Struts2

概念：是一个 mvc 框架

Servlet 的缺点

1、 在 web.xml 文件中需要配置很多行代码，维护起来很不方便，不利于团队合作

2、 一个 servlet 的入口只有一个 doPost 或者 doGet 方法，如果在一个 servlet 中写好几个方法，怎么办？

```
public class UserServlet extends HttpServlet{
    doPost(){
        this.doGet();
    }
    doGet(){
        if(request.getParameter("method").equals("save")){
            this.save();
        }else{
             this.update();
        }
    }
    private void save(){

    }
    private void update(){

    }
}
```

这样会导致代码结构很乱

3、 servlet 类与 servlet 容器高度耦合，每个方法中都有两个参数 request，response。如果服务器不启动，这两个参数没有办法初始化。

4、 如果在 servlet 中的一个方法中，有很多功能，这个时候，会导致该方法比价复杂，以致于不利于维护

```
doGet(){
    用户注册
    //用户头像的一个上传
    //验证表单
    //权限的验证
    //注册
}
```

用户注册完成了 4 件事情，所以整个方法比较杂乱

5、 如果一个 servlet 类中有很多方法，浏览器对这些方法进行请求，url 写起来很麻烦

```
http://localhost:8080/servlet/userServlet?method=save/update
```

6、在 servlet 中如果要获取页面上表单中的数据，那么在方法中会写很多行

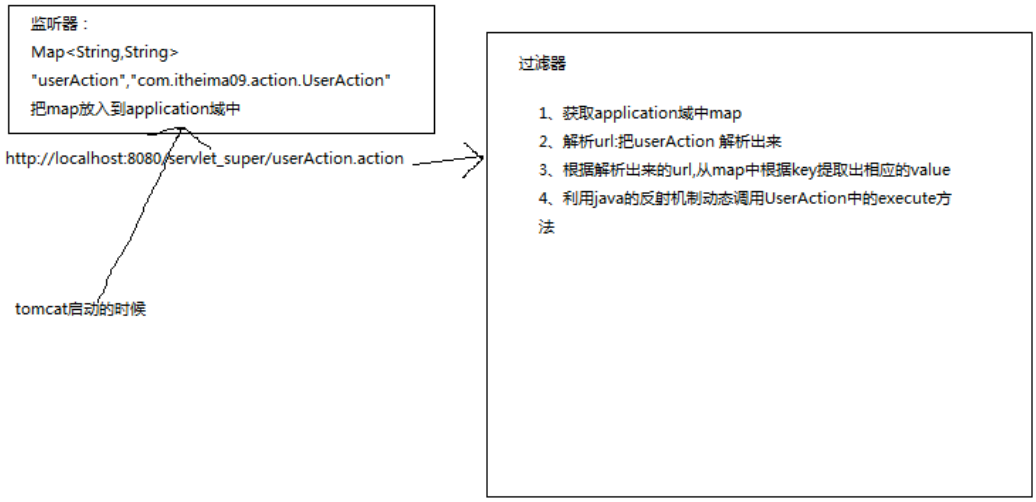
Servlet 的重构

目的

1、 在 web.xml 文件中只写一个过滤器
2、 用 action 处理业务逻辑
3、 在过滤器中动态的调用 action 中的方法处理业务逻辑

类的设计

技术路线总图：



1、 监听器
    1、 准备一个 map
    2、 把所有的 action 的 key,value 放入到 map 中
    3、 把 map 放入到 application 域中
2、 过滤器
    1、 获取 application 域中的 map
    2、 解析 url
    3、 根据解析的 url 从 map 中把 value 提取出来
    4、 根据 java 的反射机制动态调用 action
    5、 根据 action 返回的方法跳转到相应的页面
3、执行 action 的 execute 方法，该方法返回一个字符串

实现

1、 写监听器

```java
public class ServletListener implements ServletContextListener{
    /**
     * 在tomcat销毁的时候执行
     */
    @Override
    public void contextDestroyed(ServletContextEvent arg0) {
        arg0.getServletContext().setAttribute("mappings", null);
    }
    /**
     *  在tomcat启动的时候执行
     */
    @Override
    public void contextInitialized(ServletContextEvent arg0) {
        Map<String, String> map = new HashMap<String, String>();
        map.put("userAction", "com.itheima09.action.UserAction");
        arg0.getServletContext().setAttribute("mappings", map);
    }
}
```

```xml
<listener>
    <listener-class>com.itheima09.servlet.listener.ServletListener</listener-class>
</listener>
```

2、 过滤器

```java
/**
 * 1、从 application域中获取map
 */
HttpServletRequest request = (HttpServletRequest)arg0;
HttpServletResponse response = (HttpServletResponse)arg1;
Map<String, String> map = (HashMap<String, String>)request.getServletContext().getAttribute("mappings");
/**
 * 2、获取浏览器中的url,把url解析出来
 *     http://localhost:8080/itheima09_servlet_super/userAction.action
 *     ---->userAction
 */
//mapping = userAction
String mapping = ServletUtils.parse(request.getRequestURI());
String value = map.get(mapping); //value就是action的类的全名
try {
    Class class1 = Class.forName(value);
    Method method = class1.getMethod("execute", HttpServletRequest.class,HttpServletResponse.class);
    //调用了action中的方法
    String jspName = (String)method.invoke(class1.newInstance(), request,response);
    request.getRequestDispatcher(jspName).forward(request, response);
} catch (Exception e) {
```

```xml
<filter>
    <filter-name>actionFilter</filter-name>
    <filter-class>com.itheima09.servlet.filter.DispatcherFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>actionFilter</filter-name>
    <url-pattern>*.action</url-pattern>
</filter-mapping>
```

3、 写 action

```java
public class UserAction {
    public String execute(HttpServletRequest request,HttpServletResponse response){
        return "index.jsp";
    }
}
```

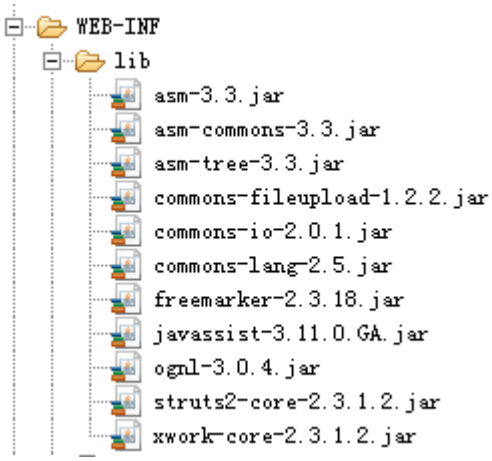
Struts2 的历史

1、 servelet
2、 struts1
    1、 写 action
    2、 写了一个中控的 servlet
    3、 actionForm 和页面上表单中的内容一致
3、webwork
    1、使得 action 与 servlet 容器完全松耦合
    2、属性驱动和模型驱动获取页面上表单中的数据
    3、利用了拦截器的概念把 servlet 容器的第 4 个缺点克服掉了
4、struts1+webwork=struts2

Struts2 的第一个例子

步骤

1、 创建一个 web project
2、 导入 jar 包



3、 编写 web.xml 文件

```xml
<filter>
    <filter-name>struts2</filter-name>
    <filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

4、 写一个 action

```java
public class HelloWorldAction {
    public String hello(){
        System.out.println("hello");
        return "index";
    }
}
```

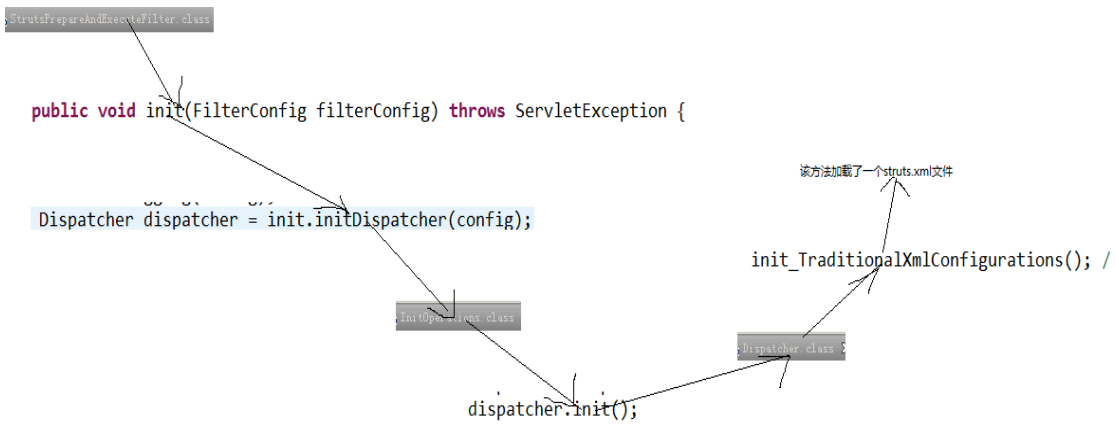
5、 编码 struts.xml 文件

