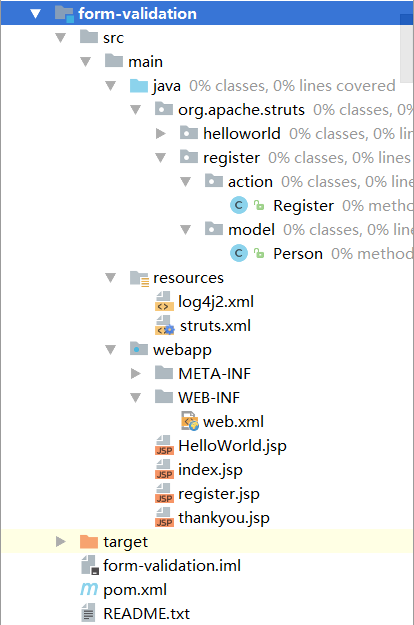
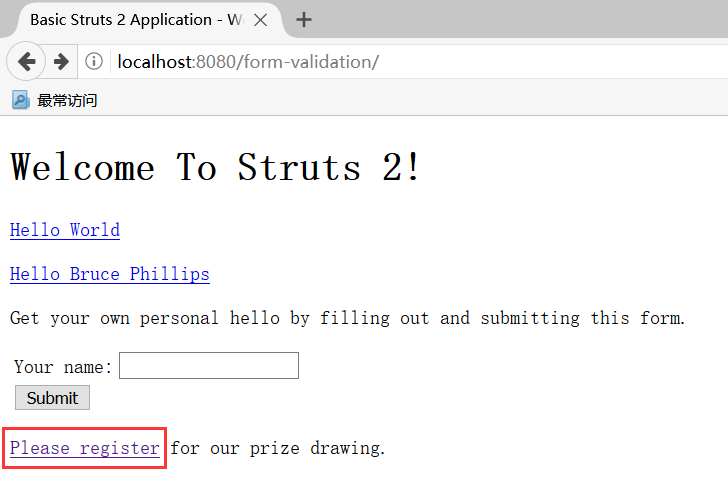
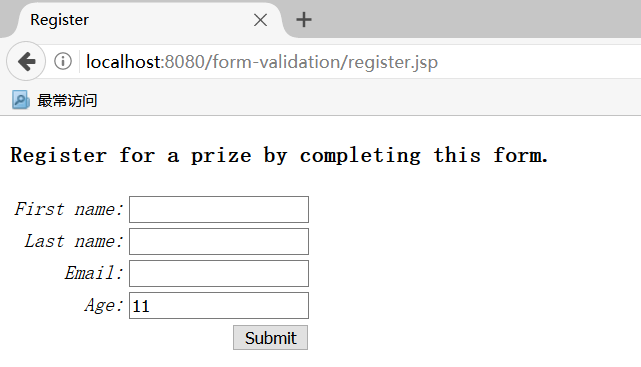
重写validate方法来验证form表单

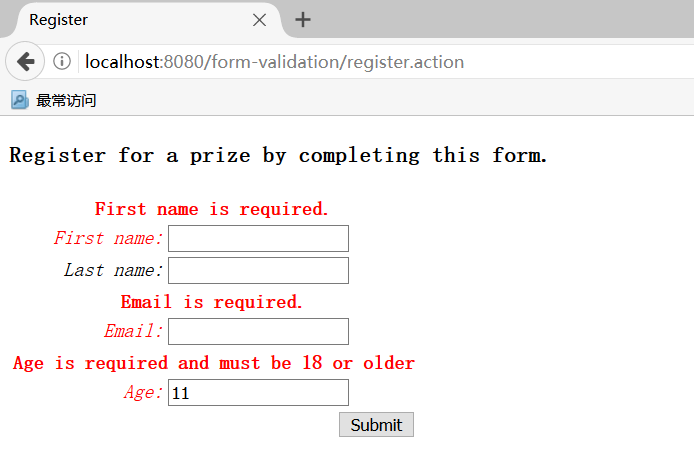
# 项目包机构



# **运行效果**







# **分析代码**

点击逐步进入注册界面：

<**s:form action="register"**>  
  
 <**s:textfield name="personBean.firstName" label="First name"** />  
 <**s:textfield name="personBean.lastName" label="Last name"** />  
 <**s:textfield name="personBean.email" label ="Email"**/>   
 <**s:textfield name="personBean.age" label="Age"** />  
   
