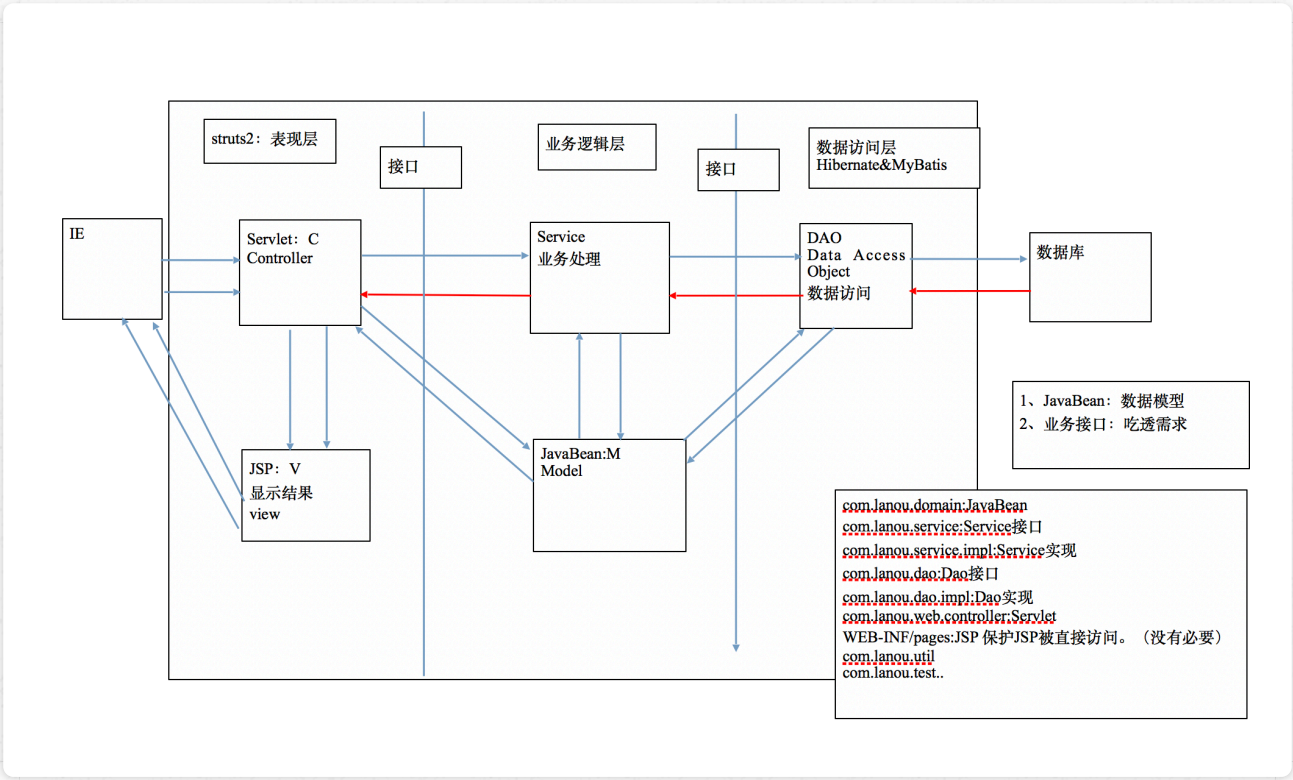
Struts2

1. Struts 历史简介

Struts是一个JavaWeb框架, Struts2并不是在Struts1基础上改进的一个框架， 而是基于WebWork的全新框架。Struts2是基于MVC开发模型的一个框架, 基于表现层框架.



2. Struts2的安装与配置

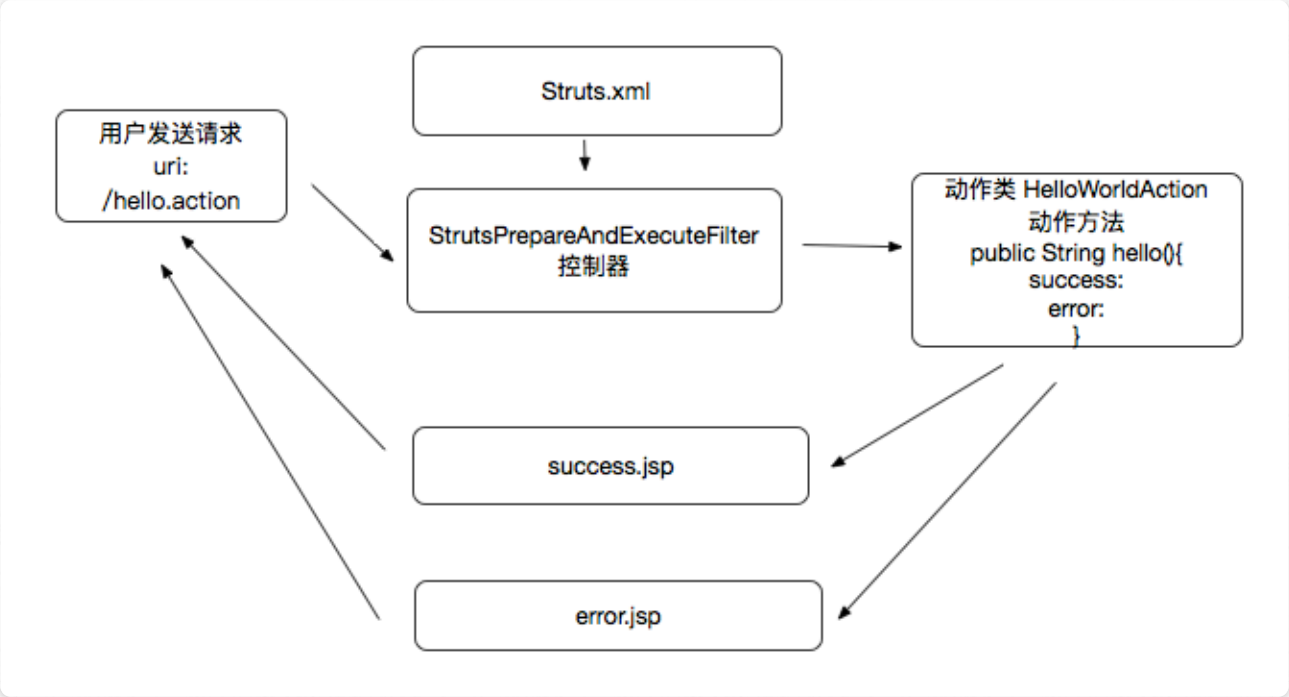
1.下载 Struts2 的发行包 http://struts.apache.org
2. 新建工程, 拷贝 jar 包
[小技巧: `struts-**/apps/struts2-blank/lib` 中的所有] 自己的版本考自己版本 jar 包
3. 构建路径的顶端, 创建配置文件 `struts.xml`
4. 配置控制器映射: 框架提供 配置过滤器 `StrutsPrepareAndExecuteFilter`

5. 部署查看

`struts2-core-*.jar` --- struts2的核心包

3. struts2的执行原理及流程

3.1 执行图



3.2 第一个案例

jsp:

```
<a href="${pageContext.request.contextPath}/hello.action">点击这里才能发现大老王的秘密</a>
```

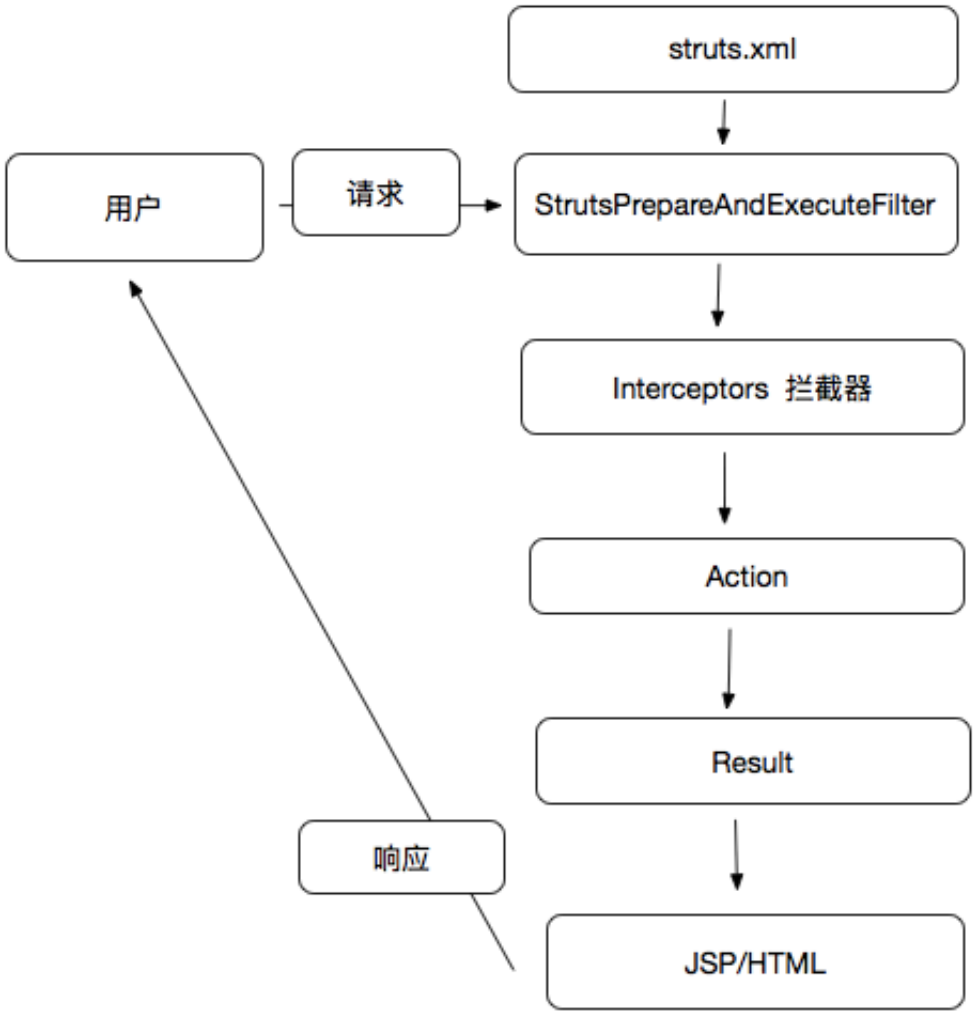
struts.xml:

