段映射到 ValueStack 栈的栈顶对象的各个属性中. **因为此时 ValueStack 栈的栈顶元素是刚被压入的模型(Employee)对象, 所以该模型将被填充.** 如果某个字段在模型里没有匹配的属性, Param 拦截器将尝试 ValueStack 栈中的下一个对象

EmployeeAction  
email

Employee  
firstName  
lastName

EmployeeAction  
email

```
<interceptor-stack name="defaultStack">
  <interceptor-ref name="exception"/>
  <interceptor-ref name="alias"/>
  <interceptor-ref name="servletConfig"/>
  <interceptor-ref name="i18n"/>
  <interceptor-ref name="prepare"/>
  <interceptor-ref name="chain"/>
  <interceptor-ref name="debugging"/>
  <interceptor-ref name="scopedModelDriven"/>
  <interceptor-ref name="modelDriven"/>
  <interceptor-ref name="fileUpload"/>
  <interceptor-ref name="checkbox"/>
  <interceptor-ref name="multiselect"/>
  <interceptor-ref name="staticParams"/>
  <interceptor-ref name="actionMappingParams"/>
  <interceptor-ref name="params"/>
</interceptor-stack>
```

```
<s:form action="/Employee create">
  <s:textfield name="firstName" label="FirstName"/>
  <s:textfield name="lastName" label="LastName"/>
  <s:textfield name="email" label="Email"/>
</s:form>
```





`<a href="Employee_edit?employeeId=1343900095706" >Edit</a>`

```
public Employee getModel() {
    employee = new Employee();
    return employee;
}
```

```
public void setEmployeeId(String employeeId) {
    this.employeeId = employeeId;
}
```

```
public String edit(){
    //1. 从数据库中获取 employeeId 对应的对象
    Employee emp = dao.get(employee.getEmployeeId());

    //2. 把数据库中获取的属性放入值栈属性中
    employee.setEmployeeId(emp.getEmployeeId());
    employee.setFirstName(emp.getFirstName());
    employee.setLastName(emp.getLastName());

    return "edit-sucess";
}
```

```
<interceptor-stack name="defaultStack">
    <interceptor-ref name="exception"/>
    <interceptor-ref name="alias"/>
    <interceptor-ref name="servletConfig"/>
    <interceptor-ref name="i18n"/>
    <interceptor-ref name="prepare"/>
    <interceptor-ref name="chain"/>
    <interceptor-ref name="debugging"/>
    <interceptor-ref name="scopedModelDriven"/>
    <interceptor-ref name="modelDriven"/>
    <interceptor-ref name="fileUpload"/>
    <interceptor-ref name="checkbox"/>
    <interceptor-ref name="multiselect"/>
    <interceptor-ref name="staticParams"/>
    <interceptor-ref name="actionMappingParams"/>
    <interceptor-ref name="params"/>
</interceptor-stack>
```

把 employee 对象置于栈顶

此时栈顶对象为 employee, 所以把请求参数赋给 employee 的对应属性

把栈顶对象的属性在表单中进行回显

```
<s:form action="Employee_update">
    <s:hidden name="employeeId"></s:hidden>
    <s:textfield name="firstName" label="Fir
    <s:textfield name="lastName" label="Last

    <s:submit></s:submit>
</s:form>
```



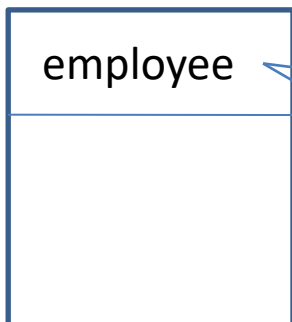
```
public String edit(){
```

```
    employee =
```

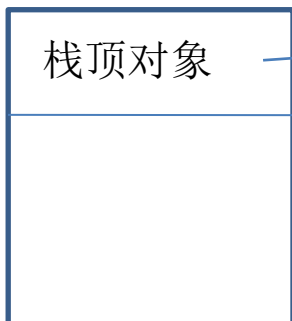
```
        employeeDao.get(employee.getEmployeeId());
```

```
    return "edit";
```

```
}
```



EmployeeAction

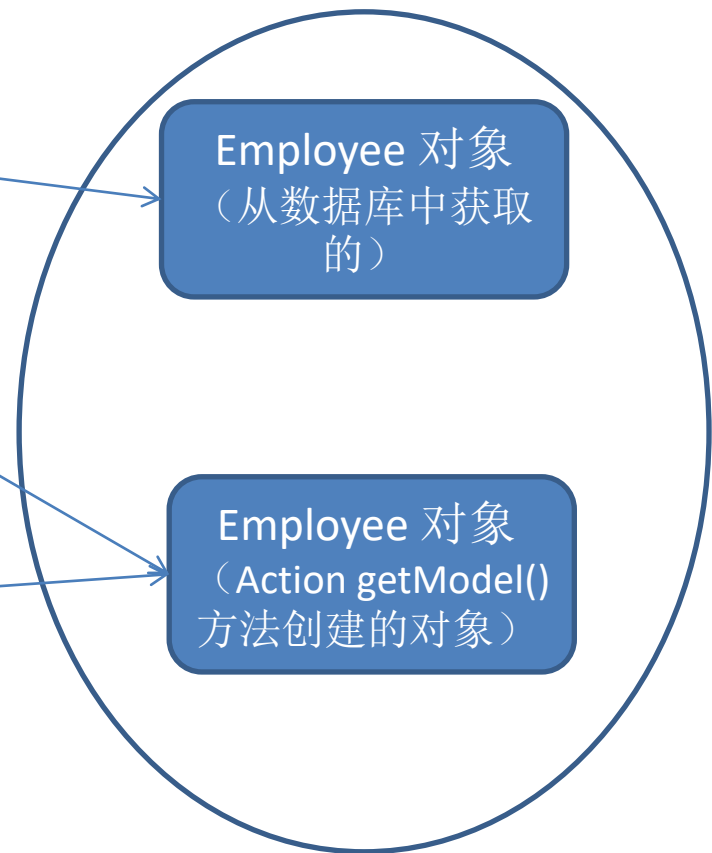


值栈

```
@Override
```

```
public Employee getModel() {  
    employee = new Employee();  
    return employee;
```

```
}
```



堆栈

`<a href="Employee_edit?employeeId=1343900095706">Edit</a>`

此时栈顶对象为 Action, 所以把请求参数赋给 Action 的对应属性

```
public void setEmployeeId(String employeeId) {  
    this.employeeId = employeeId;  
}
```

```
@Override  
public Employee getModel() {  
    //添加时: 没有 employeeId 这个请求参数  
    if(employeeId == null){  
        employee = new Employee();  
    }  
    //回显页面时: 有 employeeId 这个请求参数  
    else{  
        employee = employeeDao.get(employeeId);  
    }  
    return employee;  
}
```

```
<interceptor-stack name="paramsPrepareParamsStack">  
    <interceptor-ref name="exception"/>  
    <interceptor-ref name="alias"/>  
    <interceptor-ref name="i18n"/>  
    <interceptor-ref name="checkbox"/>  
    <interceptor-ref name="multiselect"/>  
    <interceptor-ref name="params">  
        <param name="excludeParams">dojo\..*,^struts</param>  
    </interceptor-ref>  
    <interceptor-ref name="servletConfig"/>  
    <interceptor-ref name="prepare"/>  
    <interceptor-ref name="chain"/>  
    <interceptor-ref name="modelDriven"/>  
    <interceptor-ref name="fileUpload"/>  
    <interceptor-ref name="staticParams"/>  
    <interceptor-ref name="actionMappingParams"/>  
    <interceptor-ref name="params">
```

# Preparable 拦截器

- Struts 2.0 中的 modelDriven 拦截器负责把 Action 类以外的一个对象压入到值栈栈顶
- 而 prepare 拦截器负责准备为 getModel() 方法准备 model

# PrepareInterceptor拦截器用方法

- 若 Action 实现 `Preparable` 接口，则 Action 方法需实现 `prepare()` 方法
- `PrepareInterceptor` 拦截器将调用 `prepare()` 方法，  
**`prepareActionMethodName()`**方法 或  
**`prepareDoActionMethodName()`**方法
- `PrepareInterceptor` 拦截器根据 **`firstCallPrepareDo`** 属性决定获取 `prepareActionMethodName`、`prepareDoActionMethodName`的顺序。默认情况下先获取 `prepareActionMethodName()`，如果没有该方法，就寻找`prepareDoActionMethodName()`。如果找到对应的方法就调用该方法
- `PrepareInterceptor` 拦截器会根据 **`alwaysInvokePrepare`** 属性决定是否执行`prepare()`方法

# 使用 *paramsPrepareParamsStack* 拦截器栈

- *paramsPrepareParamsStack* 从字面上理解来说，这个stack的拦截器调用的顺序为：首先 *params*，然后 *prepare*，接下来 *modelDriven*，最后再 *params*
- Struts 2.0的设计上要求 *modelDriven* 在 *params* 之前调用，而业务中 *prepare* 要负责准备 *model*，准备 *model* 又需要参数，这就需要在 *prepare* 之前运行 *params* 拦截器设置相关参数，这个也就是创建 *paramsPrepareParamsStack* 的原因。
- 流程如下：
  - 1. *params* 拦截器首先给 *action* 中的相关参数赋值，如 *id*
  - 2. *prepare* 拦截器执行 *prepare* 方法，*prepare* 方法中会根据参数，如 *id*，去调用业务逻辑，设置 *model* 对象
  - 3. *modelDriven* 拦截器将 *model* 对象压入 *value stack*，这里的 *model* 对象就是在 *prepare* 中创建的
  - 4. *params* 拦截器再将参数赋值给 *model* 对象
  - 5. *action* 的业务逻辑执行

```
<a href="Employee_edit?employeeId=1343900095706" >Edit</a>
```

此时栈顶对象为 Action, 所以把请求参数赋给 Action 的对应属性

```
public void setEmployeeId(String employeeId) {  
    this.employeeId = employeeId;  
}
```

调用 prepareEdit 方法为 ModelDriven 的 getModel() 方法准备 Model 对象

```
public void prepareEdit() {  
    emp = dao.get(employeeId);  
}
```

把 dao 中获取的 employee 对象置于栈顶

```
<interceptor-stack name="paramsPrepareParamsStack">  
    <interceptor-ref name="exception"/>  
    <interceptor-ref name="alias"/>  
    <interceptor-ref name="i18n"/>  
    <interceptor-ref name="checkbox"/>  
    <interceptor-ref name="multiselect"/>  
    <interceptor-ref name="params">  
        <param name="excludeParams">dojo\..*,^struts</param>  
    </interceptor-ref>  
    <interceptor-ref name="servletConfig"/>  
    <interceptor-ref name="prepare"/>  
    <interceptor-ref name="chain"/>  
    <interceptor-ref name="modelDriven"/>  
    <interceptor-ref name="fileUpload"/>  
    <interceptor-ref name="staticParams"/>  
    <interceptor-ref name="actionMappingParams"/>  
    <interceptor-ref name="params">
```

此时栈顶对象为 employee, 会再把 employeeId 赋给该对象的 employeeId 属性

# 类型转换

讲师：佟刚

新浪微博：@尚硅谷-佟刚



# 内容提要

- 类型转换概述
- 类型转换出错时如何处理
  - 转到哪个页面
  - 显示什么错误消息
- 自定义类型转换器
- 类型转换与复杂对象配合使用

# 概述

- 从一个 HTML 表单到一个 Action 对象, **类型转换是从字符串到非字符串.**
  - HTTP 没有 “类型” 的概念. 每一项表单输入只可能是一个字符串或一个字符串数组. 在服务器端, 必须把 String 转换为特定的数据类型
- 在 struts2 中, 把请求参数映射到 action 属性的工作由 **Parameters 拦截器**负责, 它是默认的 defaultStack 拦截器中的一员. Parameters 拦截器可以自动完成字符串和基本数据类型之间转换.