```xml
<struts>
    <package name="helloworld" namespace="/" extends="struts-default">
        <action name="helloWorldAction" class="com.itheima09.struts2.action.HelloWorldAction">
            <result name="index">index.jsp</result>
        </action>
    </package>
</struts>
```

6、 运行

解析

struts.xml 文件的内容



上图为加载流程

注意：

1、 struts.xml 文件必须放在 classpath 的根目录下

2、 名字必须为 struts.xml 文件

3、 因为整个加载过程写在了过滤器中的 init 方法中，所以 tomcat 启动的时候就把该文件加载了

Package

1、 用意：用来管理 action 的

```
<package name="system">
        <action name="userAction">
        <action name="departmentAction">
        <action name="roleAction">
</package>
```

从上图可以看出在 system 模块下有三个 action

2、 name 属性

为包的名称，是唯一的

3、 namespace

为命名空间，是针对 url 的

```
namespace="/hello"
```

上述的命名空间针对的是:itheima09_struts2_helloworld/hello

当浏览器提交一个 url:

```
localhost:8088/itheima09_struts2_helloworld/helloWorldAction.action
```

上述的 url 直接从项目的根目录查找，所以找不到

```
localhost:8088/itheima09_struts2_helloworld/hello/helloWorldAction.action
```

该路径和上述的 url 是对应的

```
localhost:8088/itheima09_struts2_helloworld/hello/a/helloWorldAction.action
```

先找 hello/a 下的 action，如果找不到，则查找上一层，再找不到再找上一层，直到找到，如果最上层找不到，则报错

```
namespace="/"
```

上述的命名空间针对的是:itheima09_struts2_helloworld

只要命名空间加一层，最后跳转到相应的 jsp 以后，也会加上相应的路径

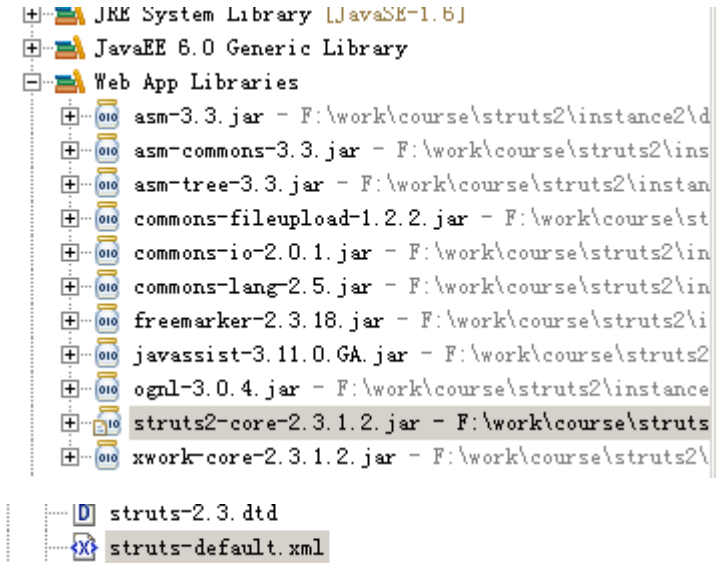
```
localhost:8088/itheima09_struts2_helloworld/hello/a/helloWorldAction.action
```

```
itheima09_struts2_helloworld/hello/a/index.jsp
```

这样的体系不好。

4、 extends

1、 在 tomcat 启动的时候，不仅加载了 struts.xml 文件，而且还加载了 struts-default.xml 文件，而这个文件在 classpath 下。针对该文件的路径在

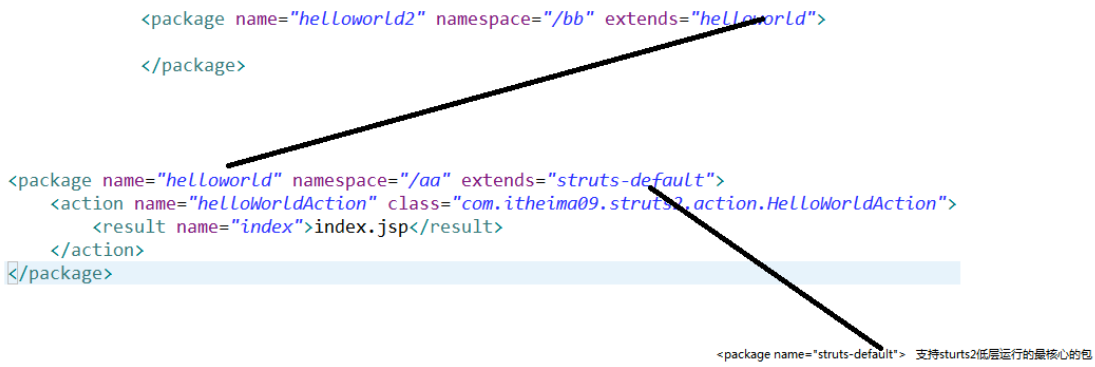


在一个配置文件中：

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

说明 helloworld 拥有 package 的名称为 struts-default 包的所有的功能

案例：



Action

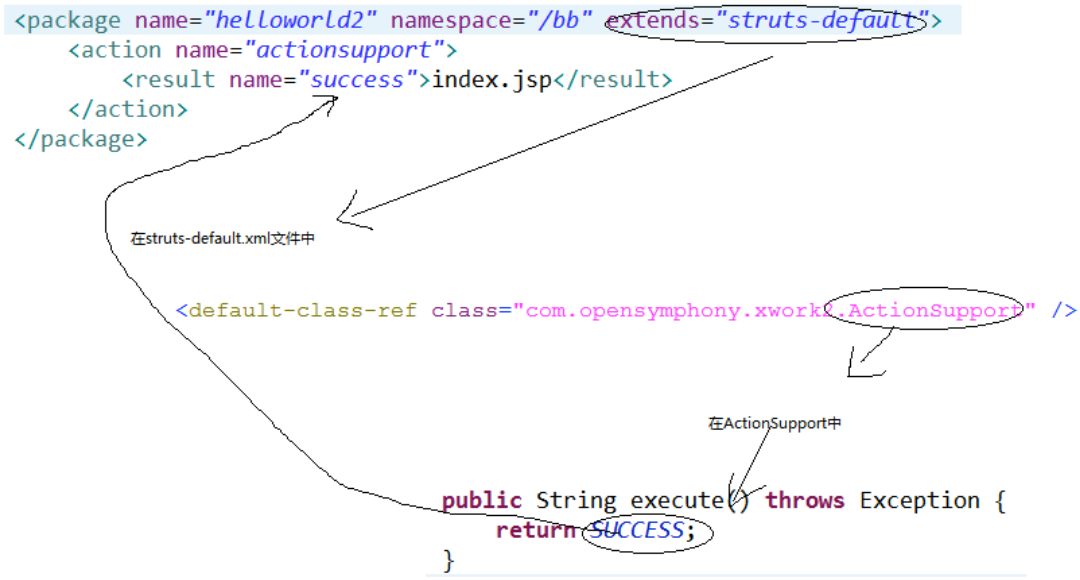
action 元素代表一个类

class 为 action 的类的全名,可以写，也可以不写

如果不写 class 属性，则默认执行 ActionSupport 中的 execute 方法

该方法什么都没有做，仅仅返回了一个 success 字符串

如图：



Result

代表一种结果集

Type 为结果集的类型

Name 属性的值和 action 中某一个方法的返回值一致

type 属性不写，则默认和 struts-default 中的结果集中的 default="true"的结果集保持一致为 dispatcher，转发

result 标签中的内容就是要转发到的页面

在 struts-default.xml 文件中

```
<result-types>
    <result-type name="chain" class="com.opensymphony.xwork2.ActionChainResult"/>
    <result-type name="dispatcher" class="org.apache.struts2.dispatcher.ServletDispatcherResult"
    default="true"/>
    <result-type name="freemarker" class="org.apache.struts2.views.freemarker.FreemarkerResult"/>
    <result-type name="httpheader" class="org.apache.struts2.dispatcher.HttpHeaderResult"/>
    <result-type name="redirect" class="org.apache.struts2.dispatcher.ServletRedirectResult"/>
    <result-type name="redirectAction"
    class="org.apache.struts2.dispatcher.ServletActionRedirectResult"/>
    <result-type name="stream" class="org.apache.struts2.dispatcher.StreamResult"/>
    <result-type name="velocity" class="org.apache.struts2.dispatcher.VelocityResult"/>
    <result-type name="xslt" class="org.apache.struts2.views.xslt.XSLTResult"/>
    <result-type name="plainText" class="org.apache.struts2.dispatcher.PlainTextResult" />
</result-types>
```

Name 属性也可以不写，如果不写，则默认值为”success”

Struts2 基本用法的其他方法

Include

在 struts.xml 文件中