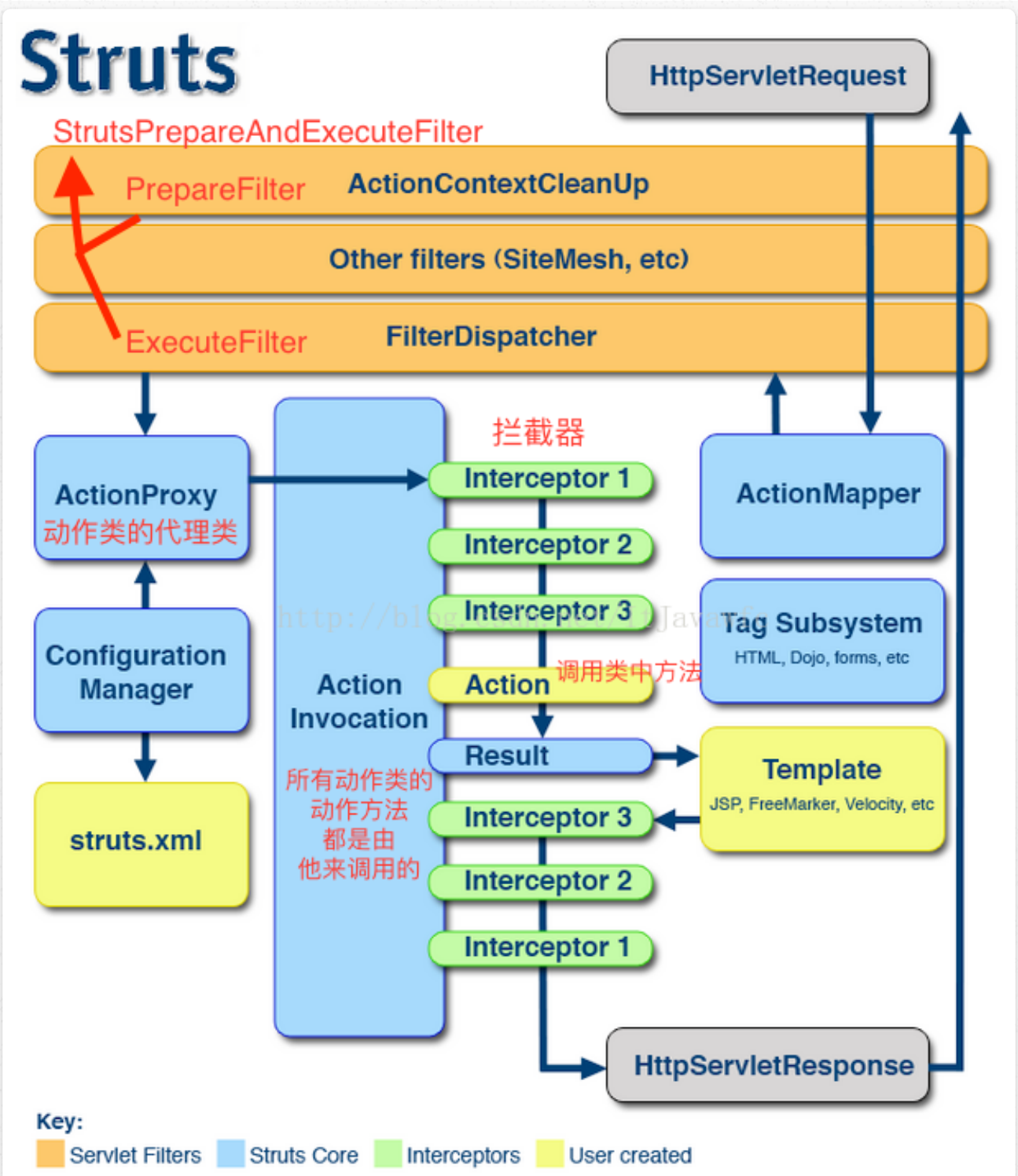
```
<struts>
    <package name="p1" extends="struts-default">
        <action name="helloworld" class="com.lanou.action.helloworld.HelloAction" method="sayHello">
            <result name="success">/success.jsp</result>
```

```
            <result name="error">/error.jsp</result>
        </action>
    </package>
</struts>
```

action:

```
public class HelloAction {
    public String sayHello(){
        System.out.println("动作方法执行了");
        return "success";
    }
}
```





3.3 拦截器的存在

刚才已经使用了很多拦截器. 查看 struts-default.xml 找到拦截器.
查看 default-stack
验证拦截器的存在, 任意找一个打断点. debug

4. Struts2 的配置文件

4.1 配置文件加载顺序

1. 配置文件加载顺序
   a、default.properties : `struts2-core**.jar org.apache.struts` 包 [只看]
   b、struts-default.xml : `struts2-core**.jar` [只看]
   c、struts-plugin.xml : 在插件的jar包 [只看]
   d、struts.xml : 在应用的构建路径顶端。自己定义的Struts配置文件（推荐）
   e、struts.properties : 在应用的构建路径顶端。程序员可以编写（不推荐）
   f、web.xml: 配置过滤器时，指定参数。程序员可以编写（不推荐）

   * 注意：顺序是固定的。后面的配置会覆盖前面的同名配置信息。

4.1.1 代码验证配置文件

需求：Struts2 的默认请求路径是 XXX.action , 改成 `***.do`
此处的默认配置在default.properties中

1. struts.xml
   ```
   <constant name="struts.action.extension" value="do,,"/>
   ```
2. struts.properties
   ```
   struts.action.extension=doo
   ```
3. web.xml
   ```
   <init-param>
       <param-name>struts.action.extension</param-name>
       <param-value>dooo</param-value>
   </init-param>
   ```

4.2 配置文件中的常用标签

4.2.1 constant

```
• 指定处理请求后缀 :
<constant name="struts.action.extension" value="do,,"></constant>
```

- 开发模式:

```
<!--开发模式: 当打开它, xml.reload 也会变成true, 并且还会打印更详细的错误信息. 看default.properties-->
<constant name="struts.devMode" value="true"></constant>
```

- 指定默认编码集:

```
<constant name="struts.i18n.encoding" value="utf-8"></constant>
```

- 上传文件大小限制:

```
<constant name="struts.multipart.maxSize" value="10701096"></constant>
```

