# 1. Struts操作域对象

能够得到域对象进行操作很重要，因为需要操作域对象中的数据，然后页面可以显示数据。在Struts的Action类中，一般使用以下三种方式来获得域对象并操作。

## 1.1 使用ServletActionContext类

此种方式是直接使用Servlet的API进行域对象的操作。因为ServletActionContext类提供了很多静态方法，可直接拿到与Servlet相关的类。常用的有：

（1）ServletActionContext.getRequest()：获得request对象；

（2）ServletActionContext.getResponse()：获得reponse对象；

（3）ServletActionContext.getServletContext()：获得ServletContext对象，可操作application域对象；

（4）ServletActionContext.getPageContext()：获得pageContext对象。

虽然没有提供session对象，但是我们可以通过request对象得到。

例如：

|  |
| --- |
| **public** String add() {  HttpServletRequest request = ServletActionContext.*getRequest*();  HttpServletResponse response = ServletActionContext.*getResponse*();  ServletContext context = ServletActionContext.*getServletContext*();  PageContext pageContext = ServletActionContext.*getPageContext*();  *// 操作域对象* request.setAttribute(**"name"**, **"张三"**);  *// 获得session* HttpSession session = request.getSession();  **return "success"**; } |

这种使用ServletAPI的方式，使软件的耦合性变高了。但是可用Servlet中通用的方法，比较简单，推荐使用，只要是在Struts框架中，基本都能使用该方法拿到Servlet对象（例如在拦截器中拿到Servlet对象）。

## 1.2 使用ActionContext类

Struts将很多Servlet中的对象进行了封装，这样就避免了和Servlet的耦合性。比如ActionContext类中，就将request、session和context这些域对象封装成了Map集合对象，即调用集合的put方法向这些“域对象”中存数据。

（1）通过ActionContext.getContext()方法获得一个ActionContext实例。

（2）调用实例的以下方法得到不同的域对象所对应的Map集合：

getContextMap()：得到request域对象对应的Map集合；

getSession()：得到session域对象对应的Map集合；

getApplication()：得到application域对象对应的Map集合。

例如：

|  |
| --- |
| **public** String add() {  *// 得到ActionContext* ActionContext ac = ActionContext.*getContext*();  *// 得到request对应的Map* Map<String, Object> request = ac.getContextMap();  *// 存数据* request.put(**"date"**, **new** Date());  *// 得到session和application* Map<String, Object> session = ac.getSession();  Map<String, Object> application = ac.getApplication();  **return "success"**; } |

## 1.3 实现XxxAware接口

Action类可分别实现RequestAware、SessionAware和ApplicationAware接口。之后实现接口的setXxx(Map<String, Object> map)方法。该方法中的参数就是获取到的对应的request、session和application的Map集合。可以在类中定义request等Map变量，通过setXxx()方法设置好值后，就可直接在类的各个地方使用了。

例如：

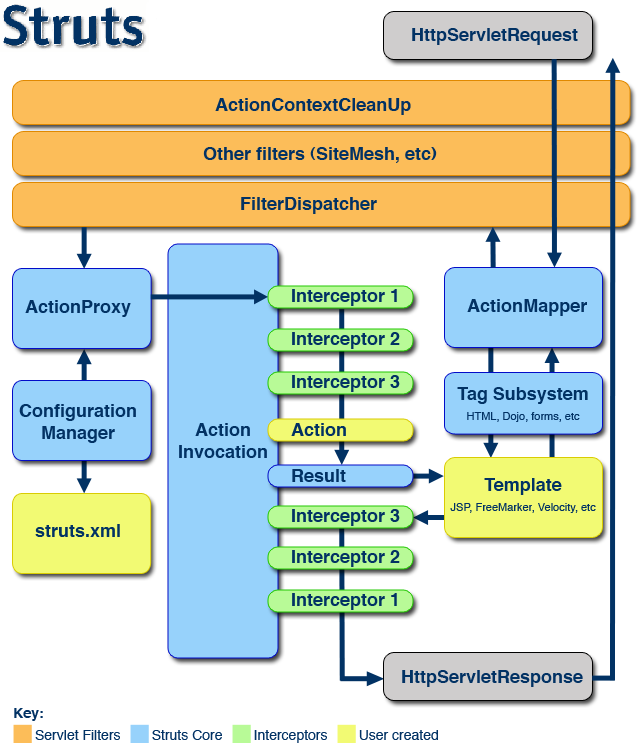
|  |
| --- |
| **public class** DemoAction **implements** RequestAware, SessionAware, ApplicationAware {  *// 下面的字段用于接收注入的变量* Map<String, Object> **request**;  Map<String, Object> **session**;  Map<String, Object> **application**;  @Override  **public void** setRequest(Map<String, Object> request) {  *// 进行注入* **this**.**request** = request;  }  @Override  **public void** setSession(Map<String, Object> session) {  **this**.**session** = session;  }  @Override  **public void** setApplication(Map<String, Object> application) {  **this**.**application** = application;  }  *// 下面在各个方法中即可方便的使用了。* **public** String add() {  **request**.put(**"name"**, **"张三"**);  **return "success"**;  } } |