# 类型转换错误

- **如果类型转换失败:**

- **若 Action 类没有实现 ValidationAware 接口** : Struts 在遇到类型转换错误时仍会继续调用其 Action 方法, 就好像什么都没发生一样.
- **若 Action 类实现 ValidationAware 接口** : Struts 在遇到类型转换错误时将不会继续调用其 Action 方法: Struts 将检查相关 action 元素的声明是否包含着一个 name=input 的 result. 如果有, Struts 将把控制权转交给那个 result 元素; 若没有 input 结果, Struts 将抛出一个异常

# 类型转换错误消息的定制

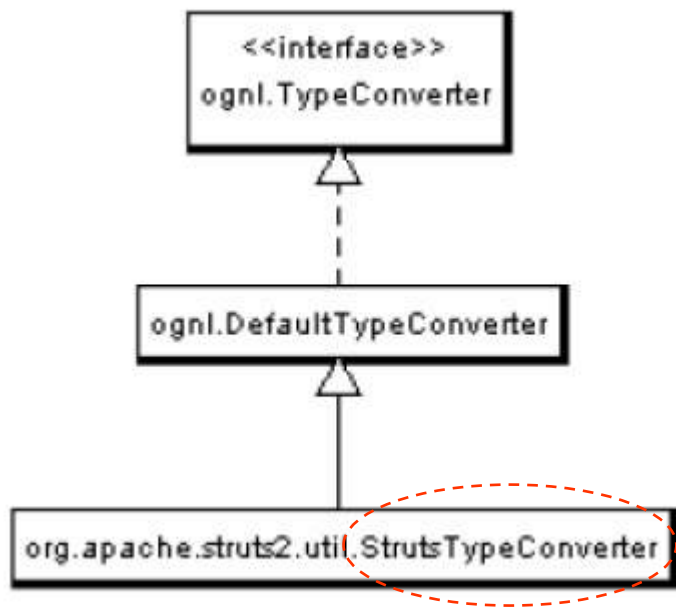
- 作为默认的 default 拦截器的一员, **ConversionError 拦截器**负责添加与类型转换有关的出错消息(前提: **Action 类必须实现了 ValidationAware 接口**)和保存各请求参数的原始值.
- 若字段标签使用的不是 simple 主题, 则非法输入字段将导致一条有着以下格式的出错消息:

`Invalid field value for field fieldName.`

- 覆盖默认的出错消息
  - 在对应的 **Action 类所在的包中新建 ActionClassName.properties 文件, ClassName 即为包含着输入字段的 Action 类的类名**
  - 在属性文件中添加如下键值对: `invalid.fieldvalue.fieldName=Custom error message`
- 定制出错消息的样式:
  - 每一条出错消息都被打包在一个 HTML span 元素里, 可以通过覆盖其行标为 errorMessage 的那个 css 样式来改变出错消息的格式.
- 显示错误消息: 如果是 simple 主题, 可以通过 `<s:fielderror fieldName= "filename" ></s:fielderror>` 标签显示错误消息

# 定制类型转换器

- 自定义类型转换器必须实现 `ognl.TypeConverter` 接口或对这个接口的某种具体实现做扩展



# 扩展 StrutsTypeConverter 类

- 在大多数类型转换器里, 需要提供从 String 类型到非 String 类型和与此相反的转换功能
- 在 StrutsTypeConverter 中有两个抽象方法:

abstract Object	<code>convertFromString(Map context, String[] values, Class toClass)</code> Converts one or more String values to the specified class.
--------------------	---

abstract String	<code>convertToString(Map context, Object o)</code> Converts the specified object to a String.
--------------------	---

# 配置自定义的类型转换器

- 在应用程序里使用一个自定义的类型转换器之前, 必须先对它进行配置. 这种配置**既可以基于字段, 也可以基于类型**
- 基于字段配置: 可以为**某个 Model (该 Model 类也可能是 Action)** 的各个属性分别配置一个自定义的转换器.
  - 1. 创建一个属性文件: ModelClassName-conversion.properties, 该文件需和相对应的 Model 类放在同一个目录下
  - 2. 编辑属性文件:

```
field1=customConverter1  
field2=customConverter2  
...
```

- 基于类型配置:
  - 在 WEB-INF/classes/ 目录下创建 xwork-conversion.properties 文件.
  - 在 xwork-conversion.properties 文件里把每一个**需要进行类型转换的类** 与一个类型转换器关联起来

```
fullyQualifiedClassName=CustomConverter1  
...
```

# 示例代码

- 实现自定义的时间类型转换器: 时间 pattern 需要以 web 应用的初始化参数配置在 web.xml 中

```
<context-param>  
    <param-name>datePattern</param-name>  
    <param-value>yyyy/MM/dd</param-value>  
</context-param>
```

- 若类型转换失败，给出自定义的信息。



EmpName:

Age:

Address:

Birth:

LiaoNing, DaLian, ShaHeKou

Submit

```
class Address{  
  
    private String province;  
    private String city;  
    private String qu;  
  
    //...  
}
```

# 类型转换与复杂属性配合使用

- form 标签的 name 属性可以被映射到一个属性的属性。

```
<s:form action="/complex-test">  
  <s:textfield name="name" label="DeptName"></s:textfield>  
  <s:textfield name="manager.name" label="ManagerName"></s:textfield>  
  <s:textfield name="manager.birth" label="ManagerBirth" value="%{birth}"></s:textfield>  
  
  <s:submit value="Submit"></s:submit>  
</s:form>
```

```
public class Department extends ActionSupport{  
  
  private static final long serialVersionUID  
  
  private Integer id;  
  private String name;  
  private Manager manager;
```

```
public class Manager{  
  
  private String name;  
  private Date birth;
```

- Struts 还允许填充 Collection 里的对象, 这常见于需要快速录入批量数据的场合

```
<tr>
  <td class="tdLabel"><label for="collection-test_emps_1_name" class="label">ManagerName1:</label></td>
  <td><input type="text" name="emps[0].name" value="" id="collection-test_emps_0_name"/></td>
</tr>

<tr>
  <td class="tdLabel"><label for="collection-test_emps_1_birth" class="label">ManagerBirth1:</label></td>
  <td><input type="text" name="emps[0].birth" value="" id="collection-test_emps_0_birth"/></td>
</tr>

<tr>
  <td class="tdLabel"><label for="collection-test_emps_2_name" class="label">ManagerName2:</label></td>
  <td><input type="text" name="emps[1].name" value="" id="collection-test_emps_1_name"/></td>
</tr>

<tr>
  <td class="tdLabel"><label for="collection-test_emps_2_birth" class="label">ManagerBirth2:</label></td>
  <td><input type="text" name="emps[1].birth" value="" id="collection-test_emps_1_birth"/></td>
</tr>
```

↓

ManagerName1:

ManagerBirth1:

ManagerName2:

ManagerBirth2:

Submit

```
public class Department extends ActionSupport{

    private static final long serialVersionUID

    private Collection<Employee> emps;
```

# 消息处理与国际化

讲师：佟刚

新浪微博：@尚硅谷-佟刚

# 概述

- 在程序设计领域, 把在无需改写源代码即可让开发出来的应用程序能够支持多种语言和数据格式的技术称为国际化.
- 与国际化对应的是本地化, 指让一个具备国际化支持的应用程序支持某个特定的地区
- Struts2 国际化是建立在 Java 国际化基础上的 :
  - 为不同国家/语言提供对应的消息资源文件
  - Struts2 框架会根据请求中包含的 `<s:text name="username"></s:text>` Locale 加载对应的资源文件
  - 通过程序代码取得该资源文件中指定 key 对应的消息

i18n\_en\_US.properties

```
<s:text name="username"></s:text>
```

en\_US

```
1 username=UserName
```

i18n\_zh\_CN.properties

zh\_CN

```
1 username=\u7528\u6237\u540D
```

# 配置国际化资源文件

- Action 范围资源文件：在Action类文件所在的路径建立名为 **ActionName\_language\_country.properties** 的文件
- 包范围资源文件：在包的根路径下建立文件名为 **package\_language\_country.properties** 的属性文件，一旦建立，处于该包下的所有 Action 都可以访问该资源文件。注意：**包范围资源文件的 baseName 就是package**，不是Action所在的包名。
- **全局资源文件**
  - 命名方式: **basename\_language\_country.properties**
  - struts.xml  
`<constant name="struts.custom.i18n.resources" value="baseName"/>`
  - struts.properties  
`struts.custom.i18n.resources=baseName`
- 临时指定资源文件：**`<s:i18n.../>`** 标签的 name 属性指定临时的国际化资源文件

# 加载资源文件的顺序

- 假设我们在某个 ChildAction 中调用了 `getText("username")` :
  - (1)加载和 ChildAction 的类文件在同一个包下的系列资源文件 **ChildAction.properties**
  - (2)加载 ChildAction 实现的接口 IChild , 且和 IChildn 在同一个包下 **IChild.properties** 系列资源文件。
  - (3)加载 ChildAction 父类 Parent , 且和 Parent 在同一个包下的 baseName 为 **Parent.properties** 系列资源文件。
  - (4) 若 ChildAction 实现 ModelDriven 接口 , 则对于 `getModel()` 方法返回的 model 对象 , 重新执行第(1)步操作。
  - (5) 查找当前包下 **package.properties** 系列资源文件。
  - (6) 沿着当前包上溯 , 直到最顶层包来查找 **package.properties** 的系列资源文件。
  - (7) 查找 **struts.custom.i18n.resources** 常量指定 baseName 的系列资源文件。
  - (8) 直接输出该key的字符串值。