```xml
<struts>
    <include file="struts-helloworld.xml"></include>
</struts>
```

就可以把 struts-helloworld.xml 文件包括进来了

Action 的写法

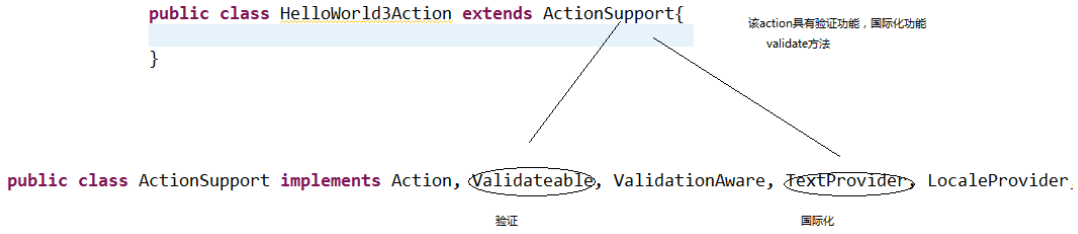
简单的 javabean

```java
public class HelloWorldAction {
    public String execute(){
        System.out.println("hello");
        return "index";
    }
}
```

实现 action 接口

```java
public class HelloWorld2Action implements Action{
    /**
     * 重写execute方法
     */
    public String execute(){
        System.out.println("hello");
        return SUCCESS;
    }
}
```

继承 ActionSupport



Actin 的模式

在 action 的构造器中输出一句话，在浏览器中多次请求，可以看到构造器执行了好几次，所以 action 是多例的。

结果集

```
<result-types>
    <result-type name="chain" class="com.opensymphony.xwork2.ActionChainResult"/>
    <result-type name="dispatcher" class="org.apache.struts2.dispatcher.ServletDispatcherResult"
    default="true"/>
    <result-type name="freemarker" class="org.apache.struts2.views.freemarker.FreemarkerResult"/>
    <result-type name="redirect" class="org.apache.struts2.dispatcher.ServletRedirectResult"/>
    <result-type name="httpheader" class="org.apache.struts2.dispatcher.HttpHeaderResult"/>
    <result-type name="redirectAction"
    class="org.apache.struts2.dispatcher.ServletActionRedirectResult"/>
    <result-type name="stream" class="org.apache.struts2.dispatcher.StreamResult"/>
    <result-type name="velocity" class="org.apache.struts2.dispatcher.VelocityResult"/>
    <result-type name="xslt" class="org.apache.struts2.views.xslt.XSLTResult"/>
    <result-type name="plainText" class="org.apache.struts2.dispatcher.PlainTextResult" />
</result-types>
```

转发

重定向

重定向到action

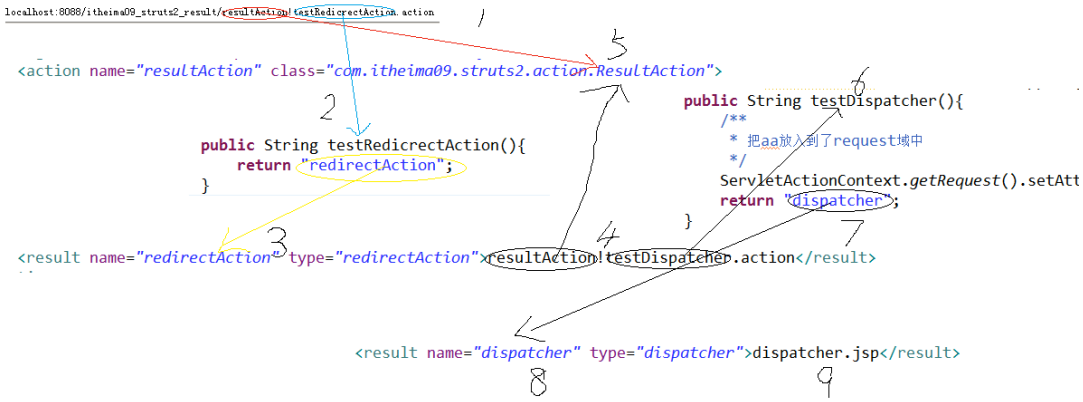
转发

```
<!--
    转发
 -->
<result name="dispatcher" type="dispatcher">dispatcher.jsp</result>
```

重定向

```
<!--
    重定向
 -->
<result name="redirect" type="redirect">redirect.jsp</result>
```

重定向到 action



通配符映射

第一种

```
<package name="urlpattern" namespace="/" extends="struts-default">
    <action name="urlPatternAction" class="com.itheima09.struts2.action.UrlPatternAction">
    </action>
</package>
```

```
localhost:8088/itheima09_struts2_urlpattern/urlPatternAction.action
```

将执行 UrlPatternAction 中的 execute 方法

第二种

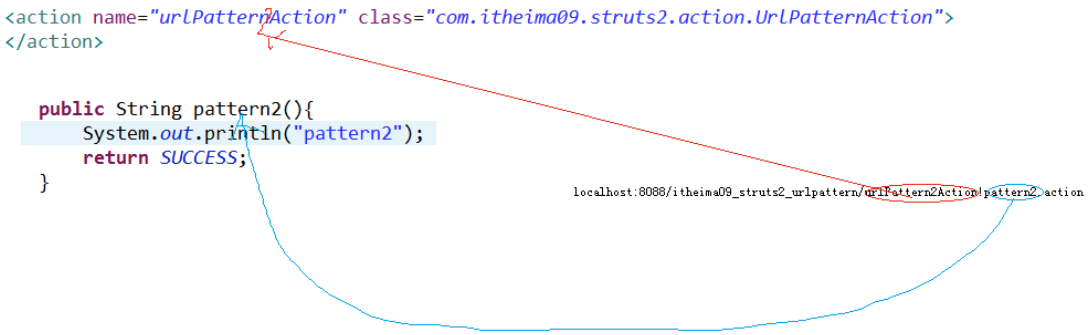
```
<action name="urlPattern2Action" method="pattern1" class="com.itheima09.struts2.action.UrlPatternAction">

</action>
```

```
public String pattern1(){
    System.out.println("pattern1");
    return SUCCESS;
}
```

```
localhost:8088/itheima09_struts2_urlpattern/urlPattern2Action.action
```

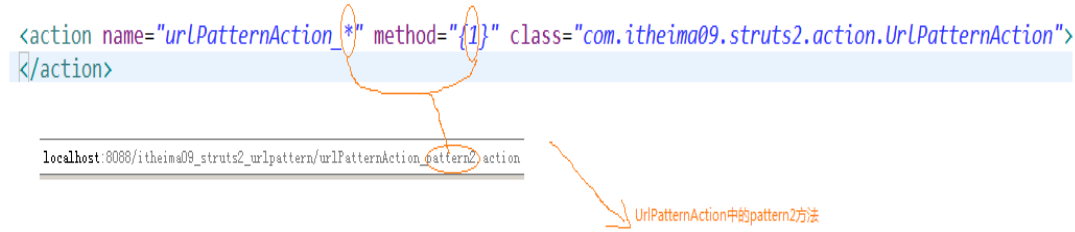
缺点：action 中有几个方法就得在配置文件中写几个 action 元素

第三种



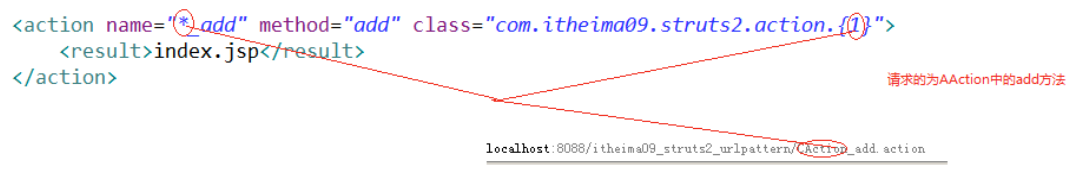
第四种

第五种



该模型的好处：如果在 action 中增加了一个方法，配置文件是不需要改变的，在写 url 时 urlPatternAction_后面的内容变成要请求的方法的名称就可以了

第六种



第七种(不推荐)