该方式提供setRequest等方法是为了让Struts框架能成功地把request等对象注入进来，这个功能也是由一个叫做“servletConfig”拦截器来做的。

总结：上述三种方式都可使用。下面两种方式对Servlet进行了解耦，但有些场景还是需要用方式1才能解决问题。

# 2. Struts2的拦截器

Struts官方文档有一幅介绍Struts架构的图，如下：



重点看其中ActionInvocation中的内容，其中在Action（即我们编写的动作类）执行前会执行一些拦截器，在Action执行后又经过这些拦截器返回。拦截器的作用就相当于过滤器Filter，它就会完成特定的功能（例如设置请求参数等）。图中标示黄色部分是需要用户创建的，也就是在开发中，我们只要编写配置文件、Action和视图（如JSP）即可，其他是框架提供的功能。

## 2.1 自定义拦截器

Struts的拦截器和Servlet中的过滤器功能类似，都用于拦截资源，可增加很多功能。

如果要自定义拦截器，只要编写一个类实现Interceptor接口即可，Interceptor接口定义如下：

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

和Filter很类似，init是初始化方法，destroy是销毁时执行方法，intercept中用于处理和放行等。一般，如果拦截器类不需要执行初始化和销毁方法的话，那么拦截器类只要继承AbstractInterceptor这个抽象类即可，因为该类空实现了init和destroy方法：

|  |
| --- |
| **package** com.opensymphony.xwork2.interceptor;  **import** com.opensymphony.xwork2.ActionInvocation;  **public abstract class** AbstractInterceptor **implements** Interceptor {  **public** AbstractInterceptor() {  }  **public void** init() {  }  **public void** destroy() {  }  **public abstract** String intercept(ActionInvocation invocation) **throws** Exception; } |

因此，我们这里自定义拦截器继承AbstractInterceptor类，实现其中intercept方法。

|  |
| --- |
| **public class** MyInterceptor **extends** AbstractInterceptor {  @Override  **public** String intercept(ActionInvocation invocation) **throws** Exception {  System.***out***.println(**"拦截前执行代码..."**);  String rtValue = invocation.invoke(); *// 这个是“放行”，invoke的返回值最终是要返回的逻辑视图名，这也是放行后本方法要返回的返回值。* System.***out***.println(**"拦截后执行代码..."**);  **return** rtValue;  } } |

上述拦截器代码和Filter如出一辙，在放行前后就是拦截前和拦截后的代码。需要注意的是intercept的返回值就是要返回的逻辑视图名，如果所有的拦截器都放行了，最终返回的就是动作类的逻辑视图名，如果某个拦截器未放行，则返回指定的逻辑视图名。

如果想让某action应用此拦截器，则应在action中这样配置：

|  |
| --- |
| <**package name="default" namespace="/" extends="struts-default"**>  *<!-- 先定义拦截器 -->* <**interceptors**>  <**interceptor name="myInterceptor" class="com.ssh.interceptor.MyInterceptor"** />  </**interceptors**>  <**action name="deleteStudent" class="studentAction" method="delete"**>  *<!-- 应用拦截器 -->* <**interceptor-ref name="myInterceptor"** />  <**result name="success"**>/index.jsp</**result**>  </**action**> </**package**> |