4.2.2 package 元素

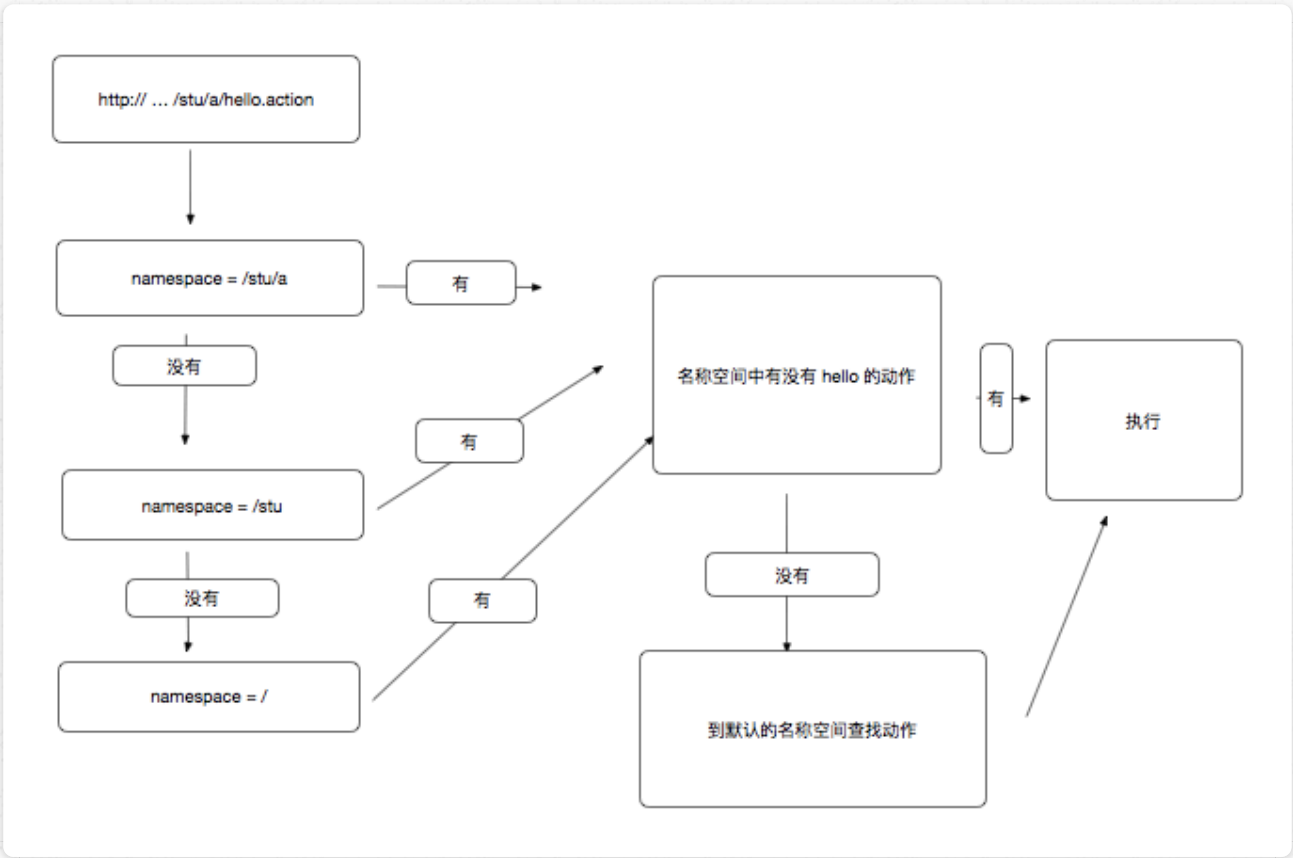
意义:分模块开发
属性:

- name: 必须要有的. 配置文件中要唯一, 一个名字
- extends: 不是必须的. 指定父包. 会把父包中的配置内容继承下来. 一般需要直接会间接的继承一个叫做 “ struts-default “的包(该包在在 struts-default.xml 配置文件中), 如果不继承该包, 那么 struts2 中的核心功能无法使用.

- abstract: 是否是抽象包. 没有任何 action 子元素的 package 可以声明为抽象包.

- namespace: 指定名称空间. 一般以 “/“ 开头 . 该包中的动作访问路径: namespace+动作名称 . 如果 namespace=““, 这是默认名称空间和不写该属性是一样的. 但与namespace=“/“ 不同

有关命名空间的访问



4.2.3 action 元素

作用: 定义一个动作
属性:

```
• name: 必须的. 动作名称, 用户用于发起请求, 在包中要唯一.
• class: 指定动作类的全名. 框架会通过反射机制去实例化. 不写 默认是 com.opensymph
ony.xwork2.ActionSupport .
• method: 指定动作类中的动作方法. 框架会执行该方法. 默认是 execute(). 而execute
() 方法默认返回 success
```

4.2.4 关于默认类

struts-default.xml 中进行了声明

```
<default-class-ref class="com.opensymphony.xwork2.ActionSupport"/>
```

我们可以在 struts.xml 中覆盖它

```
<default-class-ref class="com.lanou.action.HelloworldAction"></default-
class-ref>
```

4.2.5 包含其他配置文件

所有设置内容可以分开几个 xml 文件写, 不必都写在一个 struts.xml 中

```
<include file="user.xml"></include>
```

5. 动作类:

5.1 编写动作类的三种方式

5.1.1 POJO (Plain Old Java Object)

普通的javaBean, 没有任何继承

5.1.2 实现 Action 接口

实现 com.opensymphony.xwork2.Action 接口,
可以使用接口中的常量:

```
    • String SUCCESS :  success  动作执行正常
    • String NONE : none  动作方法执行后不转向任何视图. 或者在动作方法后 null
    • String ERROR : error. 有错误
    • String INPUT : input. 验证, 转换失败, 转入输入页面
    • String LOGIN : login. 检测用户是否登录, 没有登录转向次视图.
```

5.1.3 继承 ActionSupport : [推荐使用]

功能强大. 已经实现了 Action, 还有验证,国际化文本等都可以使用.

5.2 动作类中的动作方法编写要求

```
public String XXX(){}
```

5.3 使用通配符配置 action

在配置 元素时, 允许在指定 name 属性时, 使用模式字符串 (用 “ * “ 代表一个或多个字符)

```
<!-- {1}取*匹配的内容-->

<action name="*Customer" class="com.lanou.action.CustomerAction" method
="{1}">
    <result name="success">/customer/{1}Customer.jsp</result>
</action>
```

6.Action访问 ServletAPI

6.1 方式一： ServletActionContext

```
HttpServletRequest request = ServletActionContext.getRequest();
```

6.2 方式二： 实现 XXXAware接口

ServletRequestAware 等
原因: 在执行动作之前有拦截器给你注入这些实例 interceptor — servletcofig

---

1. 结果视图 Result

1.1 全局结果视图[global]和局部结果视图

- 局部结果视图: 将 result 作为 的子元素配置. 服务于当前的动作

```
<package name="p1" extends="struts-default">
    <action name="act1" class="com.lanou.action.Demo1Action" method="ac
t1">
        <result name="success" type="dispatcher">/success.jsp</result>
    </action>
</package>
```

- 全局结果视图 : 将 result 作为 的子元素配置. 当前包使用

```
<package name="p1" extends="struts-default">
    <!-- 全局视图: 访问动作时, 如果没有局部视图, 则查找全局视图 -->
<global-results>
        <result name="success" type="dispatcher">success.jsp</result>

</global-results>
  <action name="act1" class="com.lanou.action.Demo1Action" method="act1
">
  </action>
</package>
```

思考: 不同包中的视图都能使用这个结果视图?

1.2 result 元素的配置

属性:

```
• name: 指定配置逻辑视图名, 对应着动作方法的返回值. Name 默认值: success

• type: 到达目标的形式.
默认值: dispatcher 转发
```

1.3 Struts2 提供的结果集类型

```
1.  chain : 用户转发到另外一个动作
2.  dispatcher: 转发到 jsp 页面
3.  redirect: 用于重定向到另外一个 JSP 页面
4.  redirectAction: 用于重定向到另外一个action
5.  stream: 用于向浏览器返回一个InputStream的结果类型. 用于文件下载
6.  plainText: 用于显示某个页面原始代码的结果类型 [直接访问页面]
7.  ... ...
```

思考: 不同包中两个 action 的转发. 默认名称空间? 有 namespace?

1.4 自定义结果视图

需求: 随机验证码图片

jsp:

```
<form action="">
    username: <input><br>
    password: <input><br>
    验证码: <input name="code" size="4"><img src="${pageContext.request.
contextPath}/codeImg.action"><br>
    <input type="submit" value="登录">
</form>
```

1. 间接或直接实现 com.opensymphony.xwork2.Result接口的 doExecute 方法

```
public class CodeImgResult implements Result {
    @Override
    public void execute(ActionInvocation actionInvocation) throws Excep
tion {
        VerifyCode code = new VerifyCode();
        BufferedImage img = code.getImage();
        System.out.println("code: " + code.getText());
        VerifyCode.output(img, ServletActionContext.getResponse().getOu
tputStream());
    }
}
```

1. 声明结果类型. 这个是在包中可以使用, 若想全局, 使用包继承

```
<!-- 自定义结果集类型 -->
<package name="customresult" extends="struts-default">
<!-- 声明结果集类型-->
<result-types>
    <result-type name="codeimg" class="com.lanou.customresult.CodeImg
Result"></result-type>
</result-types>

<action name="codeImg">
    <!-- 使用 自定义的结果集类型-->
    <result type="codeimg" name="success"></result>
</action>
</package>
```

2. 数据封装

作为 MVC 框架,必须要负责解析Http请求参数,并将其封装到Model 对象中
Struts2 提供了非常强大的类型转换机制用于请求数据封装到model对象.