```
<action name="*_*" class="com.itheima09.struts2.action.{1}" method="{2}"></action>
```

这么写不好，覆盖范围太大，很有可能出现和其他的 action 的配置冲突的情况

第八种

```
<action name="urlPattern2Action_*" method="{1}" class="com.itheima09.struts2.action.UrlPatternAction">
    <result>{1}.jsp</result>
</action>
```

强制让 url 中的_后面的内容和方法保持一致，跳转到的 jsp 页面的名称和方法的名称也保持一致。这么写带有一定的规范性

Struts2 与 servlet 容器的交互

```
ServletAction
public class ServletAction extends ActionSupport implements ServletContextAware,
    SessionAware,ServletRequestAware{
    private HttpServletRequest request;
    private Map sessionMap;
    private ServletContext servletContext;
```

这种方法可以交互，但是这种方法把 ServletAction 与 servlet 容器耦合性变高了，不利于测试。

```
ServletActionContext
    SF serialVersionUID : long
    SF STRUTS_VALUESTACK_KEY : String
    SF ACTION_MAPPING : String
    C ServletActionContext(Map)
    S getActionContext(HttpServletRequest) : ActionContext
    S getValueStack(HttpServletRequest) : ValueStack
    S getActionMapping() : ActionMapping
    S getPageContext() : PageContext
    S setRequest(HttpServletRequest) : void
    S getRequest() : HttpServletRequest
    S setResponse(HttpServletResponse) : void
    S getResponse() : HttpServletResponse
    S getServletContext() : ServletContext
    S setServletContext(ServletContext) : void
```

可以通过 ServletActionContext 把 servlet 容器相关的类调出来

```
public class ServletAction extends ActionSupport{
    public String testServlet(){
        ServletActionContext.getRequest();
        return SUCCESS;
    }
}
```

该写法使得 action 与 servlet 容器的耦合性不是很强。

总结

1、 sturts2 的配置文件中用了 package 的机制，这样可以分模块
   name 是唯一的名称，extends 采用了继承的机制
2、 写的 action 与 servlet 容器完全松耦合了
3、 通配符映射解决：很容器就把一个 url 映射到一个 action 的方法中了
4、 Include 保证了可以写多个配置文件
5、 结果集的封装

值栈(重要)和 ognl 表达式

1、 只要是一个 mvc 框架，必须解决数据的存和取的问题
2、 Struts2 利用值栈来存数据，所以值栈是一个存储数据的内存结构
3、 把数据存在值栈中，在页面上利用 ognl 表达式显示出来

讨论的问题

1、 讨论值栈的生命周期
2、 值栈的内存结构
3、 通过什么样的方法把数据放入到值栈中
4、 显示出来

值栈的生命周期

一次请求

值栈的内存结构

获取值栈的路径

```
ValueStack valueStack = ActionContext.getContext().getValueStack();
ValueStack valueStack2 = ServletActionContext.getValueStack(ServletActionContext.getRequest());
ValueStack valueStack3 = (ValueStack)ServletActionContext.getRequest().getAttribute("struts.valueStack");
```

```
com.opensymphony.xwork2.ognl.OgnlValueStack@da416
com.opensymphony.xwork2.ognl.OgnlValueStack@da416
com.opensymphony.xwork2.ognl.OgnlValueStack@da416
```

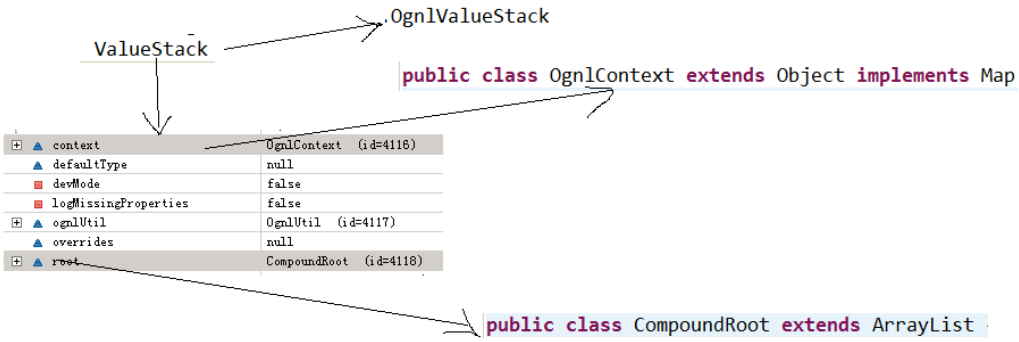
说明：

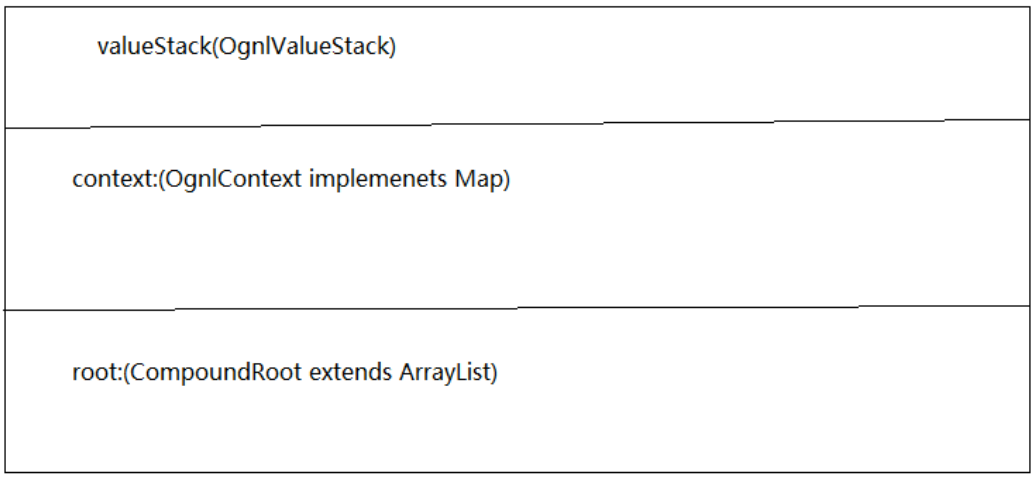
1、 值是一样的，说明只有一个对象

2、 因为有一种是从 request 域中获取的，所以是一次请求

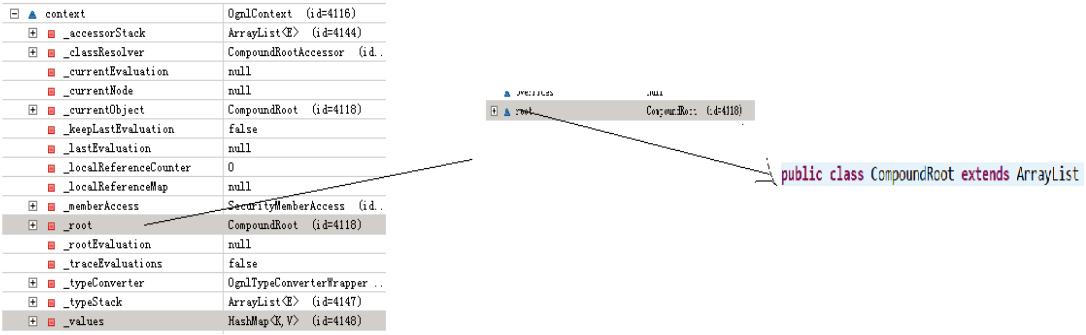
内存结构

1、 大致图：





2、 上面图中的 context 的放大



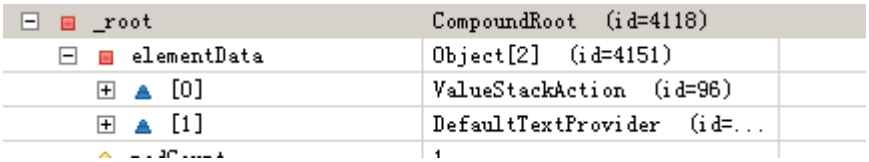
说明：

_root：(CompoundRoot)

_values：(HashMap)

在这里存放了 request，response，session，application 等 servlet 容器的内容

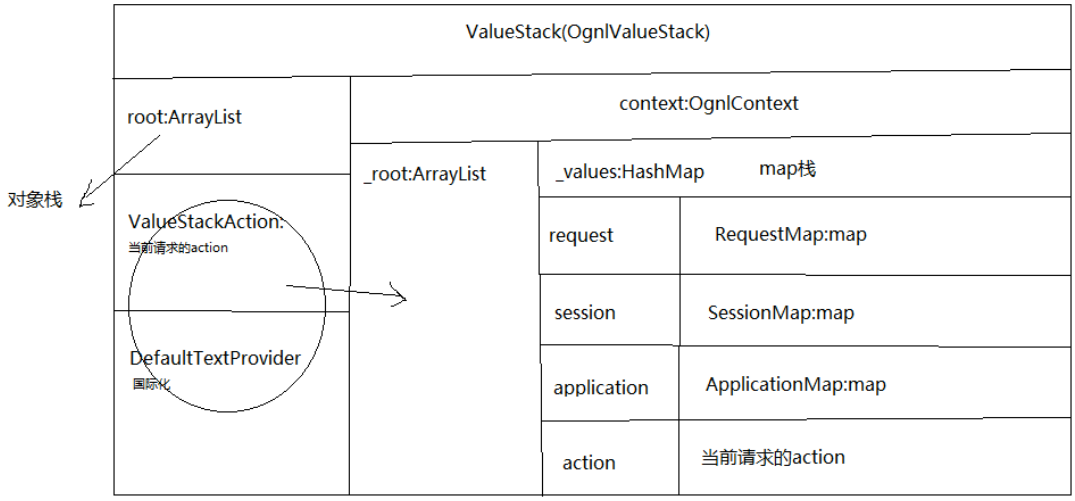
3、_root 的放大



说明：

和 ValueStack 中的 root 是一致的

值栈的内存总图



说明：

从上图中可以看出 valueStack 总共分为两个部分：

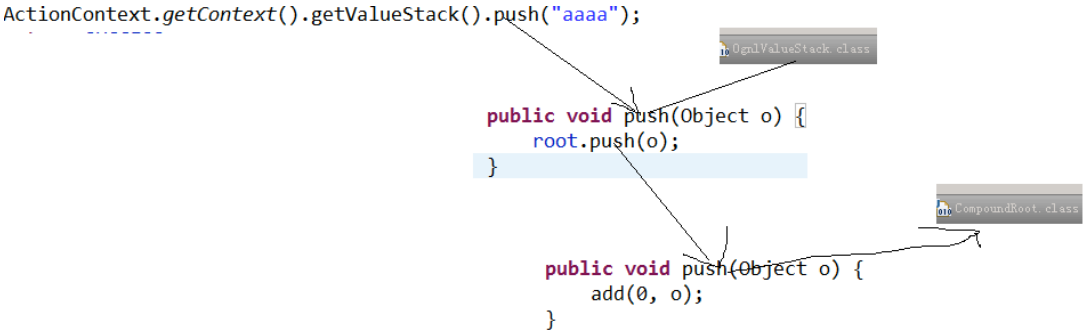
对象栈:root

Map 栈：_values

对象栈的存放

1、 Push 方法