 <**s:submit**/>  
   
</**s:form**>

点击提交，调用register action的execute方法前，都会经过一个拦截器ValidationInterceptor，这是一个使用java方法的验证form字段的拦截器，它的父类是MethodFilterInterceptor；

进入该拦截器中会调用对应action的validate()方法，如果不重写该方法，那么就会调用父类ActionSupport的validate()方法，该方法是空的实现；

**public class** Register **extends** ActionSupport ===>>

**public void** validate(){  
  
 **if** ( **personBean**.getFirstName().length() == 0 ){  
  
 addFieldError( **"personBean.firstName"**, **"First name is required."** );  
  
 }  
  
  
 **if** ( **personBean**.getEmail().length() == 0 ){  
  
 addFieldError( **"personBean.email"**, **"Email is required."** );  
  
 }  
  
 **if** ( **personBean**.getAge() < 18 ){  
  
 addFieldError( **"personBean.age"**, **"Age is required and must be 18 or older"** );  
  
 }  
  
  
}

在validate方法中对传入的参数进行后台校验，对不满足条件的字段，调用addFieldError(字段，提示信息)；该方法是ActionSupport中定义好的方法；

**public class** ActionSupport ===>>

**public void** addFieldError(String fieldName, String errorMessage) {  
 **validationAware**.addFieldError(fieldName, errorMessage);  
}

**其中validationAware是一个工具类，在每次请求action时，都会在自动创建实例**ValidationAwareSupport ==>>

**public synchronized void** addFieldError(String fieldName, String errorMessage) {  
 **final** Map<String, List<String>> errors = internalGetFieldErrors();  
 List<String> thisFieldErrors = errors.get(fieldName);  
  
 **if** (thisFieldErrors == **null**) {  
 thisFieldErrors = **new** ArrayList<>();  
 errors.put(fieldName, thisFieldErrors);  
 }  
  
 thisFieldErrors.add(errorMessage);  
}

**private** Map<String, List<String>> internalGetFieldErrors() {  
 **if** (**fieldErrors** == **null**) {  
 **fieldErrors** = **new** LinkedHashMap<>();  
 }  
  
 **return fieldErrors**;  
}

**可以看到它将我们在validate方法中定义的验证信息都放进**fieldErrors里面了。

回到上面的ValidationInterceptor拦截器：

@Override  
**protected** String doIntercept(ActionInvocation invocation) **throws** Exception {  
 doBeforeInvocation(invocation);  
 **return** invocation.invoke();  
}

可以看到里面执行了doBeforeInvocation，接着调用invocation.invoke()

**protected void** doBeforeInvocation(ActionInvocation invocation) **throws** Exception {  
 Object action = invocation.getAction();  
 ActionProxy proxy = invocation.getProxy();  
  
 *//the action name has to be from the url, otherwise validators that use aliases, like  
 //MyActio-someaction-validator.xml will not be found, see WW-3194  
 //UPDATE: see WW-3753* String context = **this**.getValidationContext(proxy);  
 String method = proxy.getMethod();  
  
 **if** (***LOG***.isDebugEnabled()) {  
 ***LOG***.debug(**"Validating {}/{} with method {}."**, invocation.getProxy().getNamespace(), invocation.getProxy().getActionName(), method);  
 }  
   
  
 **if** (**declarative**) {  
 **if** (**validateAnnotatedMethodOnly**) {  
 **actionValidatorManager**.validate(action, context, method);  
 } **else** {  
 **actionValidatorManager**.validate(action, context);  
 }  
 }   
   
 **if** (action **instanceof** Validateable && **programmatic**) {  
 *// keep exception that might occured in validateXXX or validateDoXXX* Exception exception = **null**;   
   
 Validateable validateable = (Validateable) action;  
 ***LOG***.debug(**"Invoking validate() on action {}"**, validateable);  
  