这样拦截器就成功应用了。但是需要注意的是，一旦为action指定了应用的拦截器，那么其他的拦截器将不再起作用，即此时deleteStudent这个action将不会被Struts默认拦截器拦截，这样的话封装参数等功能都失效了。要解决这个问题可以定义一个拦截器栈，例如：

|  |
| --- |
| <**package name="default" namespace="/" extends="struts-default"**>  <**interceptors**>  <**interceptor name="myInterceptor" class="com.ssh.interceptor.MyInterceptor"** />  *<!-- 定义拦截器栈，其中包含了默认拦截器栈和myInterceptor -->* <**interceptor-stack name="myStack"**>  <**interceptor-ref name="myInterceptor"** />  <**interceptor-ref name="defaultStack"** />  </**interceptor-stack**>  </**interceptors**>    <**action name="deleteStudent" class="studentAction" method="delete"**>  *<!-- 这时应用myStack拦截器栈即可 -->* <**interceptor-ref name="myStack"** />  <**result name="success"**>/index.jsp</**result**>  </**action**> </**package**> |

甚至可以把默认拦截器栈覆盖了：

|  |
| --- |
| <**package name="default" namespace="/" extends="struts-default"**>  <**interceptors**>  <**interceptor name="myInterceptor" class="com.ssh.interceptor.MyInterceptor"** />  *<!-- 定义拦截器栈，其中包含了默认拦截器栈和myInterceptor -->* <**interceptor-stack name="myStack"**>  <**interceptor-ref name="myInterceptor"** />  <**interceptor-ref name="defaultStack"** />  </**interceptor-stack**>  </**interceptors**>  *<!-- 覆盖默认的拦截器栈 -->* <**default-interceptor-ref name="myStack"** />  <**action name="deleteStudent" class="studentAction" method="delete"**>  *<!-- 这时这里可不用配置拦截器 -->* <**result name="success"**>/index.jsp</**result**>  </**action**> </**package**> |

这样action就使用新的“默认拦截器栈”了。

## 2.2 使用拦截器进行细粒度权限控制

例1：编写拦截器以实现对访问Action时判断是否登录，如果登录了就放行，否则跳转到登录页面。

拦截器代码如下：

|  |
| --- |
| **package** com.ssh.interceptor;  **import** com.opensymphony.xwork2.ActionInvocation; **import** com.opensymphony.xwork2.interceptor.AbstractInterceptor; **import** org.apache.struts2.ServletActionContext;  **import** javax.servlet.http.HttpSession;  **public class** LoginCheckInterceptor **extends** AbstractInterceptor {  @Override  **public** String intercept(ActionInvocation invocation) **throws** Exception {  *// 检查用户是否登录* HttpSession session = ServletActionContext.*getRequest*().getSession();  *// 未登录则转向登录页面* Object obj = session.getAttribute(**"user"**);  **if** (obj == **null**) {  **return "login"**; *// 返回“login”对应的逻辑视图* }  **return** invocation.invoke();  } } |

struts.xml中配置如下：

|  |
| --- |
| <**struts**>  <**package name="default" namespace="/" extends="struts-default"**>  <**interceptors**>  <**interceptor name="loginInterceptor" class="com.ssh.interceptor.LoginCheckInterceptor"** />  *<!-- 定义拦截器栈包含loginInterceptor -->* <**interceptor-stack name="myStack"**>  <**interceptor-ref name="loginInterceptor"** />  <**interceptor-ref name="defaultStack"** />  </**interceptor-stack**>  </**interceptors**>   <**action name="deleteStudent" class="studentAction" method="delete"**>  <**interceptor-ref name="myStack"** />  <**result name="success"**>/index.jsp</**result**>  *<!-- 这里可定义登录拦截器中可能返回的login视图 -->* <**result name="login"**>/login.html</**result**>  </**action**>  </**package**> </**struts**> |

这时访问Action时如果未登录就会跳转到login.html页面。利用拦截器能拦截Action中的所有方法，能实现细粒度的权限控制，也就是在每个方法执行前通过拦截器判断用户是否具有此权限，如果没有就禁止操作。

但是上述拦截器是针对所有动作方法进行拦截，为了具有较高的灵活度，我们使用功能更加强大的MethodFilterInterceptor。代码如下：

|  |
| --- |
| **package** com.ssh.interceptor;  **import** com.opensymphony.xwork2.ActionInvocation; **import** com.opensymphony.xwork2.interceptor.MethodFilterInterceptor; **import** org.apache.struts2.ServletActionContext;  **import** javax.servlet.http.HttpSession;  **public class** LoginCheckInterceptor **extends** MethodFilterInterceptor {  @Override  **public** String doIntercept(ActionInvocation invocation) **throws** Exception {  *// 里面的处理代码和之前都是一模一样的* HttpSession session = ServletActionContext.*getRequest*().getSession();  Object obj = session.getAttribute(**"user"**);  **if** (obj == **null**) {  **return "login"**;  }  **return** invocation.invoke();  } } |