# 访问国际化消息

- JSP 页面访问国际化消息：
  - 不带占位符：
    - `<s:text name="key"/>`
    - 表单元素的 label 属性：**可替换为 key 或使用 getText() 方法，并对其进行强制 OGNL 解析**
  - 带占位符：
    - 在 `<s:text.../>` 标签中**使用多个 `<s:param.../>` 标签来填充消息中的占位符。**
    - Struts2 直接在国际化消息资源文件中通过 `"${}"` 使用表达式，该表达式将从**值栈中获取对应的属性值**



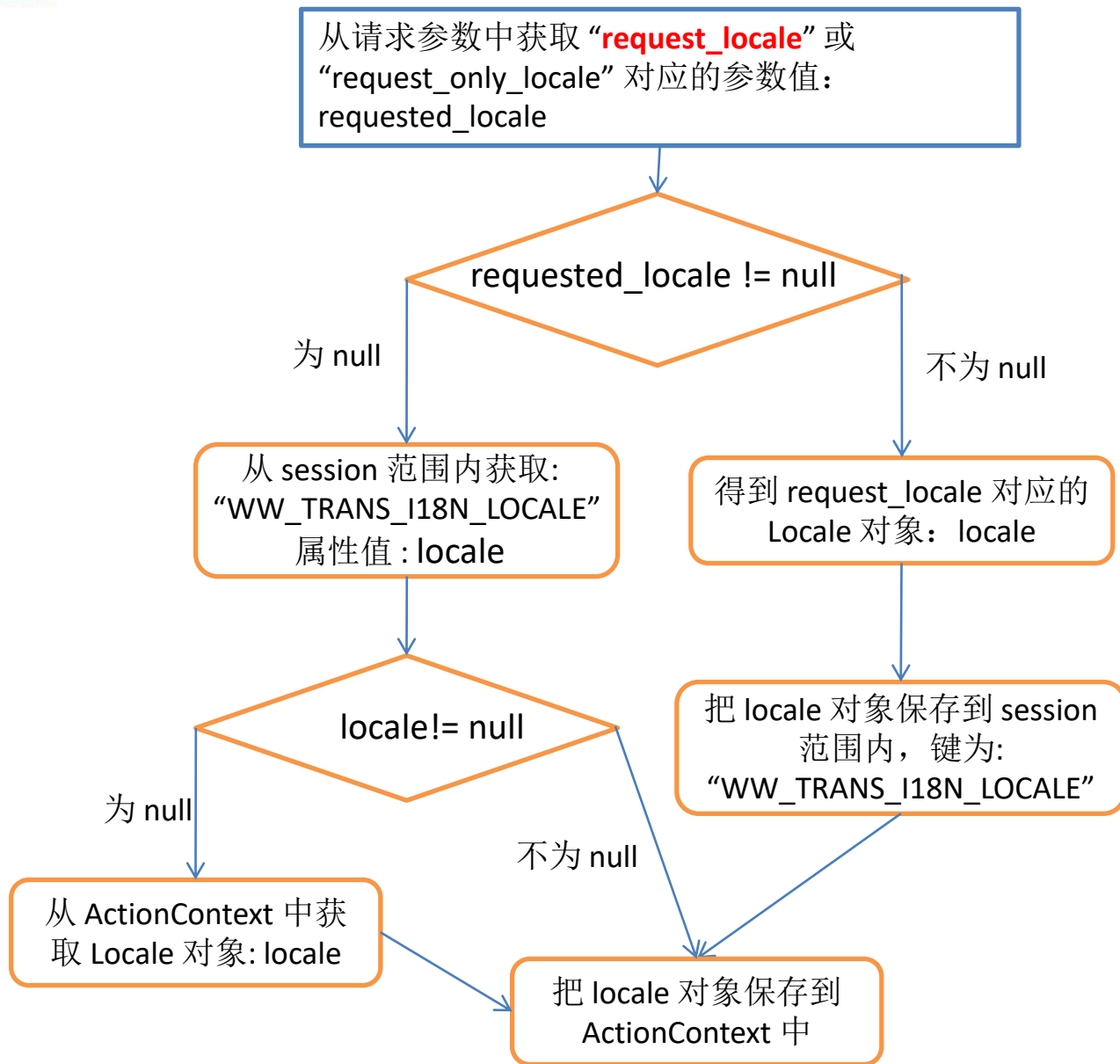
# 访问国际化消息

- Action 访问国际化消息：
  - 若 Action 类继承了 ActionSupport ，则可调用 TextProvider 接口的 getText 方法。

# 利用超链接实现动态加载国际化资源文件

- Struts2 使用 **i18n 拦截器** 处理国际化，并且将其注册在默认的拦截器中
- i18n拦截器在执行Action方法前，自动查找请求中一个名为 request\_locale 的参数。如果该参数存在，拦截器就将其作为参数，转换成Locale对象，并将其设为用户默认的Locale(代表国家/语言环境)。并把其设置为 session 的 WW\_TRANS\_I18N\_LOCALE 属性
- 若 request 没有名为request\_locale 的参数，则 i18n 拦截器会从 Session 中获取 WW\_TRANS\_I18N\_LOCALE 的属性值，若该值不为空，则将该属性值设置为浏览者的默认Locale
- 若 session 中的 WW\_TRANS\_I18N\_LOCALE 的属性值为空，则从 ActionContext 中获取 Locale 对象。

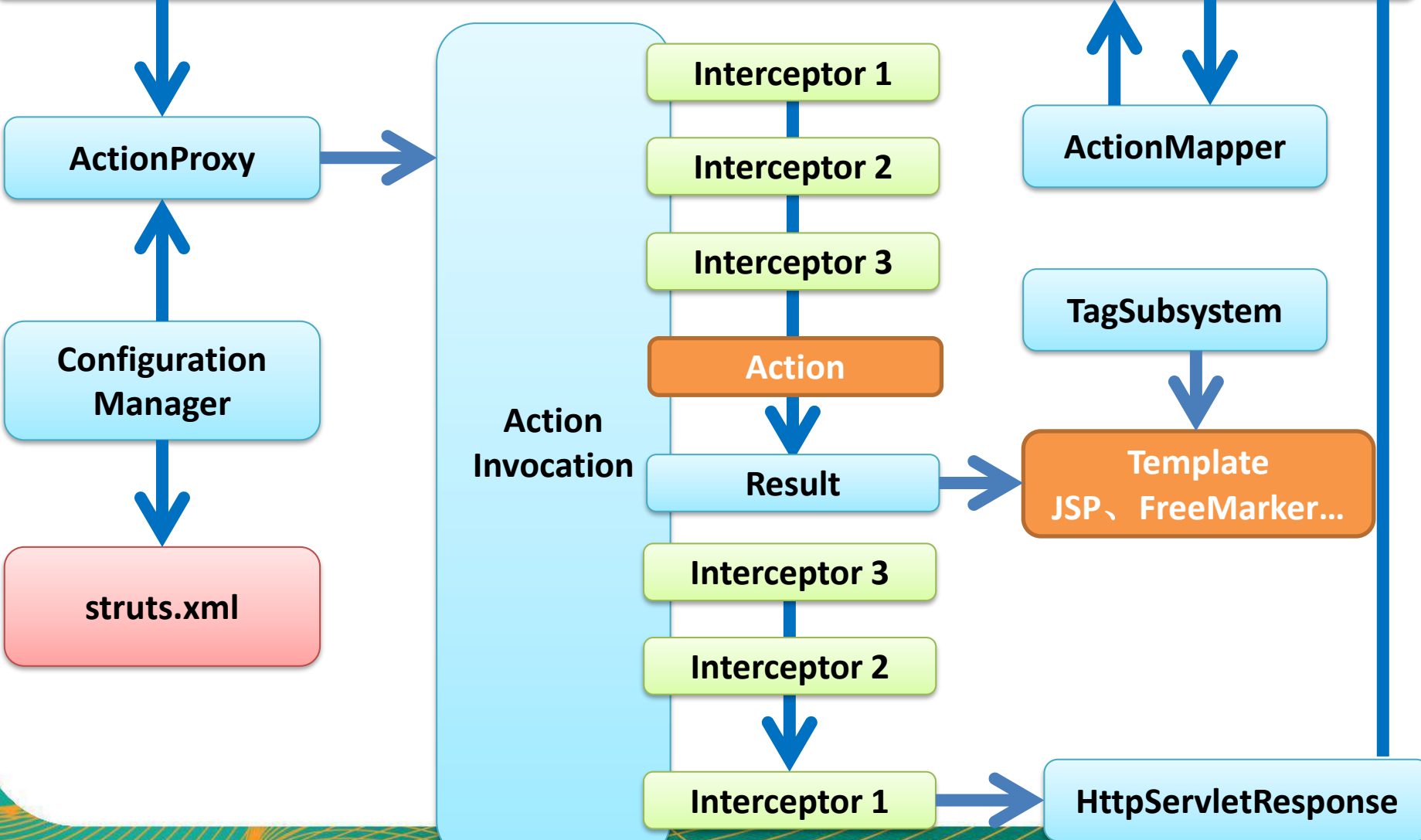
I18N  
拦截器确定 Locale 对象流程分析



# Struts2 运行流程分析

讲师：佟刚

新浪微博：@尚硅谷-佟刚



# 相关的几个 API

- **ActionMapping:** Simple class that holds the action mapping information used to invoke a Struts action. The name and namespace are required
- **ActionMapper:** When given an `HttpServletRequest`, the `ActionMapper` may return null if no action invocation request matches, or it may return an `ActionMapping` that describes an action invocation for the framework to try
- **ActionProxy:** `ActionProxy` is an extra layer between `XWork` and the action so that different proxies are possible.
- **ActionInvocation:** An `ActionInvocation` represents the execution state of an `Action`. It holds the `Interceptors` and the `Action` instance. By repeated re-entrant execution of the `invoke()` method, initially by the `ActionProxy`, then by the `Interceptors`, the `Interceptors` are all executed, and then the `Action` and the `Result`.