2.1 静态参数封装(了解)

action获取struts.xml中的参数

struts.xml

```
<package name="p1" extends="struts-default">
    <action name="staticParam" class="com.lanou.struts.staticparam.Stat
icParamAction" method="login">
        <param name="username">大老王</param>
        <param name="password">123456</param>
    </action>
</package>
```

action:

```
public class StaticParamAction extends ActionSupport {
        private String username;
        private String password;

        public String login() {
            System.out.println("动作方法执行了, 大老王 XX 照曝光地址");
            System.out.println(username + ":" + password);
            return null;
        }

        public String getUsername() {
```

```
            return username;
        }

        public void setUsername(String username) {
            this.username = username;
        }

        public String getPassword() {
            return password;
        }

        public void setPassword(String password) {
            this.password = password;
        }
}
```

底层依靠 staticParams 拦截器, 将静态参数放入值栈中，而action就在值栈中的栈顶，自然就会找到该action中的对应属性，然后进行赋值.

2.2 动态参数封装

2.2.1 属性驱动

2.2.1.1 普通属性驱动，提供get、set方法

action:

```
public class UserAction extends ActionSupport {
    private String username;
    private String password;

    public String login(){
        System.out.println("u: " +username+ " ,p: " +password);
    return null;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }
```

```
    public void setPassword(String password) {
        this.password = password;
    }
}
```

jsp:

```
<form action="${pageContext.request.contextPath}/login1.action">
    <input type="text" name="username">
    <input type="password" name="password">
<input type="submit" value="登录">
</form>
```

跟静态参数封装一样，只不过这里获取的是表单中的参数，也就是请求发送过来的数据。依靠的拦截器为params. 其中该拦截器做的事有两件，一是对提交的参数进行数据校验，判断是否合法，判断是否合法的标准就是拦截器中的excludeParams参数的正则表达式的值。二是将其封装到值栈中的栈顶元素中去，而当前action就在栈顶，所以能够将参数放入action中。

2.2.1.2 ognl表达式来封装数据

action:

```
public class PersonAction extends ActionSupport {
    private Person person;

    public Person getPerson() {
        System.out.println("per.GET");
        return person;
    }

    public void setPerson(Person person) {
        System.out.println("per.SET");
        this.person = person;
    }

    public String login(){
        System.out.println(person);
        return null;
    }
}
```

jsp:

```
<form action="${pageContext.request.contextPath}/login2.action">
    用户名:  <input type="text" name="person.username"><br>
    密码:  <input type="password" name="person.password"><br>
    性别: <input type="text" name="person.gender"><br>
    <input type="submit" value="登录">
</form>
```

在jsp页面中的 person.username 和person.password 其实就是ognl表达式，代表着往根(root，值栈valueStack)中存放值，而值栈中的栈顶元素也就是为当前action，我们在action中设置person的get、set属性，即可以让存进来的值匹配到，进而将对应属性赋值成功。

2.2.2 模型驱动

action:

```
public class CustomerAction extends ActionSupport implements ModelDrive
n<Customer> {
    private Customer customer = new Customer();

    public Customer getCustomer() {
        return customer;
    }
    public void setCustomer(Customer customer) {
        this.customer = customer;
    }

    public String login(){
        System.out.println(customer);
        return null;
    }

    @Override
    public Customer getModel() {
        return customer;
    }
}
```

jsp:

```
<form action="${pageContext.request.contextPath}/login3.action">
  用户名:  <input type="text" name="username"><br>
  密码:  <input type="password" name="password"><br>
```

```
    <input type="submit" value="登录">
</form>
```

必须实现 ModelDriven 接口，提供一个方法getModel()，初始化对象实例。

modelDriven 拦截器将getModel方法返回的结果压入值栈，而我们的表单参数会从值栈中从上往下进行查找，自然就直接将参数封装到对象中了。

作业: 封装复杂对象 List/Map

3. 学生注册案例

sql:

```
create table students( id int primary key auto_increment, username varc
har(100) not null unique, password varchar(100) not null,    gender var
char(10),    hobby varchar(100),    birthday date, email varchar(100),
   grade int);
```

jsp:

```
<form action="${pageContext.request.contextPath}/student/regist" method
="post">
   <table border="1" width="438">
      <tr>
         <td>用户名: </td>
         <td>
            <input type="text" name="username"/>
         </td>
      </tr>
      <tr>
         <td>密码: </td>
         <td>
            <input type="password" name="password"/>
         </td>
      </tr>
      <tr>
         <td>性别: </td>
         <td>
            <input type="radio" name="gender" value="male" checked="che
cked"/>男性
            <input type="radio" name="gender" value="female"/>女性
```

```
            </td>
        </tr>
        <tr>
            <td>爱好: </td>
            <td>
                <input type="checkbox" name="hobbies" value="吃饭"/>吃饭
                <input type="checkbox" name="hobbies" value="睡觉"/>睡觉
                <input type="checkbox" name="hobbies" value="学java"/>学java

            </td>
        </tr>
        <tr>
            <td>出生日期: (yyyy-MM-dd)</td>
            <td>
                <input type="text" name="birthday"/>
            </td>
        </tr>
        <tr>
            <td>邮箱: </td>
            <td>
                <input type="text" name="email"/>
            </td>
        </tr>
        <tr>
            <td>成绩: </td>
            <td>
                <input type="text" name="grade"/>
            </td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="注册"/>
            </td>
        </tr>
    </table>
</form>
```

action:

```
public class StudentAction extends ActionSupport implements ModelDriven
<Student> {
    private Student student = new Student();//模型
    private BusinessService s = (BusinessService) new BusinessServiceImp
l();
    private String [] hobbies;//小特殊


    public String[] getHobbies() {
        return hobbies;
    }
```

```
    public void setHobbies(String[] hobbies) {
       this.hobbies = hobbies;
    }

    public String regist(){
       try {
          //单独处理爱好
                if(hobbies!=null&&hobbies.length>0){
                StringBuffer sb = new StringBuffer();
                for(int i=0;i<hobbies.length;i++){
                    if(i>0){
                            sb.append(",");
                        }
                    sb.append(hobbies[i]);
                }
             student.setHobby(sb.toString());
          }

          s.registStudent(student);
          return SUCCESS;
       } catch (Exception e) {
          e.printStackTrace();
          return ERROR;
       }
    }

    public Student getModel() {
       return student;
    }
```

4. 输入验证

4.1 编程式方式 : 验证规则写到了代码中, 硬编码

4.1.1 针对动作类中的所有动作方法进行验证

1. 动作类需要实现 ActionSupport 覆盖 validate 方法
2. 方法内部: 编写验证规则, 不正确添加信息 addFieldError

```
@Overridepublic void validate() {
    if (StringUtils.isBlank(student.getUsername())) {// 没有输入用户名
```

```
                addFieldError("student.username", "用户名不能为空!"); /
 / 相当于在 map 中存放信息     }
 }
```

3. 验证失败:
   视图: 会自动转到 name = input 的逻辑视图
   错误消息提示: 建议使用 Struts2标签库, 使用`<s:fieldError/>`
   其他 css 样式: `<s:head/>`

4.1.2 针对动作类中的指定动作方法进行验证

方式一: 使用一个注解 @skipValidation 这个是忽略

```
@SkipValidation
public String findAll(){
    return null;
}
```

方式二: 如果一个动作方法名叫做 regist, 只针对该方法进行验证, 需要编写 public void validateRegist() 方法

```
 public void validateRegist(){
 .... ....
 }
```

验证功能都是由 validation 拦截器所处理的
回显错误信息是由 workflow 拦截器处理的

4.2 声明式验证 : 把验证规则写到配置文件中, 更改非常便捷

4.2.1 针对动作类中的所有动作方法进行验证

动作类就是模型: 在Action包中创建 动作类名-validation.xml 配置文件

```
<validators>
    <!-- 要验证的字段 -->
    <field name="username">
        <!-- 指定验证规则 required用于验证字段是不是 null, requiredstring 用
于验证字段是不是空字符串 -->
```

```
        <field-validator type="requiredstring">
            <!-- 不符合规则提示错误信息 -->
            <message>您的名字不能为空!</message>
        </field-validator>
    </field>
</validators>
```

4.2.2 针对动作类中的指定动作方法进行验证

在动作类Action包中创建: 动作类名-动作名-validation.xml 配置文件.
动作名是 struts.xml 中的 action name

4.3 Struts2 中提供的内置声明式验证器的使用

Struts2 提供的声明式验证器在 xwork-core-**.jar包的 xwork-core-2.3.31/com/opensymphony/xwork2/validator/validators/default.xml 配置文件中

```
<validators>
    <!-- field指定要验证的字段, name是字段名。和表单一致 -->
<field name="stu.username">
        <!-- 必须由3~8位字母组成->
        <field-validator type="regex">
            <param name="regexExpression"><![CDATA[[a-zA-Z]{3,8}]]></pa
ram>
            <message>你的名字必须由3~8位字母组成</message>
        </field-validator>
    </field>

    <!-- 验证密码必须3~8位数字组成: 换一种 -->
<validator type="regex">
        <param name="fieldName">password</param>
        <param name="regexExpression"><![CDATA[\d{3,8}]]></param>
         <message>你的密码必须由3~8位数字组成</message>
    </validator>

    <!-- 必须选择性别 -->
<field name="stu.gender">
        <field-validator type="required">
            <message>请选择性别</message>
        </field-validator>
    </field>


    <field name="stu.email">
        <field-validator type="email">
            <message>请输入正确的邮箱</message>
        </field-validator>
```

```
    </field>
    <field name="stu.grade">
        <field-validator type="int">
            <param name="min">0</param>
            <param name="max">150</param>
            <message>成绩必须在${min}~${max}之间</message>
        </field-validator>
    </field>
</validators>
```