上述的代码除了换成了继承MethodFilterInterceptor类和实现了其中的doIntercept方法，其他都是一样的。MethodFilterInterceptor又继承了AbstractInterceptor，为什么使用MethodFilterInterceptor？这是因为MethodFilterInterceptor中提供了两个属性：“excludeMethods”和“includeMethods”，利用这两个属性能设置哪些动作方法需要（或者不需要）拦截。注意的是，MethodFilterInterceptor类中的抽象类不再是intercept方法了，而是改成了doIntercept方法。

如果我们不需要login等方法被该拦截器拦截，只要这样配置即可：

|  |
| --- |
| <**struts**>  <**package name="default" namespace="/" extends="struts-default"**>  <**interceptors**>  <**interceptor name="loginInterceptor" class="com.ssh.interceptor.LoginCheckInterceptor"** />  *<!-- 定义拦截器栈包含loginInterceptor -->* <**interceptor-stack name="myStack"**>  <**interceptor-ref name="loginInterceptor"**>  *<!-- 写excludeMethods和includeMethods内容时，写的是方法名，而不是action名称 -->* <**param name="excludeMethods"**>login</**param**>  </**interceptor-ref**>  <**interceptor-ref name="defaultStack"** />  </**interceptor-stack**>  </**interceptors**>   <**action name="deleteStudent" class="studentAction" method="delete"**>  <**interceptor-ref name="myStack"** />  <**result name="success"**>/index.jsp</**result**><**result name="login"**>/login.html</**result**>  </**action**>  </**package**> </**struts**> |

当然，param的配置也可在action标签中的interceptor-ref标签中设置，但是为了通用一般就在栈中就设置了。这样就不会验证login方法的用户登录了。到底使用excludeMethods还是includeMethods，取决用包含方法多还是排除方法多。学了Spring，就能知道为什么能在XML中配置属性名进而影响对象的对应属性，其实就是根据setter方法设置的。

有时，为了拦截所有的资源（包括Action、HTML等WEB资源等），必须使用Servlet技术中的Filter过滤器，并且应该在配置Struts过滤器之前配置自定义的过滤器，这样自定义过滤器才会真正有效。

# 3. 请求数据自定义类型转换

提交请求数据时，Struts会自动将参数值封装到属性中，但是这仅仅能够封装字符串、数组、集合和基本数据类型的值。对于Date类型，则只能封装“yyyy-MM-dd”格式的时间字符串，对于其他对象，其对象的属性类型也需要是字符串或者是基本数据类型。

那么如何将请求的字符串数据转换成其他的数据类型呢？比如我们想将“yyyy年MM月dd日 hh:mm:ss”格式的时间字符串参数值，直接封装成Date类型对象。

这时，我们需要自定义类型转换器。步骤如下：

（1）编写转换器类，该类要继承StrutsTypeConverter。例如需要上传一个人的生日，其时间格式可能是“xxxx年xx月xx日”，或是“19950115”，也可能是“1995-05-04”。要求该转换器都能使Struts进行成功转换。

代码示例（在某个包（比如com.util）下新建MyDateConverter）：

|  |
| --- |
| **package** com.util;  */\*\*  \* Created by zhang on 2017/3/9.  \*/* **public class** MyDateConverter **extends** StrutsTypeConverter {  *// 其中要重写两个方法。一个处理从String转成想要的类型，一个是将该类型转换成String。主要是写第一个方法。  // 先定义支持的时间格式* **private** DateFormat[] **df** = {  **new** SimpleDateFormat(**"yyyy-MM-dd"**),  **new** SimpleDateFormat(**"yyyy年MM月dd日"**),  **new** SimpleDateFormat(**"yyyyMMdd"**)  };  @Override  **public** Object convertFromString(Map context, String[] values, Class toClass) {  *// 说明：  // context表示当前上下文  // values表示表单提交过来的值  // toClass表示要转换的目标类型。该类型是根据Action类的属性传递过来的。能传递过来的原因就是等会我们将配置Action类与转换器之间的联系。总之，对本例来说，toClass的类型需要是Date类型，否则就意味着出错了，直接返回null   // 若无请求参数，直接返回null* **if** (values == **null** || values.**length** <= 0) {  **return null**;  }  *// 若传递过来的不是时间类型，说明可能配置有误，也返回null* **if** (Date.**class** != toClass) {  **return null**;  }  *// 这时就需要对字符串进行封装了。要遍历所有支持的时间格式，若满足转换条件（即正常，没有异常），直接返回转换成功的值* **for** (DateFormat format : **df**) {  **try** {  **return** format.parse(values[0]);  } **catch** (ParseException ex) {  *// 说明转换出错，还要继续，因为可能满足下一个格式。* **continue**;  }  }  *// 最后都不满足，返回null。* **return null**;  }   @Override  **public** String convertToString(Map map, Object o) {  **return null**; *// 这个实际不需要写* } } |