# Struts2 运行流程分析

- 1. 请求发送给 StrutsPrepareAndExecuteFilter
- 2. StrutsPrepareAndExecuteFilter 询问 ActionMapper：该请求是否是一个 Struts2 请求（即是否返回一个非空的 ActionMapping 对象）
- 3. 若 ActionMapper 认为该请求是一个 Struts2 请求，则 StrutsPrepareAndExecuteFilter 把请求的处理交给 ActionProxy
- 4. ActionProxy 通过 Configuration Manager 询问框架的配置文件，确定需要调用的 Action 类及 Action 方法
- 5. ActionProxy 创建一个 ActionInvocation 的实例，并进行初始化
- 6. ActionInvocation 实例在调用 Action 的过程前后，涉及到相关拦截器（Interceptor）的调用。
- 7. Action 执行完毕，ActionInvocation 负责根据 struts.xml 中的配置找到对应的返回结果。调用结果的 execute 方法，渲染结果。在渲染的过程中可以使用 Struts2 框架中的标签。
- 8. 执行各个拦截器 invocation.invoke() 之后的代码
- 9. 把结果发送到客户端

# 输入验证

讲师：佟刚

新浪微博：@尚硅谷-佟刚



# 概述

- 一个健壮的 web 应用程序必须确保用户输入是合法、有效的.
- Struts2 的输入验证
  - **基于 XWork Validation Framework 的声明式验证** : Struts2 提供了一些基于 XWork Validation Framework 的内建验证程序. 使用这些验证程序不需要编程, 只要在一个 XML 文件里对验证程序应该如何工作作出声明就可以了. 需要声明的内容包括:
    - **哪些字段需要进行验证**
    - **使用什么验证规则**
    - **在验证失败时应该把什么样的出错消息发送到浏览器端**
  - **编程验证** : 通过编写代码来验证用户输入

# 声明式验证

- 声明式验证程序可以分为两类：
  - **字段验证**: 判断某个字段属性的输入是否有效
  - **非字段验证**: 不只针对某个字段，而是针对多个字段的输入值之间的逻辑关系进行校验。例如：对再次输入密码的判断。
- 使用一个声明式验证程序需要 3 个步骤：
  - 1. **确定哪些 Action 字段需要验证**
  - 2. **编写一个验证程序配置文件**. 它的文件名必须是以下两种格式之一：
    - 若一个 Action 类的多个 action 使用同样的验证规则:  
**ActionClassName-validation.xml**
    - 若一个 Action 类的多个 action 使用不同的验证规则: **ActionClass-alias-validation.xml**, 例如 UserAction-**User\_create**-validation.xml
  - 3. **确定验证失败时的响应页面**: 在 struts.xml 文件中定义一个 `<result name= "input" >` 的元素。

# Struts2 内建的验证规则

- conversion validator : 转换验证器
- date validator : 日期验证器
- double validator : 浮点验证器
- email validator : email 验证器
- expression validator : 表达式验证器
- fieldexpression validator : 字段表达式验证器
- int validator : 整型验证器
- regex validator : 正则表达式验证器
- required validator : 非空验证器
- requiredstring validator : 非空字符串验证器
- stringlength validator : 字符串长度验证器
- url validator : url 格式验证器
- visitor validator : 复合属性验证器

# 验证程序的配置

待验证的字段名称

```
<field name="age">
```

验证规则

```
<field-validator type="int">
```

向验证程序传递参数

```
<param name="min">20</param>
```

```
<param name="max">50</param>
```

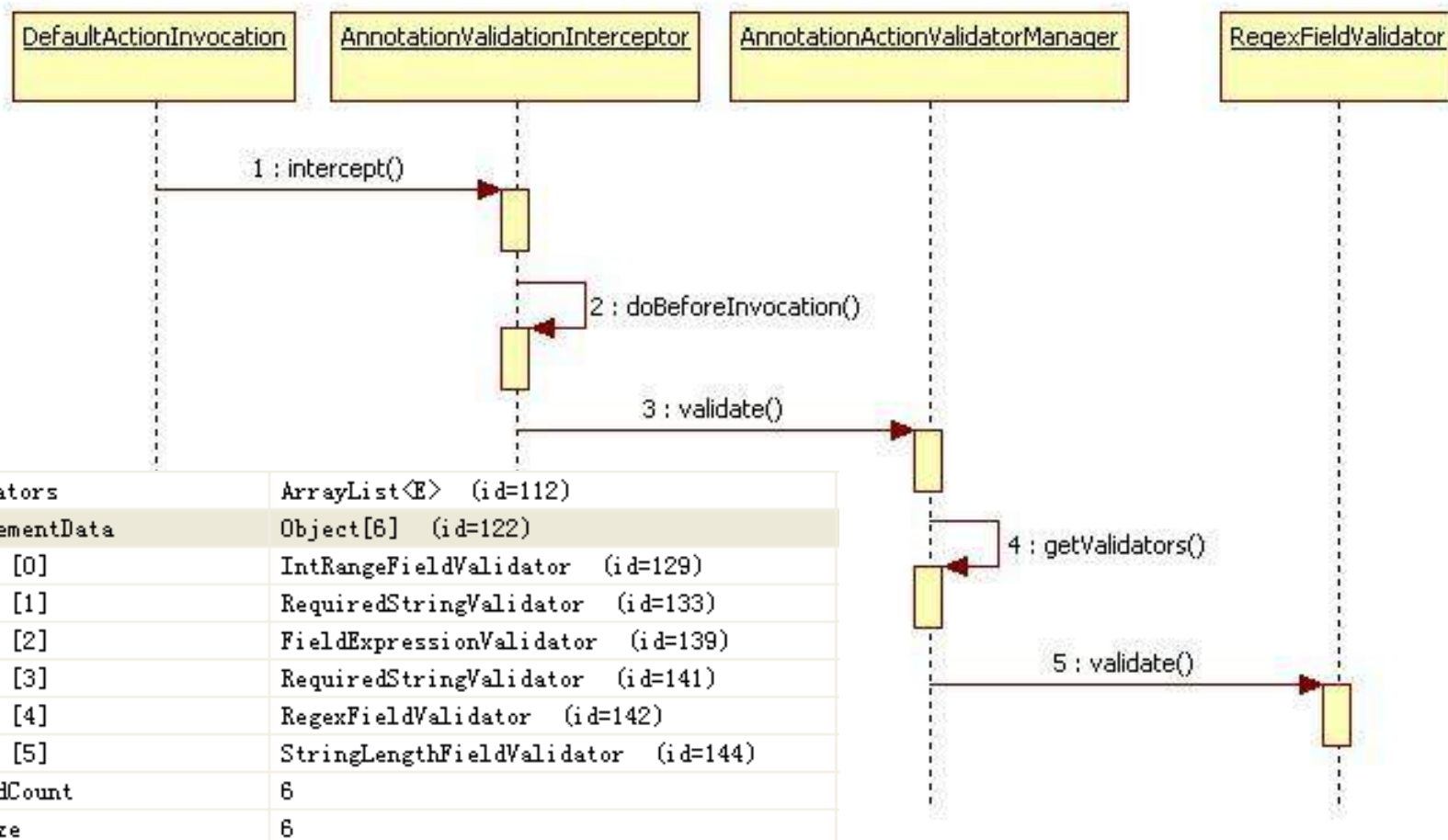
定义验证程序的出错消息

```
<message>Age needs to be between {min} and {max}</message>
```

```
</field-validator>
```

```
</field>
```

- **Struts2 的 Validation 拦截器**负责加载和执行已注册的验证程序，它是 defaultStack 拦截器的一员



# Struts2 的验证规则和验证器

- 每个验证规则都对应一个具体的验证器

```
<validators>
```

```
<validator name="required" class="com.opensymphony.xwork2.validator.validators.RequiredField
```

```
<validator name="requiredstring" class="com.opensymphony.xwork2.validator.validators.Require
```

```
<validator name="int" class="com.opensymphony.xwork2.validator.validators.IntRangeFieldValid
```

```
<validator name="long" class="com.opensymphony.xwork2.validator.validators.LongRangeFieldVal
```

```
<validator name="short" class="com.opensymphony.xwork2.validator.validators.ShortRangeFieldV
```

```
<validator name="double" class="com.opensymphony.xwork2.validator.validators.DoubleRangeFiel
```

```
<validator name="date" class="com.opensymphony.xwork2.validator.validators.DateRangeFieldVal
```

```
<validator name="expression" class="com.opensymphony.xwork2.validator.validators.ExpressionV
```

```
<validator name="fieldexpression" class="com.opensymphony.xwork2.validator.validators.FieldE
```

```
<validator name="email" class="com.opensymphony.xwork2.validator.validators.EmailValidator"/
```

```
<validator name="url" class="com.opensymphony.xwork2.validator.validators.URLValidator"/>
```

```
<validator name="visitor" class="com.opensymphony.xwork2.validator.validators.VisitorFieldVa
```

```
<validator name="conversion" class="com.opensymphony.xwork2.validator.validators.ConversionE
```

```
<validator name="stringlength" class="com.opensymphony.xwork2.validator.validators.StringLen
```

```
<validator name="regex" class="com.opensymphony.xwork2.validator.validators.RegexFieldValida
```

```
<validator name="conditionalvisitor" class="com.opensymphony.xwork2.validator.validators.Con
```

```
</validators>
```

# 配置文件与验证器属性

```
name="int" class="com.opensymphony.xwork2.validator.validators.IntRangeFieldValidator"/>
```

[-] ● this	IntRangeFieldValidator
[+] ◆ defaultMessage	"Age needs to be betwee
[+] ■ fieldName	"age" (id=110)
[+] ◆ log	JdkLogger (id=111)
[+] ▲ max	Integer (id=114)
◆ messageKey	null
■ messageParameters	null
[+] ▲ min	Integer (id=118)
■ parse	false
■ shortCircuit	false
[+] ■ stack	OgnlValueStack (id=119)
■ type	null
[+] ■ type	"int" (id=125)
[+] ■ validatorContext	DelegatingValidatorCont

```
<field name="age">
```

```
<field-validator type="int">
```

```
<param name="min">20</par
```

```
<param name="max">50</par
```

```
<message key="errors.rang
```

```
</field-validator>
```

```
</field>
```

# Struts2 内建的验证程序

- required: 确保某给定字段的值不是空值 null ""
- requiredstring: 确保某给定字段的值既不是空值 null, **也不是空白**.
  - trim 参数. 默认为 true, 表示 struts 在验证该字段值之前先剔除前后空格.
- stringlength: 验证一个非空的字段值是不是有足够的长度.
  - minLength: 相关字段的最小长度. 若没有给出这个参数, 该字段将没有最小长度限制
  - maxLength: 相关字段的最大长度. 若没有给出这个参数, 该字段将没有最大长度限制
  - trim: 在验证之前是否去除前后空格
- date: 确保某给定日期字段的值落在一个给定的范围内
  - max: 相关字段的最大值. 若没给出这个参数, 该字段将没有最大值限制
  - min: 相关字段的最小值. 若没给出这个参数, 该字段将没有最小值限制



# Struts2 内建的验证程序

- email: 检查给定 String 值是否是一个合法的 email
- url: 检查给定 String 值是否是一个合法的 url
- regex: 检查某给定字段的值是否与一个给定的正则表达式模式相匹配.
  - expression\*: 用来匹配的正则表达式
  - caseSensitive: 是否区分字母的大小写. 默认为 true
  - trim: 是否去除前后空格. 默认为 true
- int: 检查给定整数字段值是否在某一个范围内
  - min: 相关字段的最小值. 若没给出这个参数, 该字段将没有最小值限制
  - max: 相关字段的最大值. 若没给出这个参数, 该字段将没有最大值限制

