

轻量级 J2EE 框架应用

E 4 A Simple Controller with handling result view

学号: SA16225221

姓名: 欧勇

报告撰写时间: 2016/12/16

1. 主题概述

简要介绍主题的核心内容，作业内容：

1 定义一个 `success_view.xml` 作为 Action 请求成功后的结果视图。该视图分成 `<header>` 与 `<body>` 两部分；`<header>` 定义视图标题，`<body>` 定义视图内容。格式可参考如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<view>
  <header>
    <title>a viewer in MVC</title>
  </header>
  <body>
    <form>
      <name>logoutForm</name>
      <action>logout.action</action>
      <method>post</method>
      <textview>
        <name>userName</name>
        <label>Login Name:</label>
        <value>Water</value>
      </textview>
      <textview>
        <name>userAge</name>
        <label>Age</label>
        <value>30</value>
      </textview>
      <buttonview>
        <name>logoutButton</name>
        <label>Logout</label>
      </buttonview>
    </form>
  </body>
</view>
```

1.1 在 `success_view.xml` 的 `<body>` 中定义一个表单 `<form>`，表单属性有 `name`\`action`\`method`。在 `<form>` 可以定义视图组件，如 `<textview>`\`<buttonview>`\`<checkboxview>`等。

1.2 每个视图组件可以根据用途不同，定义属性 `name`\`label`\`value`\`method` 等。

2 基于 E3，将 `success_view.xml` 作为 login action 的 success 结果视图。参考代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<action-controller>

  <interceptor>
    <name>logWriter</name>
    <class>
      <name>water.servlet.interceptor.LogWriter</name>
      <method>log</method>
    </class>
  </interceptor>

  <!--other interceptors -->

  <action>
    <name>login</name>
    <class>
      <name>water.servlet.action.LoginAction</name>
      <method>login</method>
    </class>
    <interceptor-ref>
      <name>logWriter</name>
    </interceptor-ref>
    <result>
      <name>success</name>
      <type>forward</type>
      <value>pages/success_view.xml</value>
    </result>
    <result>
      <name>fail</name>
      <type>redirect</type>
    </result>
  </action>
</action-controller>
```

- 3 当客户端请求 login action 时，如果返回 result 的值为 success，再根据 success_view.xml 生成推送至客户端的视图。即，根据 success_view.xml 的定义，生成浏览器可以执行的 html 页面（可以参考 XSLT）的使用）。如，将 success_view.xml 中的<buttonView>节点翻译成 html 中的<input>节点。
- 4 测试以上任务实现结果。
- 5 请描述 Struts 框架中视图组件的工作方式，如<s:form>/<s:submit>。

2. 假设

主题内容所参照的假设条件，或假定的某故事场景，如调试工具或软硬件环境

开发环境：

Win10

Eclipse kepler

JDK 1.8

Tomcat 7.0

Chrome 浏览器

3. 实现或证明

对主题内容进行实验实现, 或例举证明, 需描述实现过程及数据。如对 MVC 中 Controller 功能的实现及例证 (图示、数据、代码等)

注: 本作业报告在第三次作业报告基础上修改添加而成, 大部分保持上次作业内容, 修改部分为: 在 WebContent/pages 下添加了 success_view.xml 和 success_xsl.xml 文件 (具体代码请看第 11,12 页), 登录成功后的跳转界面 (16 页)。

流程:

假设用户名为 world, 密码为 hello

为了方便查看, 采用 get 方式提交, 可以通过浏览器 url 看到输入的用户名和密码

(因为若采用 post 方式则无法通过 url 看到用户名和密码, 所以采用 get 方式提交)

若登录成功则展示 success_view.xml 配置的成功页面, 页面为退出功能的 form 表单

若登录失败则跳转 login_fail.jsp 页面, 页面显示 Login Fail 的字符串

若使用未知的 action 提交, 既 action="unknow.scaction" 则无法找到相应的方法处理, 则跳转 error_action.jsp 页面, 页面显示 "不可识别的 action 请求" 提示字符串。

若返回的是未知的处理结果, 则返回 error_result.jsp 页面, 页面显示 "没有请求的资源" 提示字符串

图 1: 项目目录结构, 可以看出项目名称为 SimpleController, src 文件夹下有一个 controller.xml 配置文件, 其中记录有 action 和 interceptor 的配置信息, 其中 action 标签中会有 result 相关的配置, 当启用拦截器时则在 action 中定义 interceptor-ref, 否则视为不启用拦截器功能。

同时还有名为 me.king 的包, 其下有 LoginController 作为控制层, UserBean 作为模型层, 同时定义 LoginInterface 接口, 以及动态代理的处理类 LoginActionHandler。

在包 me.king.interceptor 中有负责日志记录的 LogWriter 类。

然后还有 5 个 jsp 页面作为视图层, 分别是 login_fail.jsp, login_success.jsp, error_action.jsp, error_result.jsp 和 login.jsp。以及使用配置文件 success_view.xml 和 success_xsl.xml 的成功登录的视图。

注意在 WEB-INF/lib 下需要导入 dom4j 的 jar 包, 若仅仅只是将 jar 包放入 Java Resources/Libraries 中, 则在编译时能通过, 但是却无法完成处理, 因为会在执行到语句 new SAXReader() 时报如下错误, 提示找不到对应的类。

```

root cause
java.lang.NoClassDefFoundError: org/dom4j/io/SAXReader
    me.king.LoginController.doGet(LoginController.java:48)
    javax.servlet.http.HttpServlet.service(HttpServlet.java:621)
    javax.servlet.http.HttpServlet.service(HttpServlet.java:728)
    org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:51)