```
public String testPutObjToObjectStack_1(){
    /**
     * 把一个对象放入到对象栈的栈顶
     */
    ActionContext.getContext().getValueStack().push("aaaa");
    return SUCCESS;
}
```

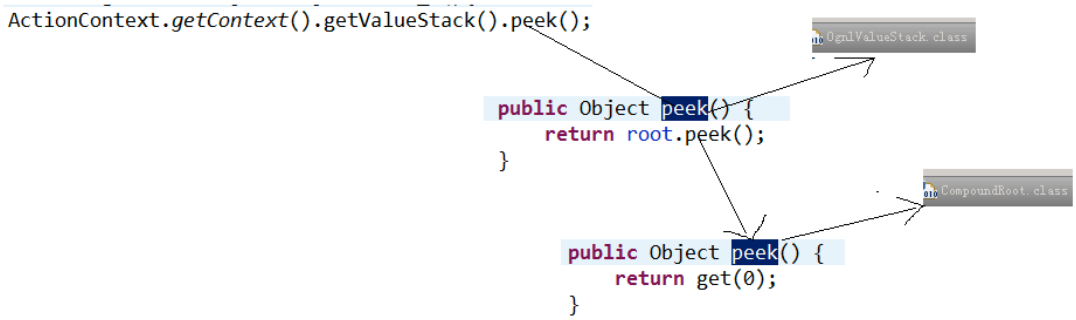


2、 add 方法

```
public String testPutObjToObjectStack_2(){
    /**
     * 把一个对象放入到对象栈的栈顶
     */
    ActionContext.getContext().getValueStack().getRoot().add(0,"aaa");
    return SUCCESS;
}
```

对象栈的提取

```
public String testGetObjFromObjectStack_1(){
    ActionContext.getContext().getValueStack().peek();
    return SUCCESS;
}
```



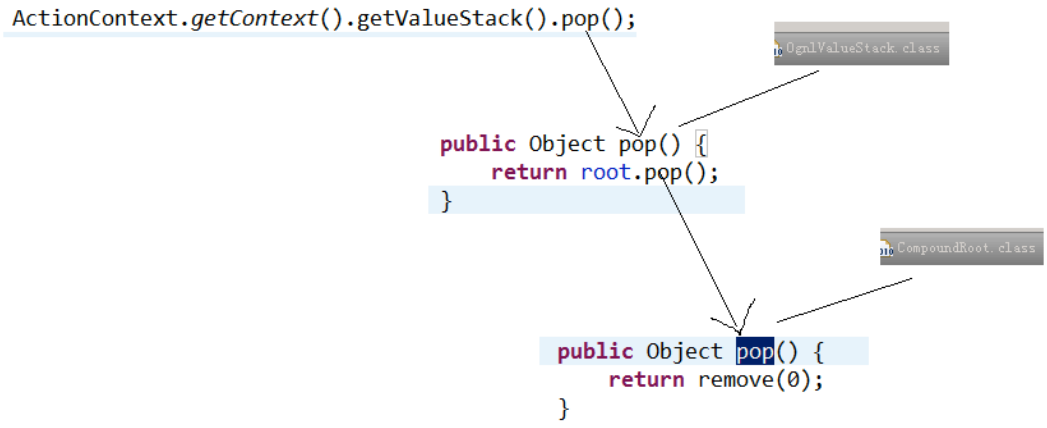
从上图中可以看出，peek 方法为获取对象栈的栈顶的元素

```
ActionContext.getContext().getValueStack().getRoot().get(0);
```

这行代码也可以获取对象栈的栈顶的元素

对象栈的元素的弹出

```
public String testPopObj(){
    ActionContext.getContext().getValueStack().pop();
    return SUCCESS;
}
```



操作对象栈中的对象

```
public String testPutObjectToObjectStack(){
    Person person = new Person();
    person.setPid(1L);
    person.setName("王二麻子");
    //把person放入到了对象栈的栈顶
    ActionContext.getContext().getValueStack().push(person);
    ValueStack valueStack = ActionContext.getContext().getValueStack();
    //通过setParameter方法把对象栈的属性赋值
    valueStack.setParameter("name", "aaaaa");
    //从对象栈中把栈顶的对象提取出来
    person = (Person)ActionContext.getContext().getValueStack().peek();
    System.out.println(person.getName());
    return SUCCESS;
}
```

说明：

可以利用 valueStack.setParameter 方法改变对象栈中对象的属性的值

Map 栈的存放

```
/**
 * 把一个字符串放入到request域中
 */
public String testPutStrToRequest(){
    ServletActionContext.getRequest().setAttribute("request_aaa", "request_aaa");
    ValueStack valueStack = ActionContext.getContext().getValueStack();
    return SUCCESS;
}
```

通过该方法可以把一个字符串放入到 map 栈中的 request 域中

```
/**
 * 把一个字符串放入到application域中
 */
public String testPutStrToApplication(){
    ServletActionContext.getServletContext().setAttribute("application_aaa", "application_aaa");
    ValueStack valueStack = ActionContext.getContext().getValueStack();
    return SUCCESS;
}
```

通过该方法可以把一个字符串放入到 map 栈中的 application 域中