5. 国际化 [分享]

国际化的 API：struts-2.3.31/docs/docs/home.html / Guides / Localization　查看资源文件的访问

Resource Bundle Search Order

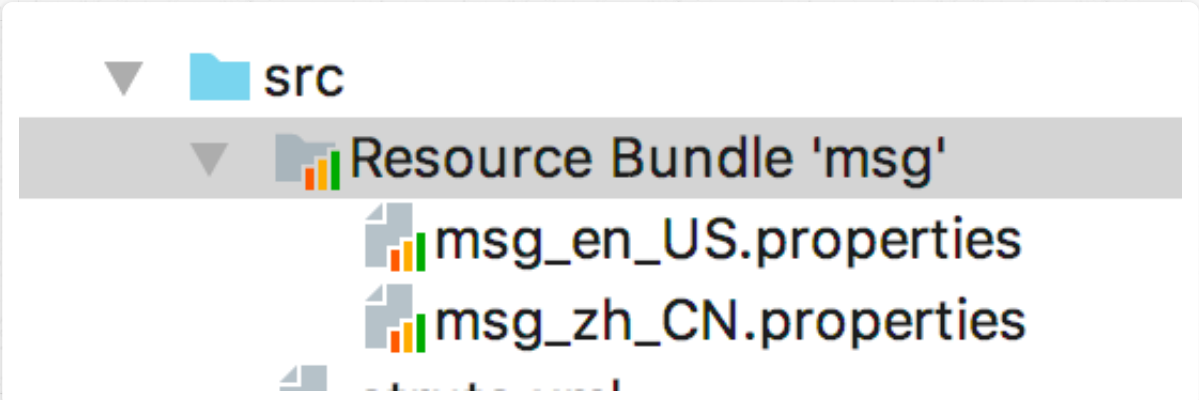
Resource bundles are searched in the following order:

1. ActionClass.properties
2. Interface.properties (every interface and sub-interface)
3. BaseClass.properties (all the way to Object.properties)
4. ModelDriven's model (if implements ModelDriven), for the model object repeat from 1
5. package.properties (of the directory where class is located and every parent directory all the way to the root directory)
6. search up the i18n message key hierarchy itself
7. global resource properties

For more, see the LocalizedTextUtil class.

5.1 配置全局消息资源文件

5.1.1 创建资源文件



5.1.2 在 struts.xml 中进行配置

```xml
<struts>
    <!-- 开发模式！ 写完配置文件可以及时刷新-->
    <constant name="struts.devMode" value="true"></constant>

    <!-- 配置全局消息资源文件-->
    <constant name="struts.custom.i18n.resources" value="msg"></constant>

</struts>
```

5.1.3 使用

- 在动作类中使用
  - 前提: 必须经过动作方法, 动作类需要继承 ActionSupport
- 在页面中使用

```xml
<body>
<s:text name="hello"></s:text>
```

5.2 配置局部消息资源文件

- 消息资源文件: **动作类名_zh_CN.properties**
- 访问:
  - 在 JSP 上使用, 直接访问的 JSP, 显示全局消息资源文件的内容, 因为没有经过动作类.
  - 如果通过访问动作, 转发到 JSP, 那么显示局部的
  - 如果通过访问动作类, 转发到 JSP, 但动作类没有继承 ActionSupport, 那么显示全局的

5.3 自由指定资源包

```
<%-- 自由选择消息资源包--%>
<s:i18n name="msg">
    <s:text name="hello"></s:text>
</s:i18n>
```

1. 拦截器

1.1 常用拦截器

ModelDiven : 模型驱动
servletConfig: 获取 ServletAPI
staicParams: 静态参数注入
params: 动态参数注入
validation: 输入验证, 声明式验证

1.2 自定义拦截器

1.编写一个类, 继承 AbstractInterceptor 抽象类, 实现intercept方法

```
public class Demo1Interceptor extends AbstractInterceptor {
   //返回值: 最终目标的返回值。就是一个结果逻辑视图。对应struts.xml中的result
public String intercept(ActionInvocation invocation) throws Exception {

      System.out.println("拦截前");
      String rtValue = invocation.invoke();//放行
      System.out.println("拦截后");
      return rtValue;
   }
```

2.定义与使用

```
<package name="p1" extends="struts-default">
      <!-- 声明自定义的拦截器-->
<interceptors>
         <interceptor name="testInterceptor" class="interceptors.Demo1I
nterceptor"/>
</interceptors>
     <action name="demo1" class="action.Demo1Action" method="m1">
         <!-- 一旦指定拦截器, 默认的那些拦截器将不起作用 ->
        <interceptor-ref name="testInterceptor"/>
        <result name="success">/success.jsp</result>
    </action>
</package>
```

思考: 如何能够让默认的拦截器和自己的拦截器都起作用?
解决: 定义拦截器栈,加入自定义拦截器和默认拦截器

```
<package name="p1" extends="struts-default">
    <!-- 声明拦截器 -->
    <interceptors>
        <interceptor name="demo1" class="com.lanou.struts.interceptors.
Demo1Interceptor"/>
        <!-- 定义拦截器栈-->
        <interceptor-stack name="myDefaultStack">
            <interceptor-ref name="demo1"/>
            <interceptor-ref name="defaultStack"/>
        </interceptor-stack>
    </interceptors>

    <action name="login1" class="com.lanou.struts.actions.Demo1Action"
method="login">
        <!-- 一旦指定了拦截器, 默认的拦截器们将不起作用 -->
        <!--<interceptor-ref name="demo1"/>-->
        <interceptor-ref name="myDefaultStack"/>
        <result name="success">/success.jsp</result>
    </action>
</package>
```

思考: 如何能够让其他的包也使用这个拦截器栈?
解决: 使用包继承

```
<package name="myDefault" extends="struts-default">
    <!-- 声明拦截器 -->
    <interceptors>
        <interceptor name="demo1" class="com.lanou.struts.interceptors.
Demo1Interceptor"/>
```

```
        <!-- 定义拦截器栈-->
        <interceptor-stack name="myDefaultStack">
            <interceptor-ref name="demo1"/>
            <interceptor-ref name="defaultStack"/>
        </interceptor-stack>
    </interceptors>
</package>

<package name="p1" extends="myDefault">
    <action name="login1" class="com.lanou.struts.actions.Demo1Action"
method="login">
        <!-- 一旦指定了拦截器, 默认的拦截器们将不起作用 -->
        <!--<interceptor-ref name="demo1"/>-->
        <interceptor-ref name="myDefaultStack"/>
        <result name="success">/success.jsp</result>
    </action>
</package>
```

思考: defaultStack 我们也没有每一个 action 都引用, 如何能让myDefault不需要 action 引用?

解决: 设置默认引用拦截器

```
<package name="myDefault" extends="struts-default">
    <!-- 声明拦截器 -->
    <interceptors>
        <interceptor name="demo1" class="com.lanou.struts.interceptors.
Demo1Interceptor"/>
        <!-- 定义拦截器栈-->
        <interceptor-stack name="myDefaultStack">
            <interceptor-ref name="demo1"/>
            <interceptor-ref name="defaultStack"/>
        </interceptor-stack>
    </interceptors>
    <default-interceptor-ref name="myDefaultStack"/>
</package>
```

1.3 拦截器案例 -- 登录拦截器

需求: 添加和更新方法需要登录才能够执行

```
public class LoginAction extends ActionSupport {
    // 添加和更新需要登录才能够执行
    public String add(){
        System.out.println("add....");
        return SUCCESS;
```

```
    }

    public String update(){
        System.out.println("update....");
        return null;
    }

    public String login(){
        ServletActionContext.getRequest().getSession().setAttribute("us
er", "DLW");
        return SUCCESS;
    }
}
```

登录拦截器:

```
public class LoginCheckInterceptor extends AbstractInterceptor {
   public String intercept(ActionInvocation invocation) throws Exceptio
n {
      //检查你有没有登录
      HttpSession session = ServletActionContext.getRequest().getSessio
n();
      Object obj = session.getAttribute("user");
         //没有登录：转向登录页面
     if(obj == null){
         return "login";
      }
         //登录：放行
      return invocation.invoke();
   }
```

struts.xml:

```
<package name="p2" extends="myDefault">
    <action name="add" class="action.Demo2Action" method="add">
        <result name="login">/login.jsp</result>
    </action>
    <action name="update" class="action.Demo2Action" method="update">
        <result name="login">/login.jsp</result>
    </action>
    <action name="login" class="action.Demo2Action" method="login">
        <result name="success">/index.jsp</result>
    </action>
</package>
```