```

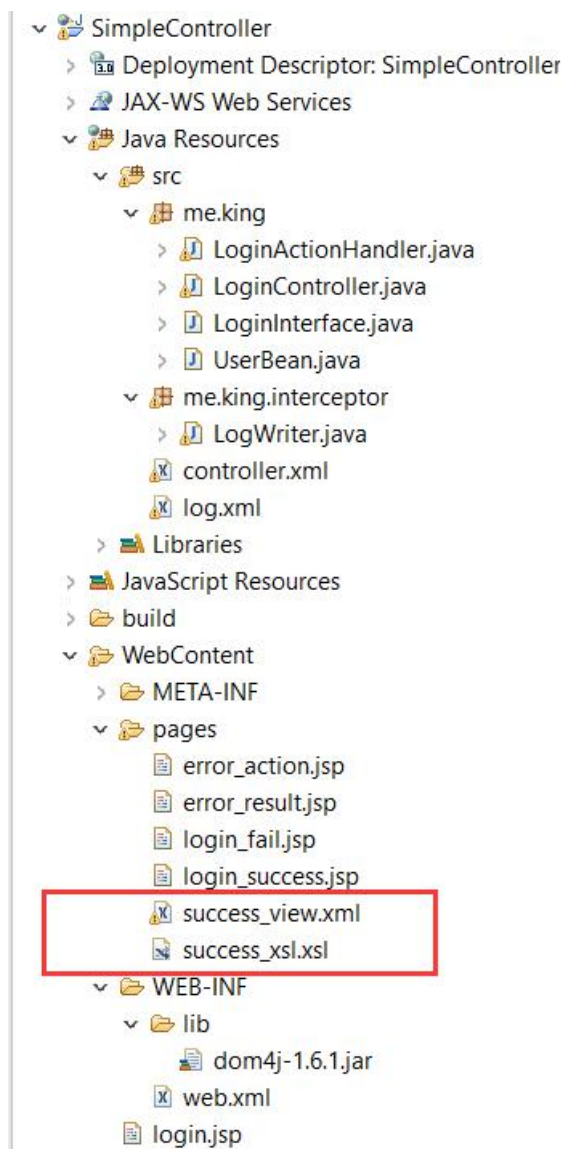


图 1: 项目目录结构

图 2.1: web.xml 配置文件截图

不使用注解的方式告知容器,而是用配置文件的方式配置控制器 `LoginController` 的映射路径,其中,将 `login.jsp` 配置为默认页面,将 `servlet` 控制层类 `LoginController` 映射名为同类名,同时,对所有以 `.saction` 结尾的 url 请求进行转发和控制。

```

LoginController.java login.jsp *web.xml controller.xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instar
  <display-name>SimpleController</display-name>
  <welcome-file-list>
    <welcome-file>login.jsp</welcome-file>
  </welcome-file-list>
  <servlet>
    <servlet-name>LoginController</servlet-name>
    <servlet-class>me.king.LoginController</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>LoginController</servlet-name>
    <url-pattern>*.saction</url-pattern>
  </servlet-mapping>
</web-app>
    
```

图 2.1: web.xml 配置文件截图

图 2.1: LoginController 代码概览截图

在 doGet 方法中进行控制转发，其他采用默认，若前端页面采用 post 方式提交，则在 doPost 方法中也需要进行转发处理，本次采用直接调用 doGet()方法进行处理。

定义私有 List<Element>类型变量 actions 和 itcpts, 用于保存所有的 action 标签和所有的 interceptor 标签，这两个变量在 init 方法中获取。

同时覆盖 HttpServlet 类中的 init()方法，用于当有用户首次访问此 servlet 的时候进行配置文件的读取操作，此操作直到本 servlet 销毁之前都会存在，这样后续的其他用户使用此 servlet 时则不会重复读取配置文件 controller.xml 了。

当然这样也无法避免在控制层代码中包含有 dom4j 的解析代码，这时可以写一个 XmlParse 类用于解析 xml 文件，同时返回一个 Action 类和 Interceptor 类用于保存解析的标签信息。

```

LoginController.java controller.xml LogWriter.java
2
3* import java.io.IOException;[]
17
18/**
19 * Servlet implementation class LoginController
20 */
21 // @WebServlet("/LoginController")
22 public class LoginController extends HttpServlet {
23     private static final long serialVersionUID = 1L;
24     private List<Element> actions = null; //保存所有的action标签
25     private List<Element> itcpts = null; //保存所有的interceptor标签
26
27     /**
28      * @see HttpServlet#HttpServlet()
29      */
30     public LoginController() {
31         super();
32         // TODO Auto-generated constructor stub
33     }
34
35     public void init() { // 不要重写init(ServletConfig config)方法, []
36
37
38
39
40
41
42
43
44
45
46
47
48
49     protected void doResult(HttpServletRequest request, HttpServletResponse response, Element crtAction, String rstName) throws Exception {[]
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65     /**
66      * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse
67      * response)
68      */
69     protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {[]
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
    
```

图 2.1: LoginController 代码概览截图

图 2.2: LoginController 类 init 方法代码截图

```

35 @Override
36 public void init() { // 不要重写init(ServletConfig config)方法,
37     try { // 获取controller.xml配置文件, 并以流的形式读入
38         InputStream is = this.getClass().getResourceAsStream( "/controller.xml");
39         Document dc = (new SAXReader()).read(is); // 获取文档对象
40         this.actions = dc.getRootElement().elements("action"); // 获取所有的action标签并加入到list中
41         this.itcptps = dc.getRootElement().elements("interceptor"); // 获取所有的interceptor标签并加入到list中
42     } catch (Exception e) {
43         // TODO Auto-generated catch block
44         e.printStackTrace();
45     }
46     System.out.println("初始化读取配置文件controller.xml完成");
47 }

```

图 2.2: LoginController 类 init 方法代码截图

图 2.3: LoginController 类中 doGet 方法代码截图

doGet 中主要控制转发流程为:

1. 获取 request 中的 action 请求名,
2. 遍历 action 标签 list 并判断是否找到需要的 action
3. 若找不到对应 action 请求的时候跳转 error_action 页面并推出 doGet 方法
4. 否则解析当前对应的 action 标签, 获取 name 和 method 标签
5. 判断是否在 action 标签中配置了拦截器 interceptor-ref 标签
6. 若存在 interceptor-ref 标签且非空则去找到对应的 interceptor 标签中设置的 name 和 method
7. 利用反射机制实例化对应的类为 cls 对象
8. 生成处理对象 handler 和代理对象 proxy
9. 获取代理对象中的 LogWriter 类, 使用代理对象调用 method 方法并获取返回结果
10. 当未使用拦截器时则跳过动态代理的部分, 直接使用反射机制获取类和方法并调用。
11. 执行自定义的返回结果给客户端的 doResult 函数