# Struts2 内建的验证程序

- conversion: 检查对给定 Action 属性进行的类型转换是否会导致一个转换错误. 该验证程序还可以在默认的类型转换消息的基础上添加一条自定义的消息
- expression 和 fieldexpression: 用来验证给定字段是否满足一个 OGNL 表达式.
  - 前者是一个非字段验证程序, 后者是一个字段验证程序.
  - 前者在验证失败时将生成一个 action 错误, 而后者在验证失败时会生成一个字段错误
  - expression\*: 用来进行验证的 OGNL 表达式

```
<validator type="expression">  
  <param name="expression"><![CDATA[(max>min)]]></param>  
  <message key="expression.max.min"></message>  
</validator>
```

# 短路验证器

- `<validator .../>` 元素和 `<field-validator .../>` 元素可以指定一个可选的 **short-circuit 属性**，该属性**指定该验证器是否是短验证器，默认值为 false**。
- 对同一个字段内的多个验证器，如果一个短路验证器验证失败，其他验证器不会继续校验

# 非字段验证示例

```
<validator type="expression">  
  <param name="expression"><![CDATA[ (password==repassword) ]]></param>  
  <message>两次输入的密码必须完全一致! </message>  
</validator>
```

# 字段验证 vs 非字段验证

- 字段验证字段优先，可以为一个字段配置多个验证规则
- 非字段验证验证规则优先
- 大部分验证规则支持两种验证器，但个别的验证规则只能使用非字段验证，例如表达式验证。

# 错误消息的重用性

- 多个字段使用同样的验证规则，可否使用同一条验证消息？

[-] ● this	IntRangeFieldValidator
[+] ◆ defaultMessage	"Age needs to be betwee
[+] ■ fieldName	("age" (id=110))
[+] ◆ log	JdkLogger (id=111)
[+] ▲ max	Integer (id=114)
◆ messageKey	null
■ messageParameters	null
[+] ▲ min	Integer (id=118)
■ parse	false
■ shortCircuit	false
[+] ■ stack	OgnlValueStack (id=119)
■ type	null
[+] ■ type	"int" (id=125)
[+] ■ validatorContext	DelegatingValidatorCont

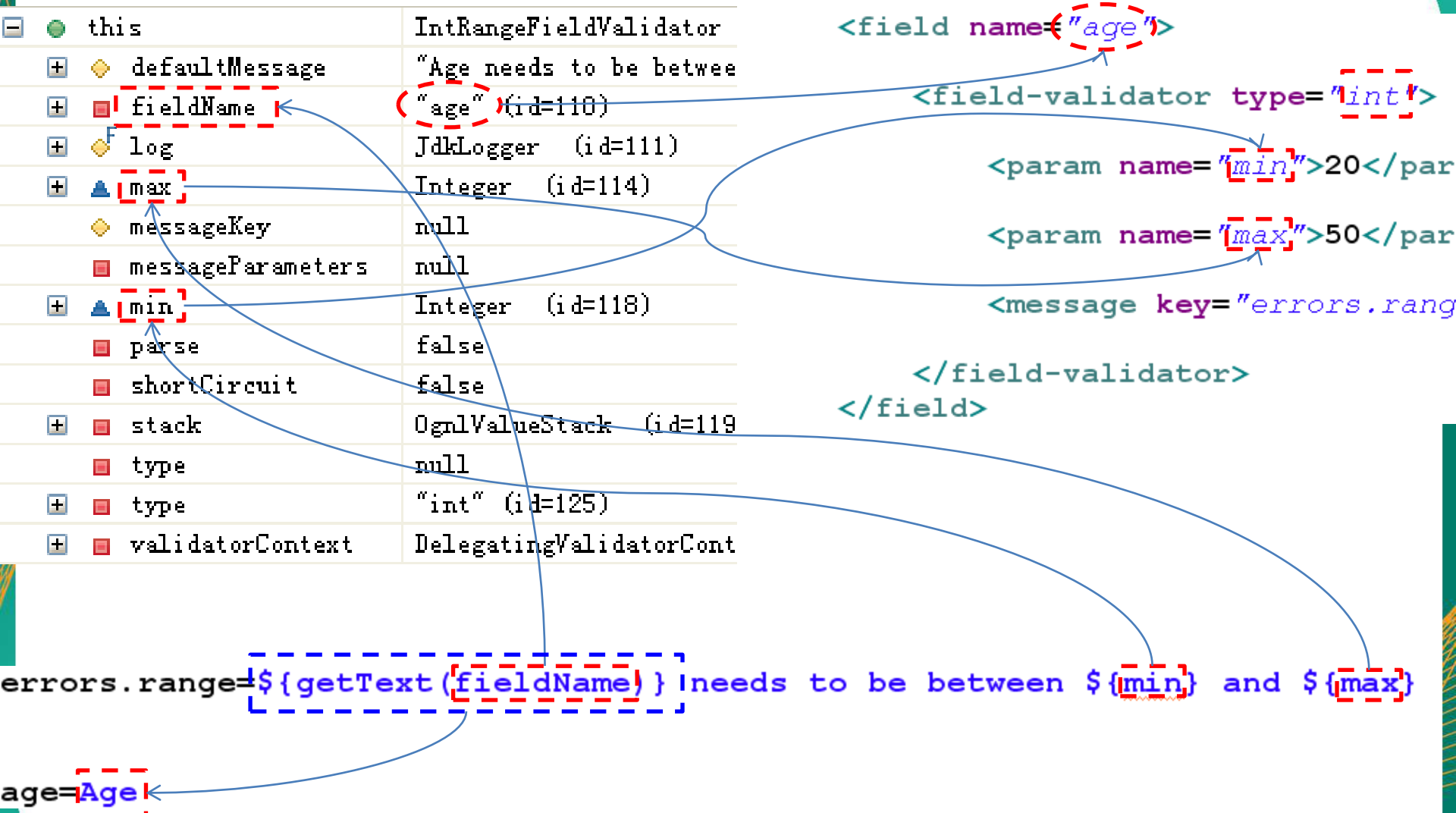
```

<field name="age">
  <field-validator type="int">
    <param name="min">20</param>
    <param name="max">50</param>
    <message key="errors.range">
      {0} needs to be between {1} and {2}
    </message>
  </field-validator>
</field>

```

errors.range=\${getText(fieldName)} needs to be between \${min} and \${max}

age=Age



# 自定义验证器

- 自定义验证器必须实现 Validator 接口。
- ValidatorSupport 和 FieldValidatorSupport 实现了 Validator 接口

```
Validator<T> - com.opensymphony.xwork2.validator
+ ValidatorSupport - com.opensymphony.xwork2.validator.validators
- FieldValidator - com.opensymphony.xwork2.validator
  - FieldValidatorSupport - com.opensymphony.xwork2.validator.validators
```

- 若需要普通的验证程序, 可以继承 ValidatorSupport 类
- 若需要字段验证程序, 可以继承 FieldValidatorSupport 类
- 若验证程序需要接受一个输入参数, 需要为这个参数增加一个相应的属性
- 注册验证程序: 自定义验证器需要在类路径里的某个 validators.xml 文件里注册: 验证框架首先在根目录下找 validators.xml 文件, 没找到 validators.xml 文件, 验证框架将调用默认的验证设置, 即 default.xml 里面的配置信息.



# 自定义验证器

- 自定义一个 18 位身份证验证器
  - 编写验证器类
  - 在 validators.xml 文件中进行注册
  - 在验证配置文件中使用的

# 编程验证

- Struts2 提供了一个 Validateable 接口, 可以使 Action 类实现这个接口以提供编程验证功能.
- ActionSupport 类已经实现了 Validateable 接口

```
@Override
public void validate() {
    if(name == null || name.trim().equals("")){
        addFieldError("name", getText("name.null"));
    }
}
```

# 文件的上传下载

讲师：佟刚

新浪微博：@尚硅谷-佟刚

# 表单的准备

- 要想使用 HTML 表单上传一个或多个文件
  - 须把 HTML 表单的 enctype 属性设置为 **multipart/form-data**
  - 须把 HTML 表单的 method 属性设置为 **post**
  - 需添加 `<input type= "file" >` 字段.

# Struts 对文件上传的支持

- 在 Struts 应用程序里, **FileUpload 拦截器**和 Jakarta Commons FileUpload 组件可以完成文件的上传.
- 步骤:
  - 1. 在 Jsp 页面的文件上传表单里使用 file 标签. 如果需要一次上传多个文件, 就必须使用多个 file 标签, 但它们的名字必须是相同的
  - 2. 在 Action 中新添加 3 个和文件上传相关的属性. 这 3 个属性的名字必须是以下格式
    - [File Name] : File -被上传的文件。例如 : data
    - [File Name]ContentType : String -上传文件的文件类型。例如 : dataContentType
    - [File Name]FileName : String -上传文件的文件名。例如 : dataFileName
  - 如果上上传多个文件, 可以使用 List

# 配置 FileUpload 拦截器

- FileUpload 拦截器有 3 个属性可以设置。
  - maximumSize: 上传单个文件的最大长度(以字节为单位), 默认值为 2 MB
  - allowedTypes: 允许上传文件的类型, 各类型之间以逗号分隔
  - allowedExtensions: 允许上传文件扩展名, 各扩展名之间以逗号分隔
  - 可以在 struts.xml 文件中覆盖这 3 个属性

```
<interceptor-ref name="defaultStack">  
  <param name="fileUpload.maximumSize">1048576</param>  
  <param name="fileUpload.allowedTypes">application/vnd.ms-powerpoint, application/msword</param>  
  <param name="fileUpload.allowedExtensions">ppt, doc</param>  
</interceptor-ref>
```

- Commons FileUpload 组件默认接受上传文件总的最大值为 2M, 可以通过在 struts 配置文件中配置常量的方式修改
- 与文件上传有关的出错消息在 struts-messages.properties 文件里预定义. 可以在文件上传 Action 相对应的资源文件中重新定义错误消息

# 文件下载概述

- 在某些应用程序里, 可能需要动态地把一个文件发送到用户的浏览器中, 而这个文件的名字和存放位置在编程时是无法预知的

# Stream 结果类型

- Struts 专门为文件下载提供了一种 Stream 结果类型. 在使用一个 Stream 结果时, 不必准备一个 JSP 页面.
- Stream 结果类型可以设置如下参数:
  - contentType : 被下载的文件 MIME 类型。默认值为 text/plain
  - contentLength : 被下载的文件的大小, 以字节为单位
  - contentDisposition : 可以设置下载文件名的 ContentDisposition 响应头, 默认值为 inline, 通常设置为如下格式: *attachment;filename="document.pdf"*.
  - inputName : Action 中提供的文件的输入流。默认值为 inputStream
  - bufferSize : 文件下载时缓冲区的大小。默认值为 1024
  - allowCaching : 文件下载时是否允许使用缓存。默认值为 true
  - contentType : 文件下载时的字符编码。
- **Stream 结果类型的参数可以在 Action 以属性的方式覆盖**