 **try** {  
 PrefixMethodInvocationUtil.*invokePrefixMethod*(invocation, **new** String[]{***VALIDATE\_PREFIX***, ***ALT\_VALIDATE\_PREFIX***});  
 }  
 **catch**(Exception e) {  
 *// If any exception occurred while doing reflection, we want   
 // validate() to be executed* ***LOG***.warn(**"an exception occured while executing the prefix method"**, e);  
 exception = e;  
 }  
   
   
 **if** (**alwaysInvokeValidate**) {  
 validateable.validate();  
 }  
   
 **if** (exception != **null**) {   
 *// rethrow if something is wrong while doing validateXXX / validateDoXXX* **throw** exception;  
 }  
 }  
}

大致说下该方法里面做了哪些事情：

invocation这里是DefaultActionInvocation，持有action和拦截器实例

proxy这里是StrutsActionProxy，它是一个介于xwork和action层之间的额外层，这个代理不同情况下是不同的，可以是远程代理，rmi,soap;

method这里是execute,也就是我们调用register方法中的execute;

declarative,默认为true,意思是基于xml配置和注解配置的验证规则是否执行；

validateAnnotatedMethodOnly，默认为false,意思是是否仅注解验证

actionValidatorManager，这里是AnnotationActionValidatorManager，注解验证器

alwaysInvokeValidate，默认为true,表示是否始终调用validate()方法

validateable.validate()，执行action中定义的validate方法

上面这些步骤执行完成之后，接着执行invocation.invoke();

**public** String invoke() **throws** Exception {  
 String profileKey = **"invoke: "**;  
 **try** {  
 UtilTimerStack.*push*(profileKey);  
  
 **if** (**executed**) {  
 **throw new** IllegalStateException(**"Action has already executed"**);  
 }  
  
 **if** (**interceptors**.hasNext()) {  
 **final** InterceptorMapping interceptorMapping = **interceptors**.next();  
 String interceptorMsg = **"interceptorMapping: "** + interceptorMapping.getName();  
 UtilTimerStack.*push*(interceptorMsg);  
 **try** {  
 Interceptor interceptor = interceptorMapping.getInterceptor();  
 **if** (interceptor **instanceof** WithLazyParams) {  
 interceptor = **lazyParamInjector**.injectParams(interceptor, interceptorMapping.getParams(), **invocationContext**);  
 }  
 **resultCode** = interceptor.intercept(DefaultActionInvocation.**this**);  
 } **finally** {  
 UtilTimerStack.*pop*(interceptorMsg);  
 }  
 } **else** {  
 **resultCode** = invokeActionOnly();  
 }  
  
 *// this is needed because the result will be executed, then control will return to the Interceptor, which will  
 // return above and flow through again* **if** (!**executed**) {  
 **if** (**preResultListeners** != **null**) {  
 ***LOG***.trace(**"Executing PreResultListeners for result [{}]"**, **result**);  
  
 **for** (Object preResultListener : **preResultListeners**) {  
 PreResultListener listener = (PreResultListener) preResultListener;  
  
 String \_profileKey = **"preResultListener: "**;  
 **try** {  
 UtilTimerStack.*push*(\_profileKey);  
 listener.beforeResult(**this**, **resultCode**);  
 }  
 **finally** {  
 UtilTimerStack.*pop*(\_profileKey);  
 }  
 }  
 }  
  
 *// now execute the result, if we're supposed to* **if** (**proxy**.getExecuteResult()) {  
 executeResult();  
 }  
  
 **executed** = **true**;  
 }  
  
 **return resultCode**;  
 }  
 **finally** {  
 UtilTimerStack.*pop*(profileKey);  
 }  
}

大致意思是，不断的获取下一个拦截器进行处理；

处理代码

**resultCode** = interceptor.intercept(DefaultActionInvocation.**this**);

处理完成之后**resultCode =“input”,**

<**action name="register" class="org.apache.struts.register.action.Register" method="execute"**>  
<**result name="success"**>/thankyou.jsp</**result**>  
<**result name="input"**>/register.jsp</**result**>  
 </**action**>