```java
/**
 * 把一个字符串放入到map栈中
 */
public String testPutStrToMap(){
    ActionContext.getContext().put("aaa", "aaa");
    ValueStack valueStack = ActionContext.getContext().getValueStack();
    return SUCCESS;
}
```

通过该方法可以把一个字符串放入到 map 栈中

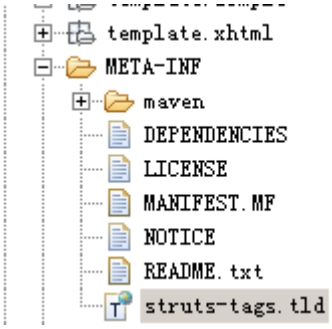
Ognl 表达式

Debug 标签(重要)

在页面上引入标签库

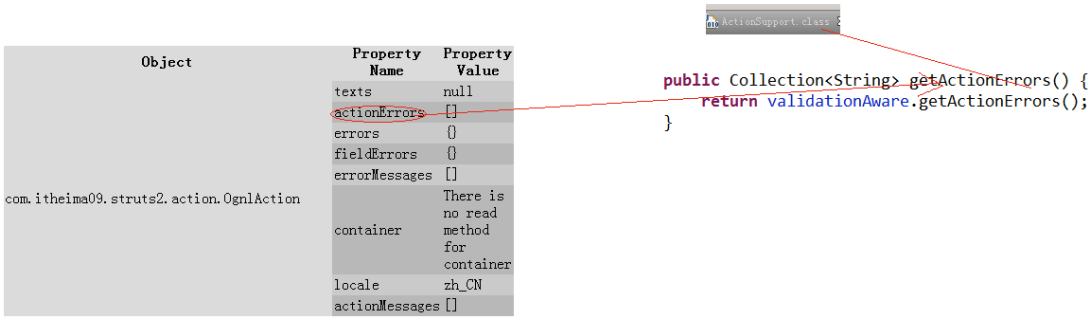
```jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
```

标签库的位置：



可以通过网页的形式把 valueStack 中值提取出来

关于对象栈：



从上图可以看出来，对象栈的属性来自于对象栈中的每一个对象的属性的 get 方法

把一个对象放入到对象栈中

```
Person person = new Person();
person.setName("王二麻子");
person.setPid(1L);
ActionContext.getContext().getValueStack().push(person);
```

在 s:debug 中

com.itheima09.bean.Person	name	王二麻子
	pid	1

如果在对象栈中有两个相同的属性:

```
public class OgnlAction extends ActionSupport{
    private String name;
```

```
public class Person {
    private Long pid;
    private String name;
```

上述的两个类中都同时有 name 属性，而且当前请求的 action 在对象栈中

```
ActionContext.getContext().getValueStack().push(person);
```

利用该方法就把 person 对象放入到栈顶了

```
ActionContext.getContext().getValueStack().setParameter("name", "aaaa");
```

可以利用上述的方法改变对象栈中的 name 属性，但是从上往下找，只要找到一个，改变了值就完事了。

Property 标签(重要)

介绍

输出标签

value 指定要输出的内容

如果在对象栈中，直接指定属性

也可以直接调用方法

如果在 map 栈中，用#指定

如果 value 属性不写，则默认输出栈顶的元素

com.itheima09.bean.Person	name	aaaa
	pid	1

页面上：

```
<s:property value="name"/>
<s:property value="pid"/>
```

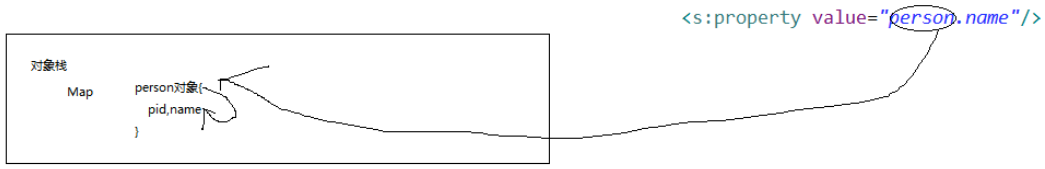
把一个对象放入到对象栈中，可以利用 s:prototype 标签的 value 属性直接访问该对象的属性。如果对象栈中出现两个相同的属性，则 prototype 标签的 value 属性会从栈顶往下查找，直到找到为止。赋值的为第一个。

```
Person person = new Person();
person.setName("王二麻子");
person.setPid(1L);
Map<String, Person> map = new HashMap<String, Person>();
map.put("person", person);
ActionContext.getContext().getValueStack().push(map);
```

把一个对象放入到 map 中，再把 map 放入到对象栈中，在页面上可以利用

```
<s:property value="person.name"/>

<s:property value="person.getName()"/>
<s:property value="person.getName()"/>
<s:property value="person.getName()"/>
```

```
Person person = new Person();
person.setName("王二麻子");
person.setPid(1L);
Map<String, Person> map = new HashMap<String, Person>();
map.put("person", person);
ActionContext.getContext().getValueStack().push(map);
```




```
<s:property value="person.getName()"/>
```

可以利用方法来访问

利用该标签访问 map 栈中的元素

```
<!--
    输出map栈中的key为aaa的元素
 -->
<s:property value="#aaa"/>
<!--
    输出map栈中的key为person对象中的属性name
 -->
<s:property value="#person.name"/>
<!--
    输出map栈中的request域中的person对象的name属性
 -->
<s:property value="#request.person.getName()"/>
```

Iterator 标签(重要)

1、 用途：主要用于迭代，可以迭代 List,Set,Map,Array

2、 案例

1、 把集合放入到对象栈中

```
List<Person> persons = new ArrayList<Person>();
Person person1 = new Person();
person1.setName("王二麻子");
person1.setPid(1L);

Person person2 = new Person();
person2.setName("林志玲");
person2.setPid(1L);

Person person3 = new Person();
person3.setName("王二麻子的哥：王三麻子");
person3.setPid(1L);

persons.add(person1);
persons.add(person2);
persons.add(person3);

ActionContext.getContext().getValueStack().push(persons);
```

```
<s:iterator>
    <!--
        当前迭代的元素在栈顶
     -->
    <s:property value="name"/>
    <s:property value="pid"/>
</s:iterator>
```

2、把 list 放入到 map 中

```
List<Person> persons = new ArrayList<Person>();
Person person1 = new Person();
person1.setName("王二麻子");
person1.setPid(1L);

Person person2 = new Person();
person2.setName("林志玲");
person2.setPid(1L);

Person person3 = new Person();
person3.setName("王二麻子的哥：王三麻子");
person3.setPid(1L);

persons.add(person1);
persons.add(person2);
persons.add(person3);

ActionContext.getContext().put("persons", persons);
```

```
<s:iterator value="#persons">
    <!--
        当前正在迭代的元素在栈顶
     -->
    <s:property value="name"/>
    <s:property value="pid"/>
</s:iterator>
```

3、 把 list 放入到 request 域中

```
List<Person> persons = new ArrayList<Person>();
Person person1 = new Person();
person1.setName("王二麻子");
person1.setPid(1L);

Person person2 = new Person();
person2.setName("林志玲");
person2.setPid(1L);

Person person3 = new Person();
person3.setName("王二麻子的哥：王三麻子");
person3.setPid(1L);

persons.add(person1);
persons.add(person2);
persons.add(person3);

ServletActionContext.getRequest().setAttribute("persons", persons);
```

```
<s:iterator value="#request.persons">
    <s:property value="name"/>
    <s:property value="pid"/>
</s:iterator>
```

4、 把 map 放入到 map 栈中

```
Map<String, Person> persons = new HashMap<String, Person>();
Person person1 = new Person();
person1.setName("王二麻子");
person1.setPid(1L);

Person person2 = new Person();
person2.setName("林志玲");
person2.setPid(1L);

Person person3 = new Person();
person3.setName("王二麻子的哥：王三麻子");
person3.setPid(1L);

persons.put("person1", person1);
persons.put("person2", person2);
persons.put("person3", person3);

ActionContext.getContext().put("persons", persons);
```

```
<!--
    Map<String,Person>
    如果迭代的元素是map,则当前正在迭代的对象是entry
 -->
<s:iterator value="#persons">
    <s:property value="key"/>
    <s:property value="value.name"/>
    <s:property value="value.pid"/>
</s:iterator>
```

5、把 List<Map<String,Person>>放入到 map 栈中

```
List<Map<String, Person>> list = new ArrayList<Map<String, Person>>();
Map<String, Person> persons = new HashMap<String, Person>();
Person person1 = new Person();
person1.setName("王二麻子");
person1.setPid(1L);

Person person2 = new Person();
person2.setName("林志玲");
person2.setPid(1L);