访问报错, 意思是 login 方法没有 login 视图.
但问题是我们不需要验证 login 这个方法是否登录
解决
让方法拦截器继承 **MethodFilterInterceptor** 类, 修改 LoginCheck 的继承和方法
查看源码多了两个参数, 可以添加 param
`excludeMethods` : 抛除哪些方法不拦截
`includeMethods` : 包含哪些方法需要拦截

```
<interceptor name="loginCheck" class="interceptors.LoginCheckIntercepto
r">
    <param name="excludeMethods">login</param>
</interceptor>
```

---

2. 文件上传下载

2.1 文件上传

必要条件:
○ 表单 的 method 必须是 post
○ 表单 enctype 必须是 mulipart/form-data
○ 提供 input type=“file”类型上传输入域
注意: 如果是大文件火狐浏览器会拦截
在 struts2中文件上传是由 fileUpload 拦截器完成的

2.1.1 单文件上传

jsp:

```
<s:form action="uploadSingleFile" enctype="multipart/form-data" method=
"post">
   <s:textfield name="name" label="名称"></s:textfield>
   <s:file name="photo" label="大老王头像"></s:file>
   <s:submit value="上传"></s:submit>
</s:form>
```

```java
public class Upload1Action extends ActionSupport {
    private String name; // 对应着表单的名字
    private File photo; // 文件必须是 File 类型, 名字对应表单
    private String photoFileName; // 固定写法: XXFileName
    private String photoContentType; //固定写法: XXContentType  文件 MIME
 类型

    public String upload() throws IOException {
        System.out.println("名字: " +name+ ",文件名" +photoFileName+ ",
文件类型: " +photoContentType);
        ServletContext servletContext = ServletActionContext.getServlet
Context();
       String dirPath = servletContext.getRealPath("/files");
        // 构建目标文件
        File target = new File(dirPath, photoFileName);
        FileUtils.copyFile(photo, target);
        return SUCCESS;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public File getPhoto() {
        return photo;
    }

    public void setPhoto(File photo) {
        this.photo = photo;
    }

    public String getPhotoFileName() {
        return photoFileName;
    }

    public void setPhotoFileName(String photoFileName) {
        this.photoFileName = photoFileName;
    }

    public String getPhotoContentType() {
        return photoContentType;
    }

    public void setPhotoContentType(String photoContentType) {
        this.photoContentType = photoContentType;
    }
}
```

2.1.2 上传出现问题

- 上传失败时会转向 input 视图
- 错误显示: <s:actionerror/>

1. 修改上传文件大小
   `<constant name="struts.multipart.maxSize" value="10485760"/>`
2. 限制上传文件的类型 • FileUpload拦截器有3个属性可以设置 ○ maximumSize:上传文件的最大长度(以字节为单位),默认值为2 MB ○ allowedTypes : 允许上传的文件的MIME 类型, 多个使用逗号分隔 ○ allowedExtensions : 允许上传的文件的扩展名, 多个使用逗号分隔

```
<action name="upload1" class="com.lanou.updown.Upload1Action"
method="upload">       <interceptor-ref name="defaultStack">
     <!- 限制文件后缀 ->            <param
name="fileUpload.allowedExtensions">.jpg, .jpeg</param>
      </interceptor-ref>                    <result
name="success">/success.jsp</result>       <result
name="input">/upload1.jsp</result> </action>
```

3. 修改错误消息提示 文件上传的所有默认消息提示在:
   `srtuts2-core.jar/org/apache/struts2/struts-messages.properties` 修改显示错误资源文件 [1] 在 src 下创建 fileUploadMessage.properties ,名字任意.

```
struts.messages.upload.error.SizeLimitExceededException=\u6587\u4ef6\u5927
{0},\u60a8\u4e0a\u4f20\u7684\u6587\u4ef6\u003a{1}
struts.messages.error.file.extension.not.allowed=\u6b64\u6587\u4ef6\u6269\
{1} {3}
```

   [2] 在 xml 文件中声明
   `<constant name="struts.custom.i18n.resources" value="fileUploadMessage"/>`

2.1.3 多文件上传 [作业]

2.2 文件下载

结果集类型的使用 stream
• 动作类的书写规范 : inputStream / fileName
• struts.xml 中 action result 的 type="stream" 两个参数:
○ inputName[输入流的属性名]
○ contentDisposition=attachment;filename=${fileName}[通知浏览器以下载形式打开]

action:

```
public class DownLoadAction extends ActionSupport{

    // 定义一个 输入流, 名字不能是 in
    private InputStream inputStream;

    public String downLoad() throws IOException {
        //  实现下载: 给 inputStream 赋值
        String realPath = ServletActionContext.getServletContext().getR
ealPath("/1.jpg");
        inputStream = new FileInputStream(realPath);

        return SUCCESS;
    }

    public InputStream getInputStream() {
        return inputStream;
    }

    public void setInputStream(InputStream inputStream) {
        this.inputStream = inputStream;
    }
}
```

struts.xml:

```
<package name="down" extends="struts-default">
    <action name="download" class="com.lanou.updown.DownAction" method=
"download">
        <result name="success" type="stream">
            <!-- 指定动作类中的输入流的属性名 -->
            <param name="inputName">inputStream</param>
            <!-- 通知浏览器以下载的方式打开  获取文件名是OGNL表达式，就是调用动作
类中的getFileName方法 -->
            <param name="contentDisposition">attachment;filename=1.jpg<
/param>
            <!-- MIME类型 -->
            <param name="contentType">application/octet-stream</param>
        </result>
    </action>
</package>
```

问题: 文件的名字不应该是固定的, 应该是动态的, 如何更改?

解决:

文件名动态处理 添加 fileName 属性, 将

`<param name="contentDisposition">attachment;filename=${fileName}</param>`

中文处理

```
fileName =
filenameEncoding(FilenameUtils.getName(realPath),ServletActionContext.getRequ
ServletActionContext.getResponse());
```

```
public String filenameEncoding(String filename, HttpServletRequest requ
est, HttpServletResponse response) throws IOException {
    String agent = request.getHeader("User-Agent"); //获取浏览器的类型
System.out.println(agent);
    if (agent.contains("Firefox")) {
        BASE64Encoder base64Encoder = new BASE64Encoder();
        filename = "=?utf-8?B?"                + base64Encoder.encode(f
ilename.getBytes("utf-8"))
                + "?=";
    } else if(agent.contains("MSIE")) {
        filename = URLEncoder.encode(filename, "utf-8");
    } else if (agent.contains("Safari")){
        filename = new String(filename.getBytes("UTF-8"),"ISO8859-1");
    }else {
        filename = URLEncoder.encode(filename, "utf-8");
    }
    return filename;
}
```

3. OGNL 表达式简介

OGNL是Object Graphic Navigation Language（对象图导航语言）的缩写，它是一个开源项目。webwork 用它作为表达式语言, Struts2框架使用OGNL作为默认的表达式语言。

3.1 OGNL 的优势

EL 表达式只能在 jsp 中使用,只能够取数据, 但是 OGNL 还有其他优势:

3.1.1 调用任意对象任意方法

```
调用任意对象任意方法<br>
<%--<s:property/> 就相当于 jstl 中的 out使用单引号括起来的是字符串, 否则是表达式
--%>
<s:property value="'YSL'.length()"/>
```

3.1.2 支持类静态的方法调用和值访问

```
访问静态变量<br>
<s:property value="@java.lang.Integer@MAX_VALUE"/>
访问静态方法<br>
<s:property value="@@abs(-100)"/>
```