# 防止表单重复提交

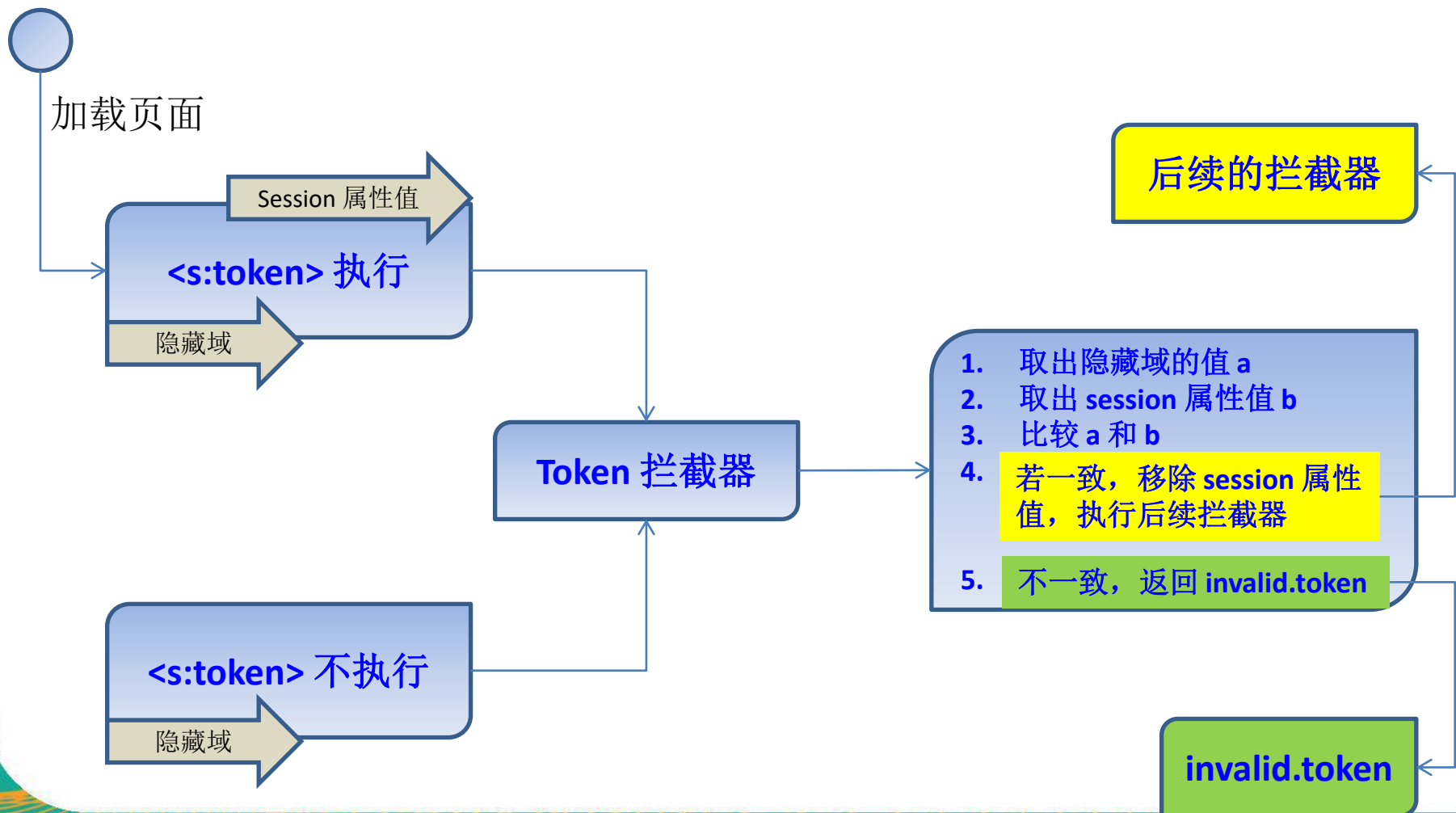
讲师：佟刚

新浪微博：@尚硅谷-佟刚

# 概述

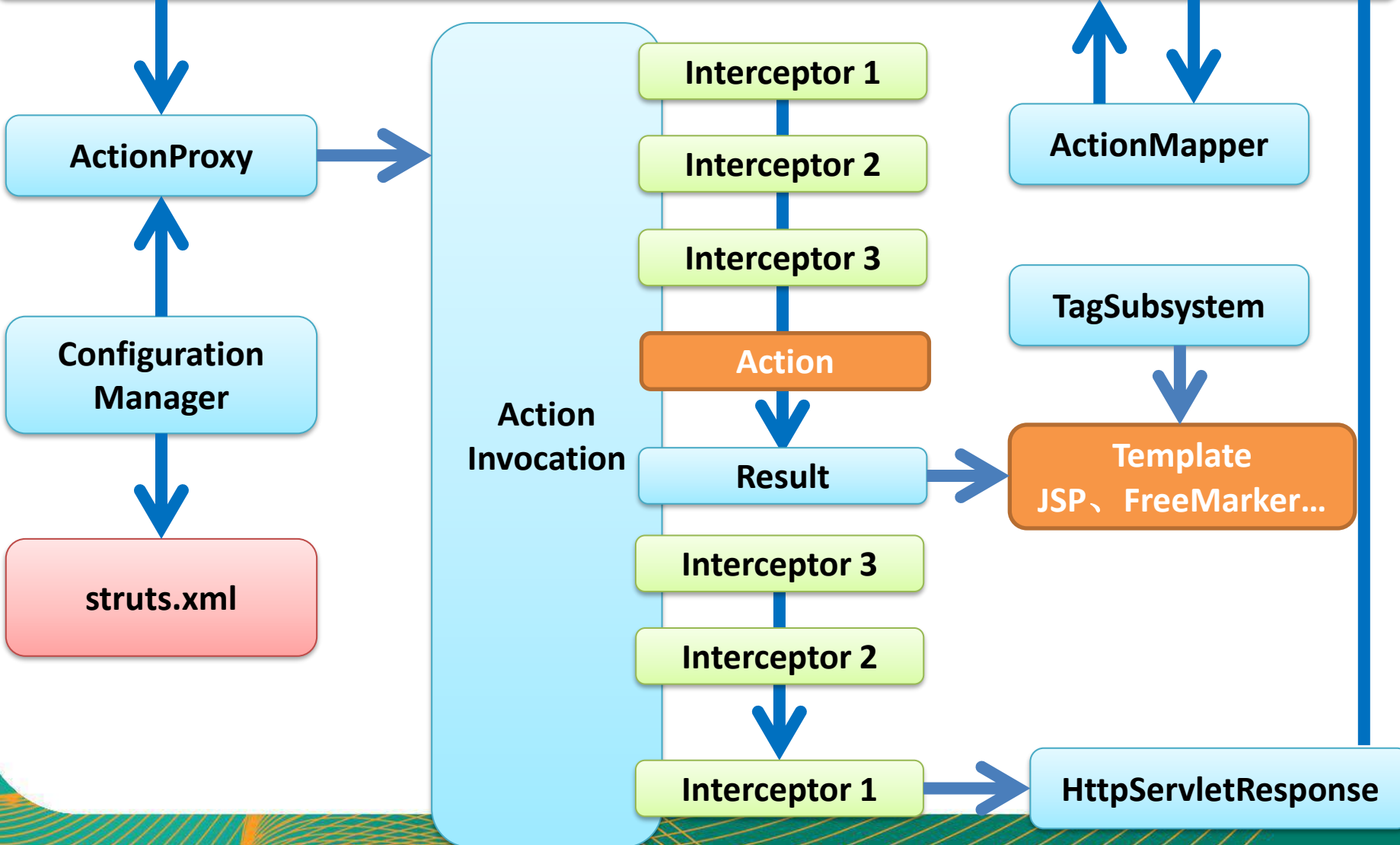
- 表单的重复提交：
  - 若刷新表单页面, 再提交表单不算重复提交.
  - 在不刷新表单页面的前提下:
    - 多次点击提交按钮
    - 已经提交成功, 按 "回退" 之后, 再点击 "提交按钮".
    - 在控制器响应页面的形式为转发情况下, 若已经提交成功, 然后点击 "刷新(F5) "
- 重复提交的缺点:
  - 加重了服务器的负担
  - 可能导致错误操作.

# Struts2 解决表单重复提交



# 标记管理

- Struts 提供的 **token 标签**可以用来生成一个独一无二的标记. 这个标签**必须嵌套在 form 标签的内部使用, 它将在表单里插入一个隐藏字段并把标记值 ( 隐藏域的字段的价值 ) 保存在 HttpSession 对象里.**
- Token 标签必须与 Token 或 TokenSession 拦截器配合使用, 这两个拦截器都能对标记进行处理.
- Token 拦截器在遇到重复提交情况时, 会返回 invalid.token 结果并加上一个 Action 错误. 这种错误默认的消息是: The form has already been processed or no token was supplied, please try again.
- **TokenSession 拦截器采取的做法只是阻断后续的提交, 用户将看到同样的响应, 但实际上并没有重复提交**