其中, try-catch 中的类型必须为 Exception, 不能为 DocumentException, 否则会在编译的时候报: `Can't load IA 32-bit .dll on a AMD 64-bit platform` 错误

```
java.lang.UnsatisfiedLinkError: D:\Program Files\apache-tomcat-7.0.47\bin\ntnative-1.dll: Can't load IA 32-bit .dll on a AMD 64-bit platform
```

同时, 在最后执行转发操作中, 使用 `response.sendRedirect()` 方法时, 需要使用相对定位, 否则会报: `Path pages/login_success.jsp does not start with a "/" character` 错误, 并且页面显示 404 错误码, 找不到资源。




```

69 protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
70     // TODO Auto-generated method stub
71
72     // 获取请求的action
73     String url = request.getRequestURL().toString();
74     1. String actionName = url.substring(url.lastIndexOf('/') + 1, url.lastIndexOf(".")).toString();
75     try { // 已经在init方法中将读入controller.xml的工作完成
76         Element crtAction = null; // 保存当前执行的action标签
77         boolean isFindAction = false;
78         for (Element ele : actions) {
79             2. if (actionName.equals(ele.elementText("name"))) {
80                 isFindAction = true; // 将标志位设置为成功找到该action
81                 crtAction = ele;
82             }
83         }
84         3. if (!isFindAction) { // 当找不到对应action请求的时候跳转error_action页面
85             response.sendRedirect("./pages/error_action.jsp");
86             return; // 退出doGet方法
87         }
88
89         // 则表示找到action, 此时开始解析该action
90         4. String className = crtAction.element("class").elementText("name"),
91             classMethod = crtAction.element("class").elementText("method"), rstName = null, // 用于保存结果字符串
92             userName = request.getParameter("name"), // 用户名
93             userPwd = request.getParameter("pwd"); // 用户密码
94         Class<?> cls = null; // 用于保存反射类
95         Method m = null; // 用于保存反射类的方法
96
97         Element itcpt = crtAction.element("interceptor-ref");
98         5. if (itcpt != null) { // 若存在interceptor-ref标签且非空
99             String itcptCls = null; // 拦截器类
100            String itcptMtd = null; // 拦截器类的方法
101
102            for (Element itcpt : itcpts) { // 找到对应的interceptor的定义
103                6. if (itcpt.elementText("name").equals(itcpt.elementText("name"))) {
104                    itcptCls = itcpt.element("class").elementText("name");
105                    itcptMtd = itcpt.element("class").elementText("method");
106                }
107            }
108            7. cls = Class.forName(className); // 利用反射机制实例化对应的类
109            Object actObj = (LoginInterface) cls.newInstance();
110
111            // 生成处理对象, 传入实例对象, action名, 拦截器类名, 拦截器方法 (此处既logWriter类和log方法)
112            LoginActionHandler handler = new LoginActionHandler(actObj, actionName, itcptCls, itcptMtd);
113            8. Object proxy = Proxy.newProxyInstance( // 生成代理对象
114                actObj.getClass().getClassLoader(), // 传入类加载器,
115                actObj.getClass().getInterfaces(), // 类接口,
116                handler); // 以及真正进行处理的对象
117
118            9. cls = proxy.getClass(); // 此处为LogWriter
119            // 指定获取当前类的classMethod方法, 同时指定参数列表的类型为string, string
120            m = cls.getDeclaredMethod(classMethod, new Class[] {String.class, String.class });
121            rstName = (String) m.invoke(proxy, new Object[] { userName, userPwd });
122        } else { // 未使用拦截器
123            10. cls = Class.forName(className); // 利用反射机制实例化对应的类
124            m = cls.getDeclaredMethod(classMethod, new Class[] { String.class, String.class });
125            rstName = (String) m.invoke(cls.newInstance(), new Object[] { userName, userPwd });
126        }
127
128        11. doResult(request, response, crtAction, rstName); // 将结果返回至客户端
129    } catch (Exception e) {
130        // TODO Auto-generated catch block
131        e.printStackTrace();
132    }
133 }

```

图 2.3: LoginController 类中 doGet 方法代码截图

图 2.4: LoginController 类中 doResult 方法代码截图

在此方法中, 对 result 的类型进行判断并使用不同的方式返回结果视图给客户端
 将返回结果与 result 标签 list 中的对应 name 对比
 获取对应 result 标签中的 value (返回地址), type (跳转还是内部重定向)
 最后执行返回结果, 若返回的字符串类型不匹配, 则返回特定页面

```

499 protected void doResult(HttpServletRequest request, HttpServletResponse response, Element crtAction, String rstName) throws Exception {
500     List<Element> rst = crtAction.elements("result");// 所有的result标签
501     boolean isErrorRst = true;
502     for (Element rstEle : rst) { // 一个rstEle对应一个result节点
503         if (rstName.equals(rstEle.elementText("name"))) { // 当存在对应result返回结果的配置信息的时候
504             isErrorRst = false; // 设置result结果错误标志为false
505             String rstValue = rstEle.elementText("value");
506
507             if (rstEle.elementText("type").equals("forward"))
508                 getServletContext().getRequestDispatcher(rstValue).forward(request, response); // forward为内部重定向
509             else response.sendRedirect('.' + rstValue); // 重定向根据相对地址跳转
510         }
511     }
512     if (isErrorRst) response.sendRedirect("./pages/error_result.jsp"); // 不存在对应result的时候跳转至error_result页面
513 }