报错

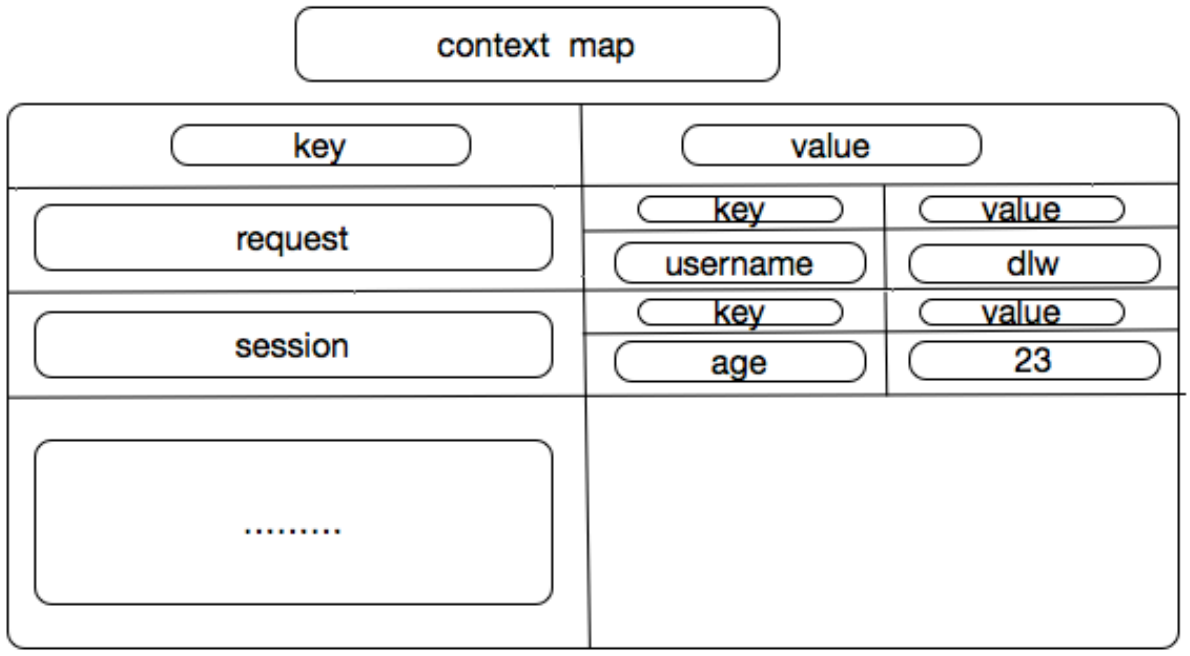
- 默认不允许使用静态方法, 设置 struts.ognl.allowStaticMethodAccess=true
- Struts2 版本问题

4. context 上下文: 数据中心

struts2/docs/docs/home.html — Guides — OGNL

```
                        |
                        |--application
                        |
                        |--session
          context map---|
                        |--value stack(root)
                        |
                        |--action (the current action)
                        |
                        |--request
                        |
                        |--parameters
                        |
                        |--attr (searches page, request, session, then application scopes)
                        |
```

key	Value	备注
application	ServletContext 中所有的属性 attributes	相当于 EL 中的内置对象 applicationScope . 就是保存着属性的 map
session	HttpSession 中的所有属性	相当于 EL 中的内置对象 sessionScope
value stack(根)	它是一个 List	
action	动作类	
request	ServletRequest 中的所有属性	相当于 EL 中的内置对象 requestScope
parameters	Map map = request.getParamterMap()	相当于 EL 中的内置对象 paramValues
attr	从四大域范围中搜索	相当于 ${}



4.1 ActionContext 的 API

操作 context Map 中的数据

```
public class ActionContext extends ActionSupport{
    public String m1(){
```

```
        // 得到 actionContext 实例: 绑定到当前线程
        ActionContext ac =  ActionContext.getContext();

       //  put 向 context map 中存放数据. 和request,session 并列
        ac.put("uName","DLW”);

        // 从 context map 取数据
        String name = (String) ac.get("uName");
        System.out.println(name);

        // 获取 context map 中的根 三种方式
         // 一:
        ValueStack vs1 = ac.getValueStack();

        // 二:
        Map<String, Object> reqAttrs = (Map<String, Object>) ac.get("re
quest");
        ValueStack vs2 = (ValueStack) reqAttrs.get("struts.valueStack")
;

        // 三:
        ValueStack vs3 = (ValueStack) ac.get("com.opensymphony.xwork2.u
til.ValueStack.ValueStack");


        // 向 session 域中存放属性
           HttpSession session = ServletActionContext.getRequest().getS
ession();
          session.setAttribute("p2", "哈哈哈哈");
      // 使用 actionContext 取 session 中数据
           ac.getSession().get("p2");


           // 获取应用范围的数据
        ServletContext servletContext = ServletActionContext.getServlet
Context();
        servletContext.setAttribute("p4", "嘻嘻嘻嘻");
        String s4 = (String) ac.getApplication().get("p4");
        System.out.println(s4);

        return null;
    }
```

4.2 ValueStack 的 API

重点操作 context Map 中根的数据

```
public class ValueStackAction1 extends ActionSupport{
    public String m1(){
        ValueStack vs = ActionContext.getContext().getValueStack();
        Map<String, Object> contextMap = vs.getContext();
        // 与  ActionContext 中的 context 是一个
        // 验证
        ActionContext.getContext().put("YYY", "zzz");
        System.out.println(contextMap.get("YYY"));

        // 先进后出
        CompoundRoot root = vs.getRoot();
        System.out.println(root);

        // 压一个对象进栈
        vs.push(new Date());

        // 取栈顶对象
        Object o = vs.peek();
        System.out.println(o);

        // 弹栈
            // vs.pop();
        vs.pop();

        return SUCCESS;
    }
```

```
        public class ValueStackAction2 extends ActionSupport {
    private String name = "大老王";

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
    public String m1(){
        ValueStack vs = ActionContext.getContext().getValueStack();
        // 检测栈顶对象是不是 map, 如果不是, 创建 map,map 中的数据 ZMY=pppp.
并且这个 map 压入栈顶
        vs.set("ZMY", "pppp");
        vs.set("DLW", "2222"); // 检测到栈顶就是一个 map, 直接把数据放在 map
中

        //----------------------
        vs.setValue("DLW","233333");
```

```
        // 第一个参数是 OGNL 表达式，从栈顶开始搜索，调用 setDLW("233333");

        vs.setValue("name","大傻子");
        // 设置动作类中的 name 改成大傻子 setName();

        vs.setValue("#abc", "66666");
        // 向 context map 中存放数据，相当于 ActionContext.getContext().pu
t("abc", "66666");

        // ----------------------
        // 取数据
        Object obj1 = vs.findValue("name");
        System.out.println(name + obj1);

        Object obj2 = vs.findValue("#abc");
        System.out.println(obj2);

        // 不建议使用
        // 先从栈顶开始找 getAbc，找不到把 abc 当做 key 到 context map 中找
        Object obj3 = vs.findValue("abc");
        System.out.println(obj3);

        return SUCCESS;
    }
```

5. OGNL 的用法

> `#` *号的用法*
> • *在 JSP 中使用 OGNL 显示 contextMap. 访问OGNL上下文和Action上下文，#相当ActionContext.getContext()*
> • *构造Map，如#{'foo1':'bar1','foo2':'bar2'}*
> • *集合的投影(只输出部分属性) （过滤）*
> `%` *号的用法*
> • *"%"符号的用途是在标签的属性值被理解为字符串类型时，告诉执行环境%{}里的是OGNL表达式。*
> `$` *号的用法*
> • *在Struts 2配置文件中，引用OGNL表达式*
> • *用于在国际化资源文件中，引用OGNL表达式*

5.1 在 JSP 中使用 OGNL 显示 contextMap或者是根中的数据

5.1.1 获取 contextMap中的数据, OGNL 表达式要用 # 开头, #相当于 ActionContext.getContext()

```
<s:property value="#p5"/>
```

获取 request 中数据 #request. p5
也可以写成 #request['p5']

5.1.2 获取 根(List)中的对象的属性, 直接写属性的名称.[会从栈顶的对象一直往下找get方法, 找到了就不需要找]

```
获取 根(List)中的对象的属性, 直接写属性的名称.
<s:property value="month"/><br>
<%-- 获取第二个对象的 month --%>
<s:property value="[2].month"/><br>
```

5.2 在 jsp 中取数据, 可以使用 OGNL 表达式,也可以使用 EL 表达式

```
<s:property value="#request.p1"/>
${p1}<br>

<%-- 写了 EL${}全域查找
    1. 先找 pageContextScope
    2. 找 request , 如果找不到
    3. 从栈顶开始找对象的属性(getXXX()), 找不到
    4. 将 XXX 作为 key 到 context map 中找, 找不到
    5. session ....       --%>
${name}<br>
${p2}<br>
```

5.3 JSP 页面 OGNL 构造 List , Map

```
<!-- 构造一个List对象 -->
    <s:property value="{'a','b','c'}"/><br/>
<!-- 构造一个Map对象 -->
    <s:property value="#{'1':'男性','0':'女性'}”/>
```

5.4 OGNL 表达式当做字符串输出' ', 字符串当做 OGNL 表达式 %{}

```
<!-- 把OGNL表达式当做普通字符串对待 -->
      <s:property value="'sex'"/><br/>

<!-- 把字符串当做OGNL表达式 使用百分号引起来-->
      <s:textfield name="username" label="%{姓名}"></s:textfield>
```