（2）注册转换器类，告诉struts应用此转换器。

需要在使用该转换器的action类所在包下，新建一个配置文件，文件名要满足“Action类名-conversion.properties”，内容就写：“表单提交的名称=转换器类名全路径”。比如在com.zhang.action下新建“DemoAction-conversion.properties”，内容是：

|  |
| --- |
| **birthday**=**com.util.MyDateConverter** |

当然，如果是user对象的birthday属性，那么就写user.birthday=com.util.MyDateConverter。这种模型与动作类分开的情况，也可以在模型类的包下新建一个“模型类-conversion.properties”文件，然后其中写上“birthday=com.util.MyDateConverter”即可。

这样，就配置好了局部类型转换器。可以写Action和JSP了。比如Action中：

|  |
| --- |
| **public class** DemoAction {  **private** Date **birthday**;  **public** String add() {  System.***out***.println(**birthday**);  **return "success"**;  }   **public** Date getBirthday() {  **return birthday**;  }  **public void** setBirthday(Date birthday) {  **this**.**birthday** = birthday;  } } |

那么JSP上传birthday即可。大家自行测试。

上面配置的是局部类型转换器，该转换器只能给一个Action使用，如果想给多个Action使用，则可以用全局类型转换器。即在src下创建文件：xwork-conversion.properties，内容是“要转换的类型=转换器类”，即比如：

|  |
| --- |
| **java.util.Date**=**com.util.MyDateConverter** |

但是在实际开发中，基本不用这样麻烦，使用基本的“yyyy-MM-dd”这些“标准格式”传递数据即可。

# 4. Struts中的文件上传和下载

## 4.1 文件上传

使用Struts上传文件很方便，只要在Action类中写好有关此文件上传的三个属性，那么在Action中就可拿到上传文件的相关对象，而且这并不影响同时接收其他类型的参数（和以前相同）。Struts中是通过文件上传拦截器fileUpload实现的。

Struts默认限制文件上传的大小是2M，这是default.properties文件中的“struts.multipart.maxSize”常量默认设置的（单位是字节）。项目中一般根据实际情况修改该配置，否则上传文件大小超过该配置，则Struts会报错。例如设置成30M：

|  |
| --- |
| **struts.multipart.maxSize**=**31457280** |

这里的31457280是通过计算出来的，在配置文件中是不能直接写“30 \* 1024 \* 1024”的，因为这又不是Java代码。

下面举例来介绍文件上传。

（1）写好JSP文件，注意表单同样要设置为POST提交，并且设置enctype为“multipart/form-data”。例如下面的JSP中上传了一个文件和一个普通参数：

|  |
| --- |
| <%@ **page pageEncoding**="**UTF-8**" **language**="**java**" **isELIgnored**="**false**" %> <**html**> <**head**>  <**meta charset="UTF-8"** />  <**title**>上传文件</**title**> </**head**> <**body**>  <**form action="${**pageContext.request.contextPath**}/add.action" method="POST" enctype="multipart/form-data"**>  <**p**>提交人：<**input type="text" name="submitter"**></**p**>  <**p**>说明书文件：<**input type="file" name="specification"**></**p**>  <**p**><**input type="submit"**></**p**>  </**form**> </**body**> </**html**> |

（2）编写Action类。对于普通参数还是直接写属性，例如private String submitter。对于文件，需要用三个属性：

属性1是File对象，这对应上传文件的File对象，对象名要和表单中名称一致，例如private File specification，Struts直接把上传文件封装为File对象，使用十分方便。