```

图 2.4: LoginController 类中 doResult 方法代码截图

```

log.xml success_view.xml success_xsl.xsl controller.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <action-controller>
3 <interceptor>
4     <name>logWriter</name>
5 <class>
6     <name>me.king.interceptor.LogWriter</name>
7     <method>log</method>
8 </class>
9 </interceptor>
10
11 <action>
12     <name>login</name>
13 <class>
14     <name>me.king.UserBean</name>
15     <method>checkLogin</method>
16 </class>
17
18 <interceptor-ref>
19     <name>logWriter</name>
20 </interceptor-ref>
21
22 <result>
23     <name>success</name>
24     <type>forward</type>
25     <value>/pages/success view.xml</value>
26 </result>
27 <result>
28     <name>fail</name>
29     <type>redirect</type>
30     <value>/pages/login_fail.jsp</value>
31 </result>
32 </action>
33 <action>
34     <name>register</name>
35 </action>
36 </action-controller>

```

图 2.5: controller.xml 配置文件截图

图 2.6: success_view.xml 配置文件截图

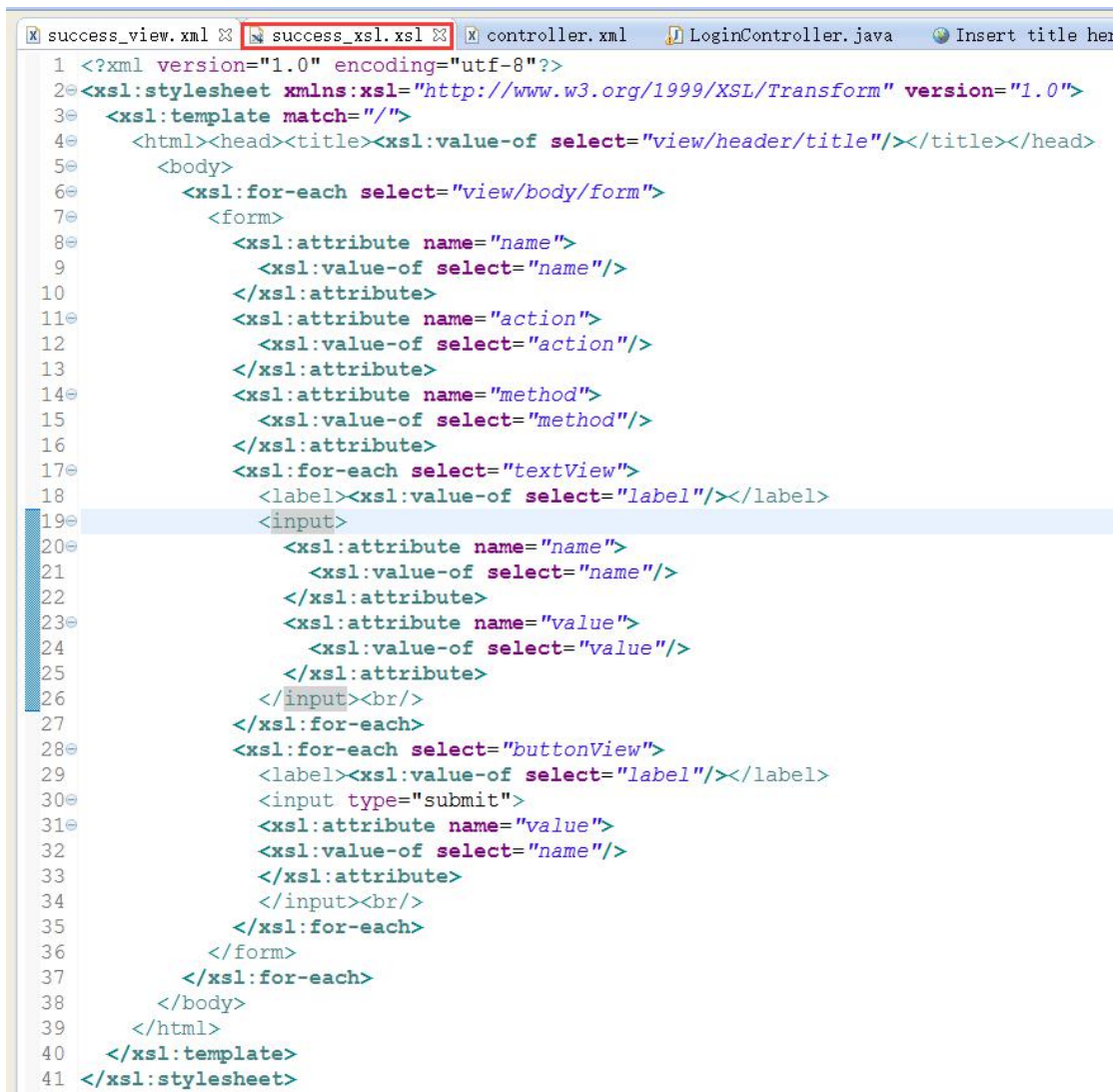
本次作业没有自己实现 xml 到 html 的转化函数，而是运用了 xslt 提供的快捷转化方式，即在 xml 配置文件中直接指定 xsl 文件位置。

注意，这里的 href 的未知是相对 WebContent 的，而不是 xml 自身的位置。所以必须写上目录名，否则前端页面会报 404 找不到 xsl 文件的错误。



```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <?xml-stylesheet type="text/xsl" href="./pages/success_xsl.xsl"?>
3 <view>
4   <header>
5     <title>a viewer in MVC</title>
6   </header>
7   <body>
8     <form>
9       <name>logoutForm</name>
10      <action>logout.action</action>
11      <method>post</method>
12      <textview>
13        <name>userName</name>
14        <label>Login Name:</label>
15        <value>KING</value>
16      </textview>
17      <textview>
18        <name>userAge</name>
19        <label>Age</label>
20        <value>30</value>
21      </textview>
22      <buttonview>
23        <name>logoutButton</name>
24        <label>Logout</label>
25      </buttonview>
26    </form>
27  </body>
28 </view>
```