# 自定义拦截器

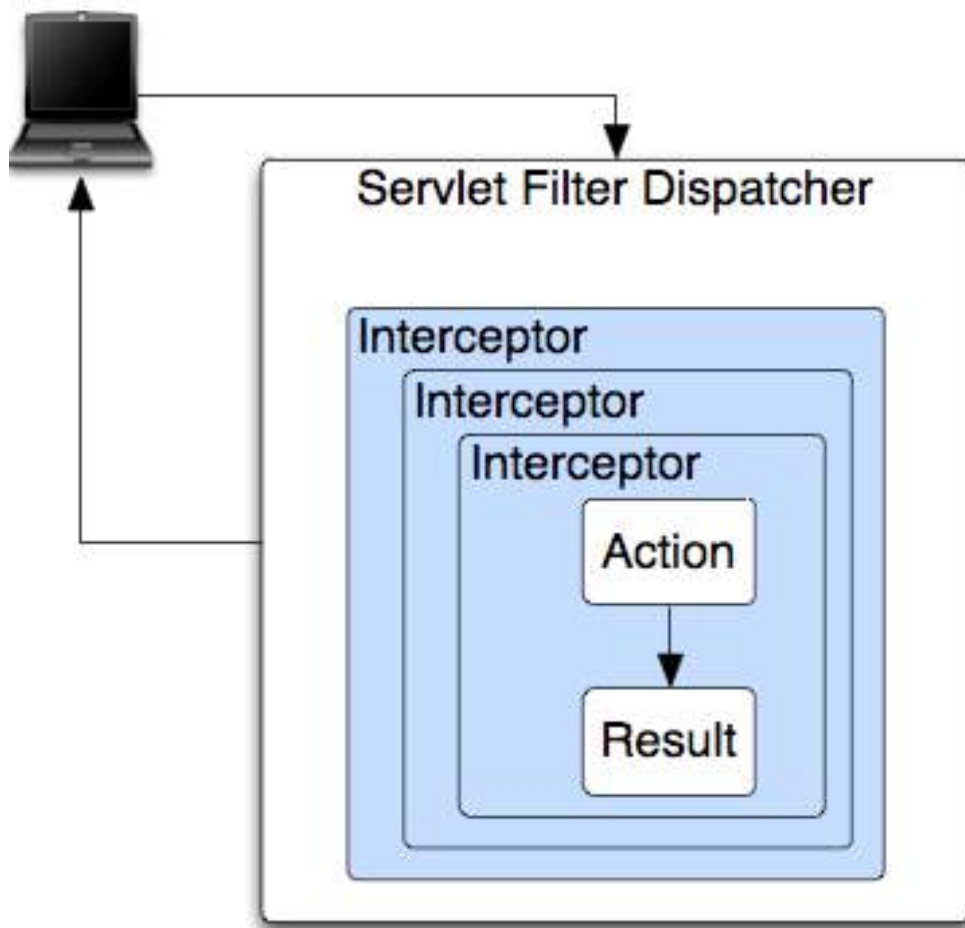
讲师：佟刚

新浪微博：@尚硅谷-佟刚

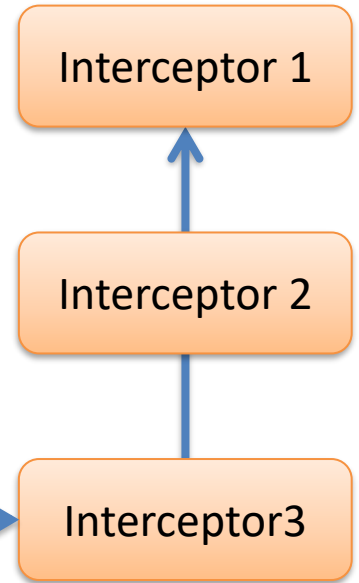
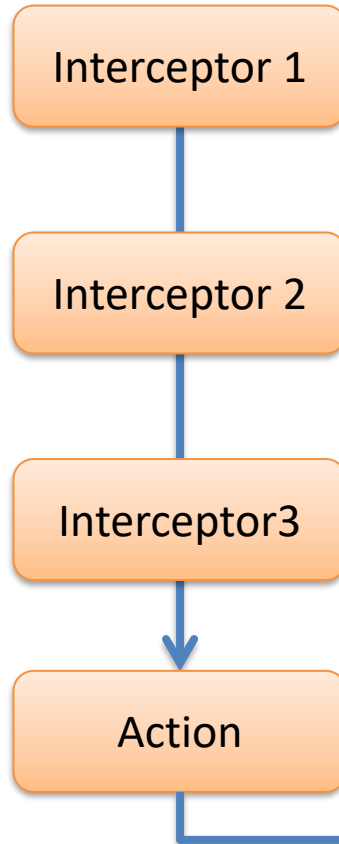
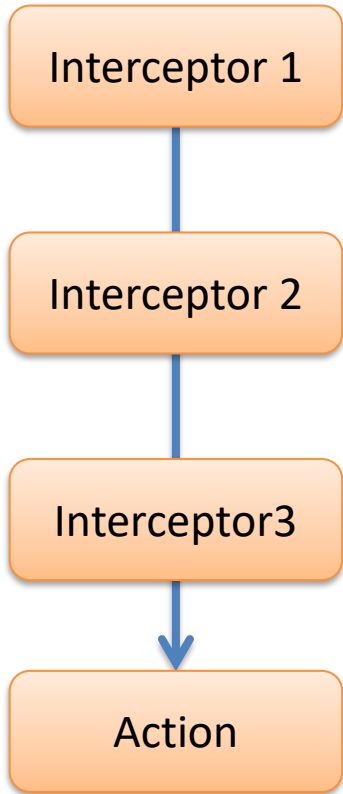
# Struts2 拦截器

- 拦截器 ( Interceptor ) 是 Struts 2 的核心组成部分。
- Struts2 很多功能都是构建在拦截器基础之上的, 例如文件的上传和下载、国际化、数据类型转换和数据校验等等。
- Struts2 拦截器在访问某个 Action 方法之前或之后实施拦截
- Struts2 拦截器是可插拔的, 拦截器是 AOP ( 面向切面编程 ) 的一种实现 .
- 拦截器栈(Interceptor Stack): 将拦截器按一定的顺序联结成一条链. 在访问被拦截的方法时, Struts2 拦截器链中的拦截器就会按其之前定义的顺序被依次调用

# Struts2 拦截器







# Struts2 自带的拦截器(1)

拦截器	名字	说明
Alias Interceptor	alias	在不同请求之间将请求参数在不同名字件转换，请求内容不变
Chaining Interceptor	chain	让前一个Action的属性可以被后一个Action访问，现在和chain类型的result（）结合使用。
Checkbox Interceptor	checkbox	添加了checkbox自动处理代码，将没有选中的checkbox的内容设定为false，而html默认情况下不提交没有选中的checkbox。
Cookies Interceptor	cookies	使用配置的name,value来是指cookies
Conversion Error Interceptor	conversionError	将错误从ActionContext中添加到Action的属性字段中。
Create Session Interceptor	createSession	自动的创建HttpSession，用来为需要使用到HttpSession的拦截器服务。
Debugging Interceptor	debugging	提供不同的调试用的页面来展现内部的数据状况。
Execute and Wait Interceptor	execAndWait	在后台执行Action，同时将用户带到一个中间的等待页面。
Exception Interceptor	exception	将异常定位到一个画面
File Upload Interceptor	fileUpload	提供文件上传功能
I18n Interceptor	i18n	记录用户选择的locale
Logger Interceptor	logger	输出Action的名字
Message Store Interceptor	store	存储或者访问实现ValidationAware接口的Action类出现的消息，错误，字段错误等。
Model Driven Interceptor	model-driven	如果一个类实现了ModelDriven，将getModel得到的结果放在Value Stack中。

拦截器	名字	说明
Scoped Model Driven	scoped-model-driven	如果一个Action实现了ScopedModelDriven, 则这个拦截器会从相应的Scope中取出model调用Action的setModel方法将其放入Action内部。
Parameters Interceptor	params	将请求中的参数设置到Action中去。
Prepare Interceptor	prepare	如果Action实现了Preparable, 则该拦截器调用Action类的prepare方法。
Scope Interceptor	scope	将Action状态存入session和application的简单方法。
Servlet Config Interceptor	servletConfig	提供访问HttpServletRequest和HttpServletResponse的方法, 以Map的方式访问。
Static Parameters Interceptor	staticParams	从struts.xml文件中将中的中的内容设置到对应的Action中。
Roles Interceptor	roles	确定用户是否具有JAAS指定的Role, 否则不予执行。
Timer Interceptor	timer	输出Action执行的时间
Token Interceptor	token	通过Token来避免双击
Token Session Interceptor	tokenSession	和Token Interceptor一样, 不过双击的时候把请求的数据存储在Session中
Validation Interceptor	validation	使用action-validation.xml文件中定义的内容校验提交的数据。
Workflow Interceptor	workflow	调用Action的validate方法, 一旦有错误返回, 重新定位到INPUT画面
Parameter Filter Interceptor	N/A	从参数列表中删除不必要的参数
Profiling Interceptor	profiling	通过参数激活profile

# Interceptor 接口

- 每个拦截器都是实现了 `com.opensymphony.xwork2.interceptor.Interceptor` 接口的 Java 类:

```
public interface Interceptor extends Serializable {  
    void destroy();  
    void init();  
    String intercept(ActionInvocation invocation) throws Exception;  
}
```

- `init`: 该方法将在拦截器被创建后立即被调用, 它在拦截器的生命周期内只被调用一次. 可以在该方法中对相关资源进行必要的初始化
- `intercept`: 每拦截一个请求, 该方法就会被调用一次.
- `destroy`: 该方法将在拦截器被销毁之前被调用, 它在拦截器的生命周期内也只被调用一次.

# Interceptor 接口

- Struts 会依次调用为某个 Action 而注册的每一个拦截器的 intercept 方法。
- 每次调用 intercept 方法时, Struts 会传递一个 **ActionInvocation** 接口的实例。
- ActionInvocation: 代表一个给定 Action 的执行状态, 拦截器可以从该类的对象里获得与该 Action 相关联的 Action 对象和 Result 对象. 在完成拦截器自己的任务之后, 拦截器将调用 ActionInvocation 对象的 invoke 方法前进到 Action 处理流程的下一个环节。
- **AbstractInterceptor** 类实现了 Interceptor 接口. 并为 init, destroy 提供了一个空白的实现

# 自定义拦截器

- 定义自定义拦截器的步骤
  - 自定义拦截器
  - 在 struts.xml 文件中配置自定义的拦截器

# 零配置

讲师：佟刚

新浪微博：@尚硅谷-佟刚

# Convention 插件

- 从 Struts 2.1 开始, Struts 可以使用 **Convention 插件**来支持零配置:
- Convention 插件**完全抛弃配置信息, 不仅不需要使用 struts.xml 文件进行配置, 甚至不需要使用 Annotation 进行配置.** 而是完全**根据约定来自动配置.**
- 安装 Convention 插件: 复制 struts-2.2.1\lib\struts2-convention-plugin-2.2.1.jar 到当前当前 WEB 应用的 WEB-INF 的 lib 目录下.