属性2和属性3都是String类型，分别对应文件的名称和文件的MIME类型，这两个属性名称的命名规则分别是“表单中名称 + FileName”和“表单中名称 + ContentType”。

代码示例：

|  |
| --- |
| **package** com.zhang.action;  **import** java.io.File;  **public class** DemoAction {  *// 接收的普通参数* **private** String **submitter**;  *// 接收文件* **private** File **specification**; *// 文件* **private** String **specificationFileName**; *// 文件名* **private** String specificationContentType; *// 文件MIME类型* **public** String add() {  System.***out***.println(**submitter**);  System.***out***.println(**specification**.getAbsolutePath());  System.***out***.println(**specification**.getName());  System.***out***.println(**specificationFileName**);  System.***out***.println(specificationContentType);  **return "success"**;  }   *// 一定要写所有属性的setter和getter。这里省略...* } |

然后自己配置action。当我们访问action提交数据和文件时，发现输出以下内容：

|  |
| --- |
| 09:29:29.164 INFO org.apache.struts2.dispatcher.Dispatcher.getSaveDir(Dispatcher.java:698) - Unable to find 'struts.multipart.saveDir' property setting. Defaulting to javax.servlet.context.tempdir  张三  D:\apache-tomcat\work\Catalina\localhost\JavaWebDemo\upload\_489500ef\_fe73\_4a90\_bb87\_1fa62c9c368a\_00000001.tmp  upload\_489500ef\_fe73\_4a90\_bb87\_1fa62c9c368a\_00000001.tmp  证件照.jpg  image/jpeg |

即Tomcat拿到上传文件后，会生成一个临时文件，而File对象只是对应了这个临时文件。而FileName和ContentType得到的是上传文件的名称和类型。运行结果提示“没有配置struts.multipart.saveDir常量，因此临时文件存放在了默认目录”，这个也可自行配置。临时文件会在此次访问之后自动删除，因此我们无需处理。

例如我们将上传的文件保存在服务器的E:\upload目录下：

|  |
| --- |
| **package** com.zhang.action;  **import** org.apache.commons.io.FileUtils;  **import** java.util.UUID;  **public class** DemoAction {  *// 接收的普通参数* **private** String **submitter**;  *// 接收文件* **private** File **specification**; *// 文件* **private** String **specificationFileName**; *// 文件名* **private** String **specificationContentType**; *// 文件类型* **public** String add() **throws** IOException {  *// 得到E:\\upload* File uploadDir = **new** File(**"E:\\upload"**);  *// 若不存在此目录，则创建* **if** (!uploadDir.exists()) {  uploadDir.mkdirs();  }  *// 用 uuid + “真”文件名 作为保存文件的名称* File saveFile = **new** File(uploadDir, UUID.*randomUUID*().toString() + **"-"** +**specificationFileName**);  *// 由于Struts引用了commons-io包，包中提供了方便操作文件的方法。可直接将一个File复制到另一个File。  // 这里就将specification复制到要保存地方的saveFile。* FileUtils.*copyFile*(**specification**, saveFile);  **return "success"**;  }   *// 一定要写所有属性的setter和getter...* } |

上述还用到了commons-io包中的方法。commons-io是apache提供的组件，用于简化IO操作，可自行研究。

这样配置好action后，我们就能成功上传文件到E:\\upload目录下。也可以自行用流的方式复制文件，注意要关闭文件流。

同样，如果要用一个表单参数上传多个文件，那么动作类中用相应的数组类型接收就可以了（三个属性都用数组类型接收即可），然后处理时进行数组的遍历即可。

此外需要注意的是，还能通过在action中设置默认拦截器栈中fileUpload拦截器中的参数属性，限制上传文件的类型：

allowedTypesSet：设置允许上传的文件的MIME类型，多个用逗号分隔

allowedExtensionsSet：设置允许上传的文件的扩展名，多个用逗号分隔

例如：

|  |
| --- |
| <**action name="uploadFile" class="com.zhang.action.DemoAction" method="upload"**>  <**interceptor-ref name="defaultStack"**>  *<!-- 设置属性。只允许特定的类型上传 -->* <**param name="fileUpload.allowedExtensions"**>.jpg,.bmp,.png</**param**>  </**interceptor-ref**> </**action**> |