图 2.6: success_view.xml 配置文件截图



```

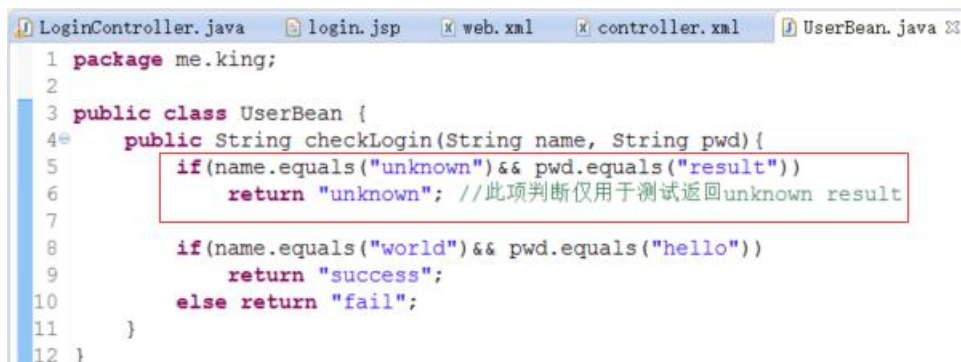
1 <?xml version="1.0" encoding="utf-8"?>
2 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
3   <xsl:template match="/">
4     <html><head><title><xsl:value-of select="view/header/title"/></title></head>
5     <body>
6       <xsl:for-each select="view/body/form">
7         <form>
8           <xsl:attribute name="name">
9             <xsl:value-of select="name"/>
10          </xsl:attribute>
11          <xsl:attribute name="action">
12            <xsl:value-of select="action"/>
13          </xsl:attribute>
14          <xsl:attribute name="method">
15            <xsl:value-of select="method"/>
16          </xsl:attribute>
17          <xsl:for-each select="textView">
18            <label><xsl:value-of select="label"/></label>
19            <input>
20              <xsl:attribute name="name">
21                <xsl:value-of select="name"/>
22              </xsl:attribute>
23              <xsl:attribute name="value">
24                <xsl:value-of select="value"/>
25              </xsl:attribute>
26            </input><br/>
27          </xsl:for-each>
28          <xsl:for-each select="buttonView">
29            <label><xsl:value-of select="label"/></label>
30            <input type="submit">
31              <xsl:attribute name="value">
32                <xsl:value-of select="name"/>
33              </xsl:attribute>
34            </input><br/>
35          </xsl:for-each>
36        </form>
37      </xsl:for-each>
38    </body>
39  </html>
40 </xsl:template>
41 </xsl:stylesheet>

```

图 2.7: success_xsl.xml 配置文件截图

图 3: UserBean 代码截图,

定义一个简单的 `checkLogin` 方法, 参数为 `name` 和 `pwd`, 函数体内直接对比两个字符串, 若正确则返回"success", 否则返回"fail". 同时为了测试方便, 当 `name` 为 `unknown` 并且 `pwd` 为 `result` 的时候返回"unknown".



```

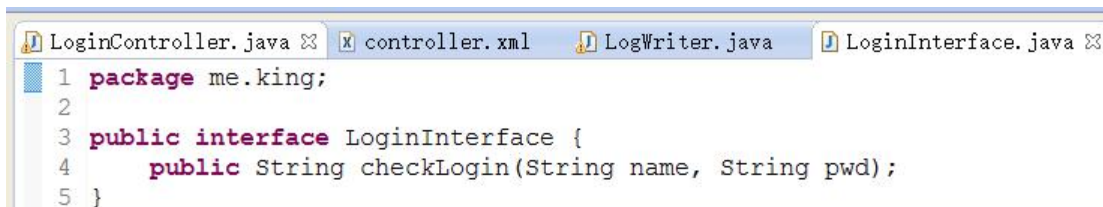
1 package me.king;
2
3 public class UserBean {
4   public String checkLogin(String name, String pwd){
5     if(name.equals("unknown") && pwd.equals("result"))
6       return "unknown"; //此项判断仅用于测试返回unknown result
7
8     if(name.equals("world") && pwd.equals("hello"))
9       return "success";
10    else return "fail";
11  }
12 }

```

图 3: UserBean 代码截图

图 3.1: LoginInterface 接口代码截图

此接口只定义了验证登录操作的方法 `checkLogin`，若使用 JDK 的动态代理机制，则必须采用接口的方式，这样的动态代理的实现方式有缺陷，那就是必须由接口，若由很多不同的类需要采用动态代理的时候，则会使接口爆炸式的增长，不利于维护。这时，可以采用继承的动态代理的实现机制来避免这个问题。具体做法请看参考资料[10]。



```

1 package me.king;
2
3 public interface LoginInterface {
4     public String checkLogin(String name, String pwd);
5 }

```

图 3.1: LoginInterface 接口代码截图

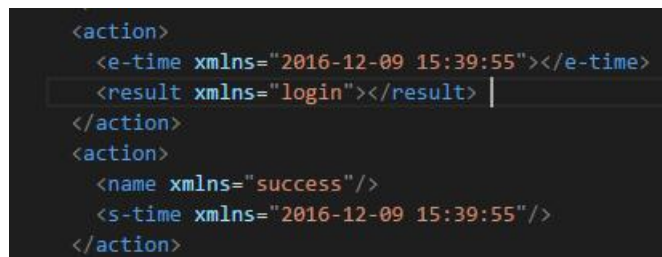
图 3.2: LoginActionHandler 代码截图

定义 `LoginActionHandler` 类的构造函数，需要真正执行的类对象，以及用于记录日志的 `action` 名，实际执行 `action` 的类的全名，包括包名，真正需要执行的 `action` 方法。