Person person3 = new Person();
person3.setName("王二麻子的哥：王三麻子");
person3.setPid(1L);

persons.put("person1", person1);
persons.put("person2", person2);
persons.put("person3", person3);
list.add(persons);
ActionContext.getContext().put("list", list);
```

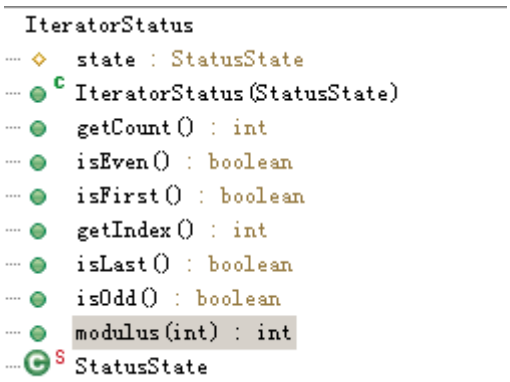
```
<!--
    List<Map<String,Person>>
 -->
<!--
    list
 -->
<s:iterator value="#list">
    <!--
        map
     -->
    <s:iterator>
        <!--
            entry
         -->
         <s:property value="key"/>
         <s:property value="value.name"/>
         <s:property value="value.pid"/>
    </s:iterator>
</s:iterator>
```

总结：

1、 如果要迭代一个集合，当前迭代的元素在栈顶

2、 如果要迭代一个 map，则当前迭代的元素是 entry

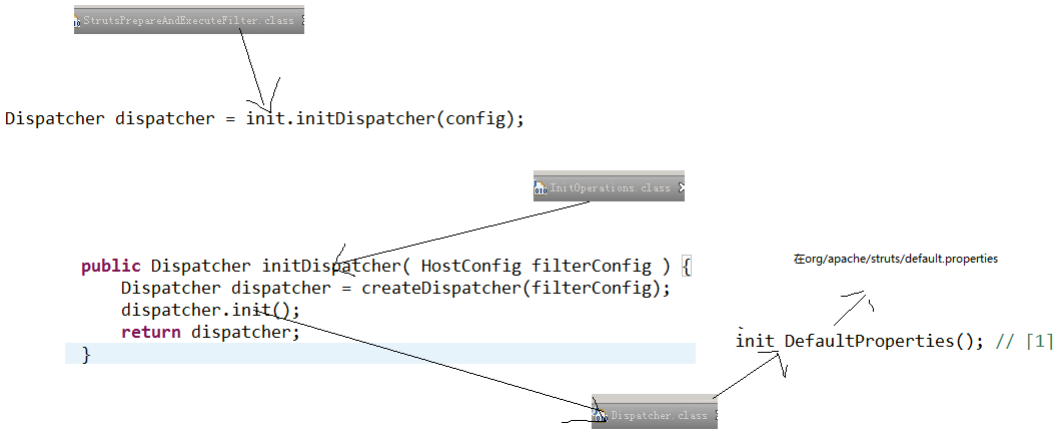
3、 Iterator 标签中有一个属性 status



Ui 标签

1、 struts2 提供了一套标签机制，由 struts2 框架把 struts2 的标签解析成 html 标签。

2、 在 struts2 的配置文件中，增加



在 struts2 的配置文件中加入如下的内容：

```
<constant name="struts.ui.theme" value="simple"></constant>
```

由 struts2 容器翻译过程来的 html 代码和 struts2 标签的代码就能保持一致。

Select 标签

```
public String testSelect(){
    List<Province> provinces = new ArrayList<Province>();
    Province province1 = new Province();
    province1.setPid(1L);
    province1.setName("山西省");
    Province province2 = new Province();
    province2.setPid(2L);
    province2.setName("河南省");

    Province province3 = new Province();
    province3.setPid(3L);
    province3.setName("四川省");

    provinces.add(province1);
    provinces.add(province2);
    provinces.add(province3);

    this.pid = 3L;

    ActionContext.getContext().put("provinces", provinces);
    return "ui";
}
```

```
<td><s:select list="#provinces" listKey="pid" listValue="name"/></td>
<!--
    listKey  为option元素的value属性的值
    listValue  option元素的文本内容
    name
      该属性的值用来获取选中的元素
      用于回显
 -->
```

Checkbox:

```
public String testCheckbox(){
    List<Hobby> hobbies = new ArrayList<Hobby>();
    Hobby hobby1 = new Hobby();
    hobby1.setHid(1L);
    hobby1.setName("足球");
    Hobby hobby2 = new Hobby();
    hobby2.setHid(2L);
    hobby2.setName("蓝球");

    Hobby hobby3 = new Hobby();
    hobby3.setHid(3L);
    hobby3.setName("荷兰球");

    hobbies.add(hobby1);
    hobbies.add(hobby2);
    hobbies.add(hobby3);

    ActionContext.getContext().put("hobbies", hobbies);

    return "ui";
}
<!--
    list代表数据源
    listKey为value值
    listValue为显示的值
 -->
<td><s:checkboxlist list="#hobbies" listKey="hid" listValue="name" name="hid"/></td>
```

回显

1、 如果把要回显的数据放入到 map 栈中，则必须根据 value 进行回显
Value=”%{ognl 表达式}”

```
如果要回显的内容在map栈中，不能根据name属性进行回显,必须用value属性进行回显
    <td><s:textfield name="username" value="%{#user.username}"></s:textfield></td>
```

2、 一般情况下，应该把回显的数据放入到对象栈中，因为页面上可以直接根据 name

进行回显

```
如果把要回显的对象放入到对象栈中，则页面上可以根据name进行回显
    <s:textfield name="username"></s:textfield>
```

3、 因为当前请求的 action 在对象栈中，所以可以利用 action 中的属性进行回显，action 中的属性必须有 get 方法。

4、 Checkbox 的回显

```
/**
 * 用于回显
 */
this.hids = new Long[2];
this.hids[0] = 1L;
this.hids[1] = 2L;
return "backview";
```

上述图表示要回显的 id 值

```
<!--
    用name进行回显
 -->
<td><s:checkboxlist list="#hobbies" listKey="hid" listValue="name" name="hids"/></td>
```

根据 name 属性进行回显

拦截器

需求

如果要访问某一个 action 类中的某一个方法的内容，如果是 admin 用户，则访问，如果不是 admin 用户，则不能访问。

实现

```
public class PrivilegeAction extends ActionSupport{
    public String testPrivilege(){
        String username = ServletActionContext.getRequest().getParameter("username");
        if("admin".equals(username)){//权限判断
            ActionContext.getContext().getValueStack().push("正确的访问");//业务逻辑
        }else{
            ActionContext.getContext().getValueStack().push("权限不足，没有办法访问");
        }
        return "privilege";
    }
}
```

步骤

1、 编写 interceptor

```
public class PrivilegeInterceptor implements Interceptor{
        @Override
        public String intercept(ActionInvocation invocation) throws Exception {
                String username = ServletActionContext.getRequest().getParameter("username");
                if("admin".equals(username)){
                        return invocation.invoke();
                }else{
                        ActionContext.getContext().getValueStack().push("权限不足，没有办法访问");
                        return "error";
                }
        }

}
```

返回到当前请求的action中

struts2配置文件中result所指向的页面

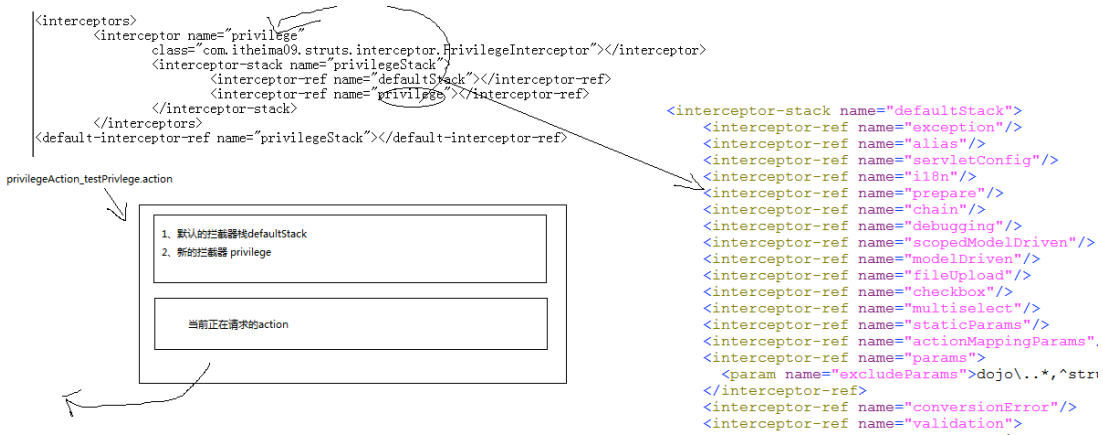
2、 写一个 PvilegeSuperAction

```
public class PrivilegeSuperAction extends ActionSupport{
    public String testPrivilege(){
        ActionContext.getContext().getValueStack().push("正确的访问");//业务逻辑
        return "privilege";
    }
}
```

3、 在配置文件中进行配置