## 4.2 文件下载

文件下载本身就较简单，可以直接使用Servlet的方法（这时就必须使用ServletActionContext来获取response对象了）。若处理下载的action只负责返回下载的文件而不需跳转页面，那么action方法中直接返回null，然后action配置时不需要写result节点。例如：

|  |
| --- |
| **public class** DemoAction {  **public** String down() **throws** IOException {  File file = **new** File(**"E:\\upload\\安装包.exe"**);  HttpServletResponse response = ServletActionContext.*getResponse*();  *// 设置响应头信息* response.setContentLengthLong(file.length());  response.setHeader(**"Content-Disposition"**, **"attachment;filename="** + URLEncoder.*encode*(file.getName(), **"UTF-8"**));  *// FileUtils可直接将文件发送到输出流输出。很方便* FileUtils.*copyFile*(file, response.getOutputStream());  **return null**;  } } |

struts.xml中不需配置result（当然若有跳转页面，则可以配置）：

|  |
| --- |
| <**action name="down" class="com.zhang.action.DemoAction" method="down"**> </**action**> |

上述的方式相当于直接使用“Servlet”的方式进行文件下载的，现在我们用Struts提供的文件下载功能。

首先，Action类中提供一个InputStream类型的字段及其setter、getter方法，然后只要在处理方法中给这个InputStream设置要下载的输入流即可。下面的代码中也维护了一个fileName字段及其getter方法，并在处理方法中设置了fileName，这个属性的作用是在struts.xml中配置文件下载的文件名的，使文件下载显示的名称是处理方法提供的。代码如下：

|  |
| --- |
| **package** com.ssh.action;  **import** java.io.File; **import** java.io.FileInputStream; **import** java.io.FileNotFoundException; **import** java.io.InputStream;  **public class** FileDownload {  **private** InputStream **inputStream**; *// 定义一个输入流，但名字不能为in。* **private** String **fileName**;   **public** String download() {  **try** {  File file = **new** File(**"d:/a.txt"**);  **inputStream** = **new** FileInputStream(file);  **fileName** = file.getName();  } **catch** (FileNotFoundException e) {  e.printStackTrace();  }  **return "success"**;  }   *// 提供fileName的getter方法* **public** String getFileName() {  **return fileName**;  }  *// 提供getter和setter方法* **public** InputStream getInputStream() {  **return inputStream**;  }   **public void** setInputStream(InputStream inputStream) {  **this**.**inputStream** = inputStream;  } } |

struts.xml中设置为：

|  |
| --- |
| <**action name="download" class="com.ssh.action.FileDownload" method="download"**>  *<!-- type设置为stream -->* <**result name="success" type="stream"**>  *<!-- 指定动作类中输入流的属性名 -->* <**param name="inputName"**>inputStream</**param**>  *<!-- 指定浏览器以文件下载的方式打开，${fileName}实际上就是调用动作类中getFileName方法拿到下载的文件名 -->* <**param name="contentDisposition"**>attachment;filename=${fileName}</**param**>  *<!-- 指定下载文件的MIME类型，application/octet-stream表示二进制文件，则浏览器肯定会以下载文件的方式打开 -->* <**param name="contentType"**>application/octet-stream</**param**>  </**result**> </**action**> |

其实在以前的文件下载中也能设置contentType的值。使用Struts提供的文件下载，只要指定文件输入流即可，比较简便，输入流会自动由框架负责关闭。

# 5. 结语

至此，Struts基本使用已经讲完，应付大部分开发已经没有问题。关于Struts还有较多的知识，比如：输入验证、ONGL表达式、值栈（StackValue）和Struts标签等，这些就不深入讨论了。大家可参照Struts官方文档自行研究。

但这里再提一下输入验证问题。Struts本身提供了输入验证的功能，能够验证用户提交的数据符合系统要求，但是现在一般不使用Struts进行验证了。实际上Struts的验证属于服务器端的验证，在开发中，客户端和服务器端都要进行数据验证，因为仅仅用客户端验证是“防君子不防小人”的。服务器端验证无论何时都必须写。

由于近几年Struts出现较多安全漏洞，使用人数较少，目前主流趋势是使用Spring框架。Spring可以说提供了Java EE的一整套解决方案，是Java EE的“事实标准”。因此也不推荐大家深入研究Struts，以后可重点深入研究后面要讲的Spring，因此这里对于Struts只要达到应用即可。