然后覆盖 `invoke` 方法，需要传入代理类 `proxy` 的实例对象，以及真正执行的方法，以及对应的参数。最后需要将结果返回。

在这里注释有写到：此处不能使用默认的 `act.newInstance()`；否则每次调用 `invoke` 会新建一个 `action` 标签，

即：`actM.invoke(act.newInstance(), this.action, false)`；会出现如下情况：

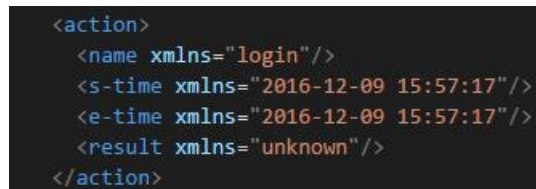


```

<action>
  <e-time xmlns="2016-12-09 15:39:55"></e-time>
  <result xmlns="login"></result>
</action>
<action>
  <name xmlns="success"/>
  <s-time xmlns="2016-12-09 15:39:55"/>
</action>

```

其中 `name` 和 `result` 分别在一个 `action` 标签中，使用同一个 `obj` 而不是 `actM.newInstance()` 的结果才是正确的在同一个 `action` 标签中：



```

<action>
  <name xmlns="login"/>
  <s-time xmlns="2016-12-09 15:57:17"/>
  <e-time xmlns="2016-12-09 15:57:17"/>
  <result xmlns="unknown"/>
</action>

```

调用处理器的 `handler` 实现类，每次生成动态代理类对象时都需要指定一个实现了 `InvocationHandler` 接口的调用处理器对象。

其中在代理真实对象前，已经在构造函数中初始化了响应的一些对象，包括利用反射初始化的 `LogWriter` 类和 `log` 方法。

```

1 package me.king;
2
3 *import java.lang.reflect.InvocationHandler;[]
4
5
6
7
8
9 //import me.king.action.LoginAction;
10
11 public class LoginActionHandler implements InvocationHandler {
12     private Class<?> act;
13     private Method actM;
14     private Object target; //真正的执行登录验证的对象
15     private String action; //保存执行任务的action名
16     private Object ob; //用于保存logWriter的对象
17
18     public LoginActionHandler(Object target, String action, String actionName, String actionMethod) throws Exception{
19         this.action = action;
20         this.target = target;
21         this.act = Class.forName(actionName); //此处的actionName实际为LogWriter类的全名
22         this.actM = act.getDeclaredMethod(actionMethod, new Class[] {String.class, boolean.class}); //此处为log方法
23         this.ob = act.newInstance();
24     }
25
26     @Override
27     public Object invoke(Object proxy, Method method, Object[] args)
28         throws Throwable {
29         // TODO Auto-generated method stub
30
31         //将action name和传入logWriter, 同时true表示是开始记录日志, false表示结束记录日志
32         //同时此处不能使用默认的act.newInstance(); 否则每次调用invoke会新建一个action标签
33         //actM.invoke(act.newInstance(), this.action, false);
34         actM.invoke(ob, action, true);
35         String result = (String) method.invoke(target, args);
36         actM.invoke(ob, result, false);
37
38         //将真正的执行结果返回给控制器
39         return result;
40     }
41 }

```

图 3.2: LoginActionHandler 代码截图

图 3.3: LogWriter 代码截图

在构造函数中初始化 log.xml 的文档对象，以及需要操作的 action 标签。在 log 方法中执行写入固定格式的日志信息到 log.xml 文件中。

在 tomcat 内置浏览器测试的时候真正读写的 log.xml 的位置并不在 src 目录下，而是在一个临时的位置：

```

/E:/DOCUMENT/J2EE/.metadata/.plugins/org.eclipse.wst.server.core/tmp0/wtpwebapps/SimpleController/WEB-INF/classes/log.xml

```

同时，此处需要注意，当到处为 war 包的时候，Tomcat 的全目录中，不能存在空格（即 Tomcat 不能放在有空格的文件夹名路径下），否则会报错，在.../WEB-INF/classes/log.xml 处找不到文件。

在图 3.4: log.xml 截图 中，展示了经过一些测试后，日志文件中记录的信息。

分别是：返回未知的 result 类型，登录成功，登录失败，返回未知的 result 类型。在找不到 action 的情况下，并不会进入拦截器，而是直接将结果返回客户端了。

```

LoginController.java  controller.xml  LogWriter.java  *LoginActionHandler.java
1  package me.king.interceptor;
2
3  import java.io.File;
17
18 public class LogWriter {
19     private Element crtEle;
20     private Document dc;
21     private String logXml;
22
23     public LogWriter() throws DocumentException{ //构造函数, 获取log.xml文件
24         logXml = Thread.currentThread().getContextClassLoader().getResource("").getPath()+"log.xml";
25         //System.out.println(logXml); //测试使用, 输出真正的log.xml的位置
26         dc = (new SAXReader()).read( new File(logXml));
27         crtEle = dc.getRootElement().addElement("action"); //保存当前正在使用的action元素
28     }
29     public void log(String recordStr, boolean isStartTime) throws IOException{
30         SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
31
32         if(isStartTime){ //若是记录开始时间则表示需要创建新的action节点
33             crtEle.addElement("name", recordStr);
34             crtEle.addElement("s-time", sdf.format(new Date()));
35         }else{ //否则操作当前的action节点
36             crtEle.addElement("e-time", sdf.format(new Date()));
37             crtEle.addElement("result", recordStr);
38         }
39         //写入到文件中
40         OutputFormat format = OutputFormat.createPrettyPrint();
41         format.setEncoding("gbk");
42         Writer out = new FileWriter(logXml);
43         XMLWriter writer = new XMLWriter(out, format);
44         writer.write(dc);
45         writer.close();
46         System.out.println("写入logxml完成");
47     }
48 }

```

图 3.3: LogWriter 代码截图

```

log.xml
1  <?xml version="1.0" encoding="gbk"?>
2
3  <log>
4      <action>
5          <name xmlns="login"></name>
6          <s-time xmlns="2016-12-09 16:53:32"></s-time>
7          <e-time xmlns="2016-12-09 16:53:32"></e-time>
8          <result xmlns="unknown"></result>
9      </action>
10     <action>
11         <name xmlns="login"></name>
12         <s-time xmlns="2016-12-09 16:53:56"></s-time>
13         <e-time xmlns="2016-12-09 16:53:56"></e-time>
14         <result xmlns="success"></result>
15     </action>
16     <action>
17         <name xmlns="login"></name>
18         <s-time xmlns="2016-12-09 18:26:30"></s-time>
19         <e-time xmlns="2016-12-09 18:26:30"></e-time>
20         <result xmlns="fail"></result>
21     </action>
22     <action>
23         <name xmlns="login"/>
24         <s-time xmlns="2016-12-09 18:33:38"/>
25         <e-time xmlns="2016-12-09 18:33:38"/>
26         <result xmlns="unknown"/>
27     </action>
28 </log>