# 搜索 Action

- 搜索 Action: 对于 Convention 插件, 它会自动搜索位于 **action, actions, struts, struts2** 包下的所有 Java 类, Convention 会把如下两种 Java 类当成 Action 处理
  - 所有**实现了 Action 接口**的 Java 类
  - 所有类名以 **Action 结尾**的 Java 类
- 下面是符合 Convention 插件的 Action 类:
  - org.simpleit.**actions.LoginAction**
  - org.simpleit.**actions.book.Books**(Books 实现了 Action 接口)
  - org.simpleit.**action.LoginAction**
  - org.simpleit.**struts.action.book.BookAction**
  - org.simpleit.**struts2.emp.EmployeeAction**

# 按约定映射命名空间

- 找到合适的 Action 类之后, Convention 插件会按约定部署这些 Action, 部署 Action 时, actions, action, struts, struts2 包会被映射为根命名空间, 而这些包下的子包则被映射成对应的命名空间:
  - org.simpleit.**actions.LoginAction** /
  - org.simpleit.**actions**.book.Books(Books 实现了 Action 接口) /book
  - org.simpleit.**action.LoginAction** /
  - org.simpleit.**struts**.action.book.**BookAction** /action/book
  - org.simpleiti.**struts2**.emp.**EmployeeAction** /emp

# 按约定映射 Action

- Action 的 name 属性(也就是该 Action 所要处理的 URL) 则根据该 Action 的类名映射. 映射 Action 的 name 时, 遵循如下规则:
  - 若该 Action 类名包含 Action 后缀, **将该 Action 类名的 Action 后缀去掉.** 否则不作任何处理
  - 将 Action 类名的驼峰写法转成中横线 (-) 写法: 所有字母小写, 单词之间使用 - 隔开.
- 例子:
  - org.simpleit.**actions.LoginAction** → /login.action
  - org.simpleit.**actions.book.Books**(Books 实现了 Action 接口) → /book/books
  - org.simpleit.**action.LoginAction** → /login.action
  - org.simpleit.**struts.action.book.BookAction** → /action/book/book.action
  - org.simpleiti.**struts2.emp.EmployeeAction** /emp/employee.action

# 按约定映射 Result

- 默认情况下, Convention 总会到 WEB 应用的 WEB-INF/content 路径下定位物理资源, 定位资源的约定是: `actionUrl + resultCode suffix`. 当某个逻辑视图找不到对应的视图资源时, Conversion 会自动试图使用 `actionUrl` 作为物理资源.
- 例子:
  - `org.simpleit.actions.LoginAction` → `/WEB-INF/content/login.jsp` 或 `login-success.jsp`
  - `org.simpleit.actions.book.Books`(Books 实现了 Action 接口) → `/WEB-INF/content/book/books-success.jsp` 或 `books.jsp`
  - `org.simpleit.struts.action.book.BookAction` → `/WEB-INF/action/book/book-success.jsp` 或 `book.jsp`
  - `org.simpleti.struts2.emp.EmployeeAction` `/WEB-INF/content/emp/employee-success.jsp` 或 `employee.jsp`

# Action 链的约定

- 如果希望一个 Action 处理结束后不是进入视图页面, 而是进行另一个 Action 形成 Action 链. 通过 Conversion 插件则只需遵守如下三个约定即可.
  - 第一个 Action 返回的逻辑视图字符串没有对应的视图资源
  - 第二个 Action 与第一个 Action 处于用一个包下
  - 第二个 Action 映射的 URL 为: `firstActionUrl + resultCode`。

常量名	说明
<code>struts.conversion.disableJarScanning</code>	设置是否从 JAR 包里搜索 Action 类。
<code>struts.conversion.result.path</code>	设置 Convention 插件定位视图资源的根路径. 默认值为 <code>/WEB-INF/content</code>
<code>struts.conversion.action.suffix</code>	设置 Convention 搜索 Action 类的类名后缀. 默认为 <code>Action</code>
<code>struts.conversion.checkImplementsAction</code>	设置是否将实现了 Action 接口的类映射成 Action, 默认为 <code>true</code>
<code>struts.conversion.action.name.lowercase</code>	设置映射 Action 时, 是否将 Action 的 name 属性值转为所有字母小写, 默认是 <code>true</code>
<code>struts.convention.default.parent.package</code>	指定 Convention 映射的 Action 所在包的默认父包, 默认为 <code>convention-default</code>
<code>struts.conversion.action.name.seperator</code>	设置映射 Action 时指定 name 属性各单词之间的分隔符. 默认值是 <code>中划线(-)</code>
<code>struts.convention.package.locators</code>	在 <code>struts.convention.package.locators.basePackage</code> 的基础上, 指定 Convention 插件搜索 Action 的包. 默认值时: <code>action, actions, struts, struts2</code>
<code>struts.convention.package.locators.basePackage</code>	指定 Convention 插件从哪个包搜索 Action
<code>struts.convention.exclude.packages</code>	指定排除在搜索 Action 之外的包. 默认值为: <code>org.apache.struts.*,org.apache.struts2.*,org.springframework.web.struts.*,org.springframework.web.struts2.*,org.hibernate.*</code>
<code>struts.convention.relative.result.types</code>	指定 Convention 映射 Result 时默认支持的结果类型. 默认值为: <code>dispatcher,velocity,freemarker</code>

# Convention 的 Annotation

- Conversion 插件使用 Annotation 来管理拦截器, 异常处理等相关配置. Conversion 还允许使用 Annotation 管理 Action 和 Result 的配置, 从而覆盖 Conversion 的约定.

# Action 配置相关的 Annotation

- 与 Action 相关的两个 Annotation 是 @Action 和 @Actions
- @Action 主要用于修饰 Action 类里的方法, 用于将方法映射为指定的 URL.
  - @Action 可以指定一个 value 属性, 用于指定该 Action 映射的 URL(类似于在 struts.xml 文件中配置该 Action 时为 <action /> 元素指定的 name 属性值)
  - @Action 还可以指定一个 param 属性, 该属性是一个字符串数组, 用于该 Action 指定参数名和参数值. params 属性值应该遵守如下格式: { "name1", "value1", "name2", "value2", ...}. 该属性用于为该 Action 注入属性值
- @Actions 也用于修饰 Action 类里的方法, 用于将该方法映射到多个 URL. @Actions 用于组织多个 @Action.



# Result 配置相关的 Annotation

- 和 Result 配置相关的 3 个 Annotation 是 @Result , @Results 和 @ResultPath
- @Results 用于组织多个 @Result, 因此它只需一个 value 属性值, 该 value 属性值为多个 @Result。
- @Result 用于定义逻辑视图和物理视图之间的对应关系, 也就是相当于 struts.xml 文件里 <result .../> 元素里的作用
  - name\*: 指定 result 的名字, 相当于 <result .../> 节点的 name 属性
  - type: 指定视图资源的类型, 相当于 <result .../> 节点的 type 属性
  - locations: 指定实际视图的位置, 相当于 <result ...></result> 的中间部分
  - params: 为视图资源指定参数值. 属性值应满足 {name1, value1, name2, value2...} 的格式. 相当于 <result...></result> 的 <param> 子节点
- @Result 有如下两种用法
  - Action 级的 Result 映射: 以 @Actions 组合多个 @Action 后修饰的 Action 类. 这种 Result 映射对该 Action 里的所有方法都有效
  - 方法级的 Result 映射: 将多个 @Result 组成数组后作为 @Action 的 results 属性值. 这种 Result 映射仅对被修饰的方法有效
- @ResultPath 用于改变被修饰 Action 所对应的物理视图资源的根路径. 例如: 默认情况下, Conversion 插件会到 WEB-INF/content 路径下寻找物理视图资源. 但若使用 @ResultPath( "/simpleit" ) 修饰 Action, 系统将会到 simpleit 目录下寻找物理资源

# 包和命名空间相关的 Annotation

- 与包和命名空间相关的 Annotation 有如下 2 个
  - @Namespace: 修改 Action. 该 Annotation 只需指定一个 value 属性值, 用于指定被修改的 Action 所在的命名空间.
  - @ Namespaces: 修饰 Action. 用于组合多个 @Namespace.

- 与异常相关的 Annotation 有 @ExceptionHandler 和 @ExceptionHandler
- @ExceptionHandler 用于定义异常类和物理视图之间的对应关系, 也就是它只需指定一个 value 属性值, 该 value 属性值为多个 @ExceptionHandler
- @ExceptionHandler: 用于定义异常类和物理视图之间的对应关系, 也就是相当于 struts.xml 文件里 <exception-mapping .../> 元素的作用. 使用 @ExceptionHandler 时必须指定如下两个属性:
  - exception: 用于指定异常类, 相当于 <exception-mapping.../> 元素的 exception
  - result: 用于指定逻辑视图名, 相当于 <exception-mapping .../> 元素的 result
- @ExceptionHandler 有如下两种用法:
  - Action 级的异常定义: 以 @ExceptionHandler 组合多个 @ExceptionHandler 后修饰 Action 类. 这种异常定义对 Action 里的所有方法都有效.
  - 方法级的异常定义: 将多个 @ExceptionHandler 组成数组后作为 @Action 的 exceptionMappings 属性值. 这种异常定义仅对修饰的方法有效.

- 拦截器配置相关的 Annotation 有 @InterceptorRef, @InterceptorRefs, @DefaultInterceptorRef
- @InterceptorRefs 用于指定多个 @InterceptorRef, 因此该 Annotation 只需指定一个 value 属性值, 该 value 属性值为多个 @ InterceptorRef
- @InterceptorRef 用于为指定 Action 引用拦截器或者拦截器栈. 也就是 struts.xml 文件中 <action...> 节点内部的 <interceptor-ref .../> 子元素的作用. 属性如下
  - value\*: 用于指定所引用拦截器或拦截器栈的名字, 相当于 <interceptor-ref .../> 子元素中的 name 属性
  - params: 用于覆盖所引用该拦截器的默认参数值. 该属性应满足 {name1, value1, name2, value2, ...} 的格式. 相当于 <interceptor-ref .../> 元素的 <param> 子元素.
- @InterceptorRef 有如下两种用法
  - Action 级的拦截器配置
  - 方法级的拦截器配置
- @DefaultInterceptorRef: 主要用于修饰包, 用于指定该包的默认拦截器. 这个 Annotation 只有一个 value 属性, 用于指定默认拦截器的名字.

# 整合 Spring

讲师：佟刚

新浪微博：@尚硅谷-佟刚

# 概述

- Struts2 通过插件实现和 Spring 的整合.
- Struts2 提供了两种和 Spring 整合基本的策略:
  - 将 Action 实例交给 Spring 容器来负责生成, 管理, 通过这种方式, 可以充分利用 Spring 容器的 IOC 特性, 提供最好的解耦
  - 利用 Spring 插件的自动装配功能, 当 Spring 插件创建 Action 实例后, 立即将 Spring 容器中对应的业务逻辑组件注入 Action 实例.

# 让 Spring 管理控制器

- 将 Action 实例交给 Spring 容器来负责生成, 管理, 通过这种方式, 可以充分利用 Spring 容器的 IOC 特性, 提供最好的解耦
- 整合流程:
  - 安装 Spring 插件: 把 struts2-spring-plugin-2.2.1.jar 复制到当前 WEB 应用的 WEB-INF/lib 目录下
  - 在 Spring 的配置文件中配置 Struts2 的 Action 实例
  - 在 Struts 配置文件中配置 action, 但其 class 属性不再指向该 Action 的实现类, 而是指向 Spring 容器中 Action 实例的 ID

# 自动装配

- 利用 Spring 插件的自动装配功能, 当 Spring 插件创建 Action 实例后, 立即将 Spring 容器中对应的业务逻辑组件注入 Action 实例.
- 配置自动装配策略: Spring 插件的自动装配可以通过 `struts.objectFactory.spring.autoWire` 常量指定, 该常量可以接受如下值:
  - name: 根据属性名自动装配.
  - type: 根据类型自动装配. 若有多个 type 相同的 Bean, 就抛出一个致命异常; 若没有匹配的 Bean, 则什么都不会发生, 属性不会被设置
  - auto: Spring 插件会自动检测需要使用哪种方式自动装配方式
  - constructor: 同 type 类似, 区别是 constructor 使用构造器来构造注入所需的参数
- 整合流程:
  - 安装 Spring 插件
  - 正常编写 struts 配置文件
  - 编写 spring 配置文件, 在该配置文件中不需要配置 Action 实例