5.5 在配置文件中使用 ognl 表达式 ${ognl}

6. Struts2中的常用标签, 主题

6.1 基础标签

有100多个. struts.jar/META-INF/struts-tags.tld

Guides > Tag Developers Guide > Struts Tags > Generic Tags

6.1.1 property

```
<%-- property: 输出数据到页面-%>
<%-- 不能直接访问jsp, 需要访问动作类, 动作类才会在栈顶, 才能搜得到 name 属性 -%>
<s:property value="name"/><br>
<%-- value 不写, 取栈顶对象-%>
<s:property/><br>
<s:property value="'<hr/>'" escapeHtml="false"/><br>
```

6.1.2 set

```
<%-- set 用于将某个值放置  查看文档-%>
<%-- value有可能认为是一个OGNL 表达式,由于没有取值,所以找不到就没有保存,需要转成字符串才能保存-%>
<s:set value="'value1'" var="v1" scope="session"></s:set><br>
<%--  放数据:    scope: application/request/session/page/action
     默认范围: action : 1.请求范围,2.放到 contextMap 中    但是 debug 标签没有显示 request 中有-%>
<s:set value="'value2'" var="v2"/><br>
contextMap中取:
<s:property value="#v2"/><br>
<%-- 在这取就能看出来 -%>
request: <s:property value="#request.v2"/><br>
```

6.1.3 action

```
<%-- action : 通过指定命名空间和action名称,该标签允许在jsp页面直接调用Action
    name:action名字(不包括后缀,如.action)
    namespace:action所在命名空间
    executeResult:Action的result是否需要被执行,默认值是false不执行   struts2中的包含-%>
<s:action name="demo2" executeResult="true"></s:action><br>
```

6.1.4 Iterator

```
<%--!!! Iterator: foreach . 标签用于对集合进行迭代，这里的集合包含List、Set和数组。
--%><table border="1”>
    <tr>
        <th>key</th>
        <th>value</th>
    </tr>
<%-- 指定了 var: 把当前遍历的元素存到了contextMap 中, key 就是 var 指定的值 map=Map.Entry--%>
<s:iterator value="#request" var="map”>
        <tr>
        <%--<s:property value="#map.key"/>
       <s:property value="#map.value”/> --%>
            <td>
               <s:property value="%{#map.getKey()}”/>
           </td>
           <td>
```

```
                <s:property value="#map.value”/>
            </td>
        </tr>
     </s:iterator>
</table>
<hr><hr>
<table border="1”>
    <tr>
        <th>key</th>
        <th>value</th>
        <th>序号</th>
    </tr>
<%-- 不指定了 var: 把当前遍历的元素存到了根的栈顶
     status: 指向一个对象, 记录当前遍历元素的信息,放到 contextMap 中

     该对象有以下方法:
     int getCount(), 返回当前迭代了几个元素。
     int getIndex(), 返回当前迭代元素的索引。
     boolean isEven(), 返回当前被迭代元素的索引是否是偶数
     boolean isOdd(), 返回当前被迭代元素的索引是否是奇数
     boolean isFirst(), 返回当前被迭代元素是否是第一个元素。

     boolean isLast(), 返回当前被迭代元素是否是最后一个元素。          --%>
<s:iterator value="#request" status="s”>
<%--<tr class="<s:property value='#s.odd?"odd":"even"'/>">--%>
<tr class="${s.odd?'odd':'even'}”>
            <td>
                <s:property value="key”/>
            </td>
            <td>
                <s:property value="value”/>
            </td>
            <td>
                <s:property value="#s.getCount()”></s:property>
            </td>
        </tr>
    </s:iterator>
</table>
<hr>
```

6.1.5 if elseif else

```
<%-- if elseif else  类似: jstl  when otherwise--%>
<s:set value="'B'" var="grade”/>
<s:if test="#grade == 'A'”>
    优秀
</s:if>
<s:elseif test="#grade=='B'”>
    尚需努力
```

```
</s:elseif>
```

6.1.6 url

```
<%-- url, 对应和 jstl 中的 url 一样. 保存在 context map 中-%>
<s:url action="demo2" var="u1">
    <s:param name="username" value="'你好'"></s:param>
</s:url>
<a href="${u1}">猛点</a>


<s:a action="demo2">
    再戳啊~
    <s:param name="username" value="'你也好'"></s:param>
</s:a>
```

6.2 UI 标签

Home > Guides > Tag Developers Guide > Struts Tags > Tag Reference > UI Tag Reference

6.2.1 表单标签 form

```
<%-- form 标签.
      优点:  表单回显 | 对页面进行布局和排版
      默认情况下, form 标签将被呈现为一个表格形式的 HTML 表单
    action: 设置提交的 action 名字, 不需要添加 .action 后缀
    method: 默认是 post-%>
<s:form action="demo2" method="" namespace="">
</s:form>
```

6.2.2 textfield, password, hidden, submit

```
<%-- textfield 标签将被呈现为一个输入文本字段, [input type="text"]
     password 标签将被呈现为一个口令字段,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
     hidden 标签将被呈现为一个不可见字段.
     submit 提交-%>
<s:textfield name="username" label="用户名" requiredLabel="true" require
dPosition="left"></s:textfield>
<s:password name="password" label="密码" showPassword="true"></s:passwor
d>
```

```
<s:submit value="提交"></s:submit>
```

6.2.3 radio

```
<%-- 选中女  var 和 name 必须一致-%>
<s:set value="'女'" var="sex"></s:set>
<s:radio name="sex" list="{'男','女'}" value="'女'"></s:radio><hr>
<%--选中可以在上面写 set, 会寻找 map 里如果有同样的 key, 那个 value 就被选中. 一般使用标签中value设定-%>
<s:radio name="sex" list="#{'man':'男','woman':'女'}" value="'man'"></s:radio>
```

6.2.4 checkbox, checkboxList

```
<%-
    checkbox :  http://www.blogjava.net/SpartaYew/archive/2011/05/19/checkbox.html
    value    :  指是否选中，其值只能为True或False，相当于传统checkbox中的checked。
    fieldValue :  相当于传统checkbox中的value值。
    label    :  对于该checkbox显示在页面上方框后面的描述。[选中这里可以使用 checked 的, 但是 API 中是没有的. 不建议使用]-%>
<s:checkbox name="hobby" fieldValue="eat" label="吃饭" value="true"></s:checkbox>
<s:checkbox name="hobby" fieldValue="sleep" label=" 喝水" value="true"></s:checkbox>
<s:checkbox name="hobby" fieldValue="beat" label="打豆豆" value="true"></s:checkbox><hr>

<%-- checkboxList-%>
<%-- 默认选择-%>
<s:set value="'eat'" var="hobby1"/>
<s:checkboxlist name="hobby1" list="#{'sleep':'睡觉', 'eat':'吃饭', 'beat':'打豆豆'}"/><hr>

<s:checkboxlist name="hobby" list="%{#{'eat':'吃饭', 'sleep':'睡觉'}}" label="爱好"></s:checkboxlist><br>

<%-- 如果想让存数据库时候 key,value 反过来取, 就添加 listKey, listValue-%>
<s:checkboxlist name="hobby" list="%{#{'eat':'吃饭', 'sleep':'睡觉'}}" label="爱好" listKey="value" listValue="key"></s:checkboxlist><br>
<%-- 单个选中状态-%>
<s:checkboxlist name="hobby" list="%{#{'eat':'吃饭', 'sleep':'睡觉'}}" label="爱好" value="'eat'"></s:checkboxlist><br>
<%-- 多个选中状态-%>
```

```
<s:checkboxlist name="hobby" list="%{#{'eat':'吃饭', 'sleep':'睡觉'}}" la
bel="爱好" value="{'eat','sleep'}"></s:checkboxlist><br>

<%-- 集合中放置的 javabean --%>
<%        Person person1 = new Person(1,"第一个");
          Person person2 = new Person(2,"第二个");
          Person person3 = new Person(3,"第三个");
          List<Person> list = new ArrayList<Person>();
          list.add(person1);
          list.add(person2);
         list.add(person3);
         request.setAttribute("persons",list);
%>
    <s:checkboxlist name="beans" list="%{#request.persons}" listKey="id
" listValue="name" label="你喜欢的童鞋"></s:checkboxlist>
```

6.2.5 select

```
<s:select name="city" list="#{'BJ':'北京', 'SD':'山东'}" label="城市”>
</s:select>
<s:select name="city" list="#request.persons" listKey="id" listValue="n
ame" label="人物"></s:select>
```

6.3 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

• 提供的主题:
○ simple (实际用): 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. （默认值）
○ 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 Struts2 的主题
○ A. 通过 UI 标签的 theme属性(只适用于当前的标签)
○ B. 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(适用于整个form标签)
○ C. 修改 struts.properties 文件中的 struts.ui.theme 属性.(适用整个环境)
• 优先级: A>B>C

```
<s:textfield name="username" label="用户名" theme="simple"></s:textfield
>
<!-- 修改表单 -->
<s:form action="demo2" method="" namespace="" theme="simple"> </s:form>

<!-- 修改全局-->
<constant name="struts.ui.theme" value="simple"/>
```

提供的主题在 jar 包中, 直接在工程中翻看