```

图 3.4: log.xml 截图

图 4.1: login.jsp 代码截图,

编写 2 个 form 表单, 第一个为正常登陆表单 action 设置为"login.scaction" (必须设置为后缀.scaction, 具体配置在 controller.xml 中已经设置完成), 设置用户名和密码的 name

属性分别为“name”和“pwd”。

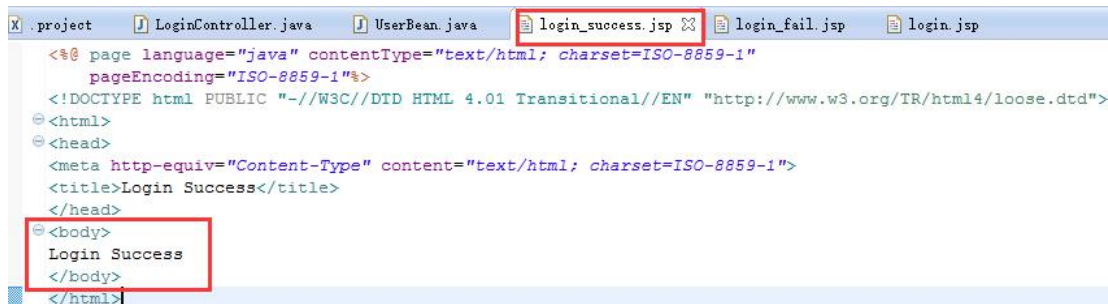
同时为了测试方便，定义一个 action 为“unknown.scaction”的表单，用于测试当 action 为未知的情况。



```
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www
4 <html>
5 <head>
6 <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
7 <title>Insert title here</title>
8 </head>
9 <body>
10 <form action="login.scaction">
11   <label>NAME:</label><input type="text" name="name"><br>
12   <label>PASSWORD:</label><input type="password" name="pwd"><br>
13   <input type="submit" value="LOGIN">
14 </form>
15 <br><br><br>
16 <form action="unknow.scaction">
17   <label>This is a unknown action request</label><br>
18   <input type="submit" value="unknown">
19 </form>
20
21 </body>
22 </html>
```

图 4.1: login.jsp 代码截图

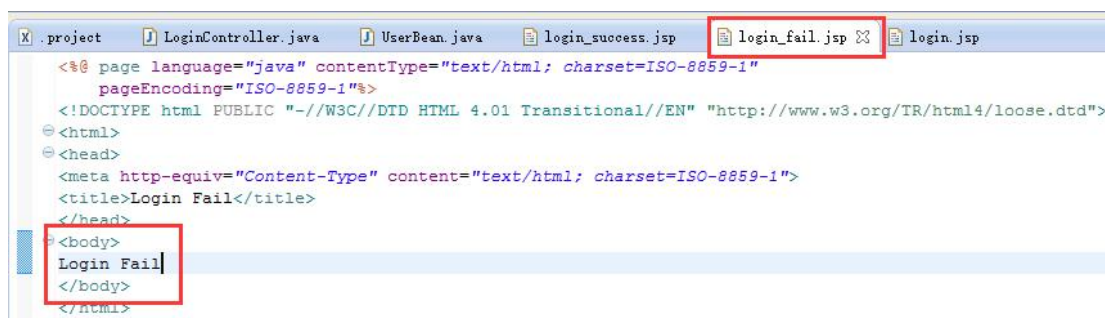
图 4.2: login_success.jsp 代码截图，
简单的在 body 中写入 Login Success。



```
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
4 <html>
5 <head>
6 <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
7 <title>Login Success</title>
8 </head>
9 <body>
10 Login Success
11 </body>
12 </html>
```

图 4.2: login_success.jsp 代码截图

图 4.3: login_fail.jsp 代码截图，
简单的在 body 中写入 Login Fail。



```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Login Fail</title>
</head>
<body>
Login Fail
</body>
</html>
```

图 4.3: login_fail.jsp 代码截图

图 4.4: error_action.jsp 代码截图

简单的在 body 中写入 `Sorry, this is a unrecognized action request.`

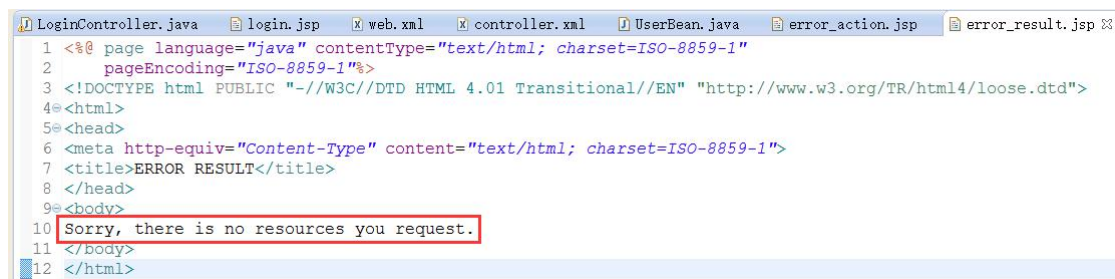


```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>ERROR ACTION</title>
</head>
<body>
Sorry, this is a unrecognized action request.
</body>
</html>
```

图 4.4: error_action.jsp 代码截图

图 4.5: error_result.jsp 代码截图

简单的在 body 中写入 `Sorry, there is no resources you request.`



```
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
4 <html>
5 <head>
6 <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
7 <title>ERROR RESULT</title>
8 </head>
9 <body>
10 Sorry, there is no resources you request.
11 </body>
12 </html>
```