```
<interceptors>
    <!--
        声明一个拦截器
     -->
    <interceptor name="privilege"
        class="com.itheima09.struts.interceptor.PrivilegeInterceptor"></interceptor>
    <!--
        声明一个拦截器栈
     -->
    <interceptor-stack name="privilegeStack">
        <interceptor-ref name="defaultStack"></interceptor-ref>
        <interceptor-ref name="privilege"></interceptor-ref>
    </interceptor-stack>
</interceptors>
<default-interceptor-ref name="privilegeStack"></default-interceptor-ref>
```

详细解析



说明：

通过该图可以看出：当提交一个 url 请求时，先执行所有的拦截器(按照声明的从上到下的顺序)，再执行 action。

声明一个拦截器

```
<interceptor name="privilege"
    class="com.itheima09.struts.interceptor.PrivilegeInterceptor"></interceptor>
```

声明一个拦截器栈

```
<interceptor-stack name="privilegeStack">
    <interceptor-ref name="defaultStack"></interceptor-ref>
    <interceptor-ref name="privilege"></interceptor-ref>
</interceptor-stack>
```

既可以引用一个拦截器

```
<interceptor-ref name="privilege"></interceptor-ref>
```

也可以引用一个拦截器栈

```
<interceptor-ref name="defaultStack"></interceptor-ref>
```

真正的做法：

```
<package name="privilege" namespace="/" extends="struts-default">
    <interceptors>
        <interceptor name="privilege" class="com.itheima09.struts.interceptor.PrivilegeInterceptor">
        </interceptor>
        <interceptor-stack name="privilegeStack">
            <interceptor-ref name="defaultStack"></interceptor-ref>
            <interceptor-ref name="privilege"></interceptor-ref>
        </interceptor-stack>
    </interceptors>
    <default-interceptor-ref name="privilegeStack"></default-interceptor-ref>
</package>
```

```
<package name="p" namespace="/" extends="privilege">
```

如果 p 包继承了 privilege 包就把所有的新的拦截器栈继承过来了，如果没有继承，则执行默认的拦截器栈

缺点

把权限判断的代码和业务逻辑的代码混合在一起了

利用拦截器改进

拦截器的优点

1、 把项目中一些经常用到的业务逻辑从 action 中分离出来，写到拦截器中
这样可以做到这些常用的逻辑和真正的逻辑的松耦合
2、 拦截器和 action 是真正的松耦合了
3、 Struts2 把开发过程中的一些常用的业务已经封装到各个拦截器中了

案例 2

在执行 action 的方法之前，
1、 开启日志
2、 权限的检查

```
<interceptor name="privilege" class="com.itheima09.struts.interceptor.PrivilegeInterceptor">
</interceptor>
<interceptor name="logger" class="com.itheima09.struts.interceptor.LoggerInterceptor"></interceptor>
```

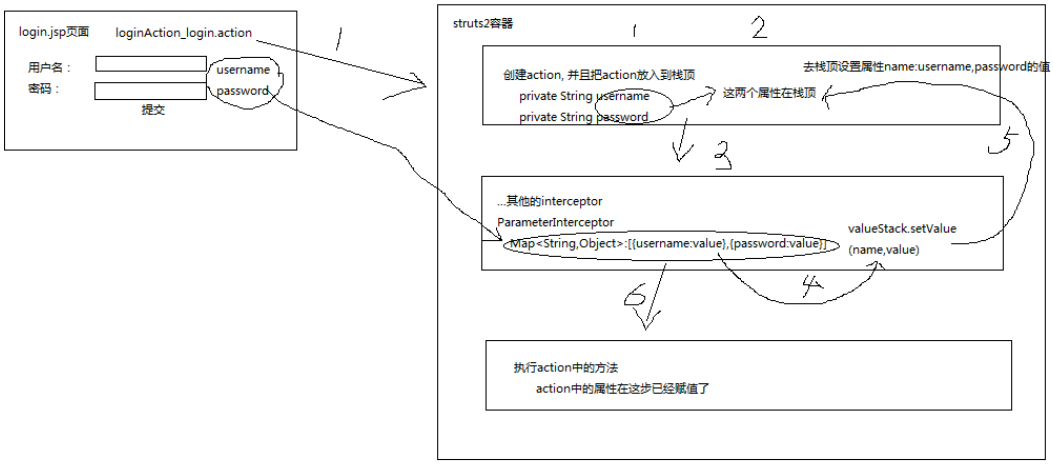
声明两个拦截器

```
<interceptor-ref name="logger"></interceptor-ref>
<interceptor-ref name="privilege"></interceptor-ref>
```

在执行的拦截器栈中，按照执行的先后顺序引入拦截器

拦截器的应用

属性驱动



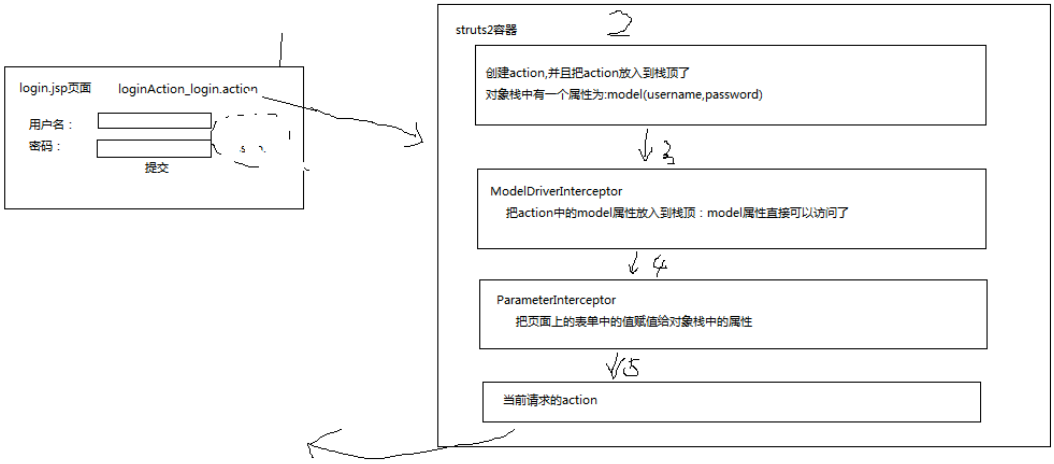
注意事项：

1、 必须使用 struts2 默认的拦截器栈中的 ParametersInterceptor

2、 Action 中的属性和表单中的 name 属性的值保持一致

3、 利用 valueStack.setValue 方法可以赋值了

模型驱动




```
@Override
public String intercept(ActionInvocation invocation) throws Exception {
    Object action = invocation.getAction();

    if (action instanceof ModelDriven) {
        ModelDriven modelDriven = (ModelDriven) action;
        ValueStack stack = invocation.getStack();
        Object model = modelDriven.getModel();
        if (model !=  null) {
            stack.push(model);
        }
        if (refreshModelBeforeResult) {
            invocation.addPreResultListener(new RefreshModelBeforeResult(modelDriven, model));
        }
    }
    return invocation.invoke();
}
```

要想使用模型驱动，则action必须实现ModelDriven接口

该拦截器的作用是把模型驱动对象放入到栈顶

从上图可以看出，ModelDriverInterceptor 有两个作用：

1、 当前请求的 action 必须实现 ModelDriver 接口

2、 把 model 对象放入到了栈顶

Threadlocal

1、 本地线程类

2、 可以存放一个对象

3、 Threadlocal 对象只能当前线程访问，其他的线程是不能访问的

4、 传递参数

案例：

参数的传递

```
public class TL2 {
    public void tl2(String s){
        System.out.println(s);
    }
}
```

```
public class TL1 {
    public static void tl1(String s){
        TL2 tl2 = new TL2();
        tl2.tl2(s);
    }

    @Test
    public void test(){
        TL1.tl1("aaa");
    }
}
```

说明：通过参数的传递，一个字符串很容易的从客户端传递到 tl1 中的 tl1 方法，再传递到 tl2 中的 tl2 方法

```
public class TL1Super {
    public static void tl1(){
        System.out.println(ThreadLocalStr.getStr());
    }

    @Test
    public void test(){
        TL1Super.tl1();
        TL2Super.tl2();
    }
}
```

```
public class TL2Super {
    public static void tl2(){
        System.out.println(ThreadLocalStr.getStr());
    }
}
```

说明：

在 TL1Super 中用到一个字符串

在 TL2Super 中用到同样的字符串，但是这两个类是完全松耦合，这个时候要用 Threadlocal 传递数据，因为不管是否是送耦合的，但是可以肯定在同一个线程中。所以可以得到同一个线程 threadlocal 对象中的数据