图 4.5: error_result.jsp 代码截图

图 5.1: 使用 Chrome 和 Eclipse 内置浏览器测试登录页面截图, 可以看到 url 为: <http://localhost:8080/SimpleController/login.jsp>, 可以在表单中分别填入 NAME 和 PASSWORD, 点击 LOGIN 按钮提交至后台服务器。

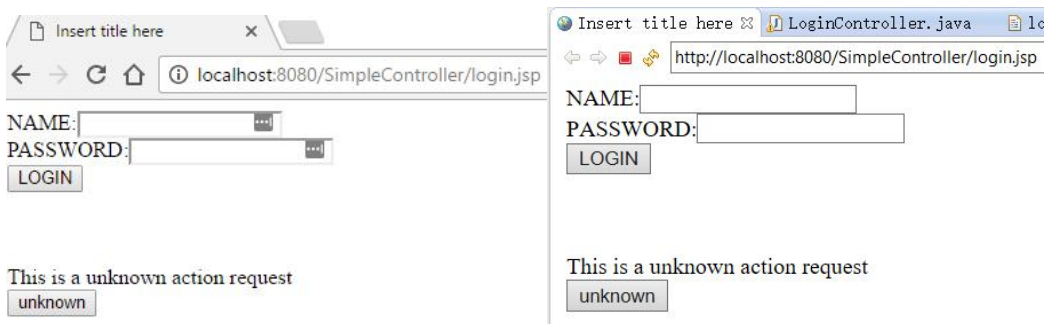


图 5.1: 使用 Chrome 和 Eclipse 内置浏览器测试登录页面截图

图 5.2: 登录成功截图，
可以通过 url 看到，当输入的用户名 name 为 world，密码 pwd 为 hello 时验证通过时显示为 success_view.xml + success_xsl.xsl 定义的视图样式。



图 5.2: 登录成功截图



图 5.2.1: 登录成功跳转页面源代码截图

图 5.3: 登录失败截图,

可以通过 url 看到, 当输入的用户名 name 或密码 pwd 错误时, 验证失败, 跳转至 login_fail.jsp 页面, 显示 Login Fail

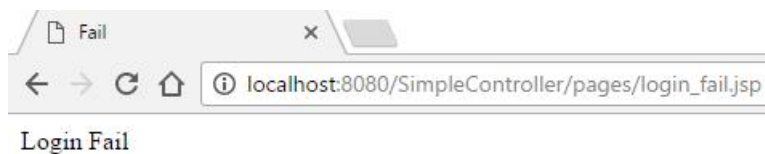


图 5.3: 登录失败截图

图 5.4: 不可识别的 action 请求截图

当点击图 5.1 中的 unknown 按钮的时候, 页面跳转至 error_action.jsp 页面, 显示 Sorry, this is a unrecognized action request.

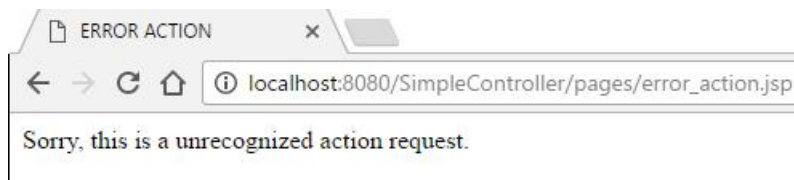


图 5.4: 不可识别的 action 请求截图

图 5.5: 没有请求的资源截图

当在登陆表单中输入的用户名 name 为 unknown, 密码 pwd 为 result 时, 跳转至 error_result.jsp 页面。显示 Sorry, there is no resources you request.



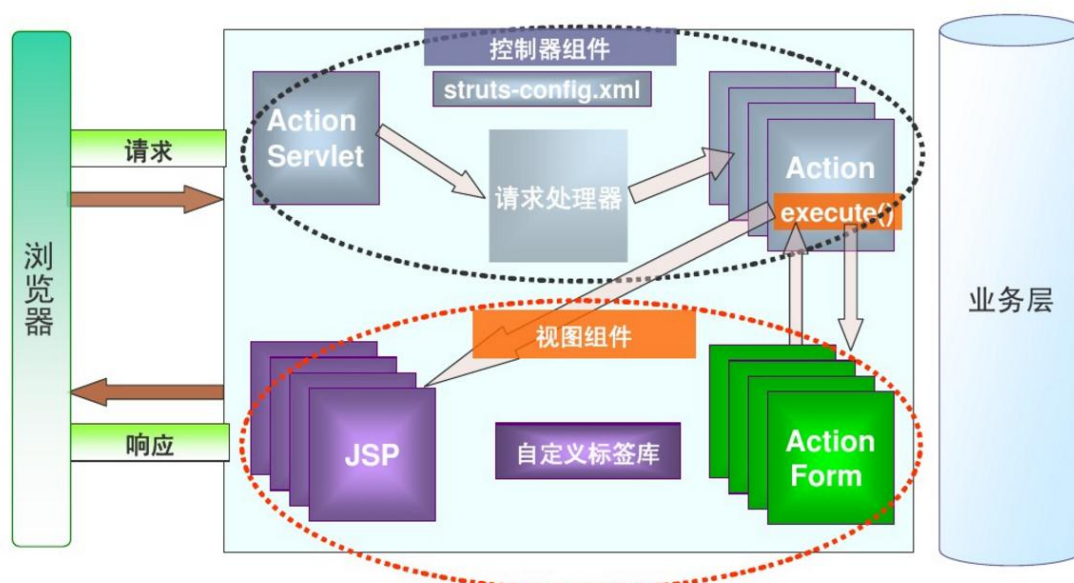
图 5.5: 没有请求的资源截图

4. 结论

对主题的总结，结果评论，发现的问题，或你的建议和看法。如 MVC 中 Controller 优点与缺点，个人看法（文字、图标、代码辅助等）

请描述 Struts 框架中视图组件的工作方式，如<s:form>/<s:submit>

Struts 组件图



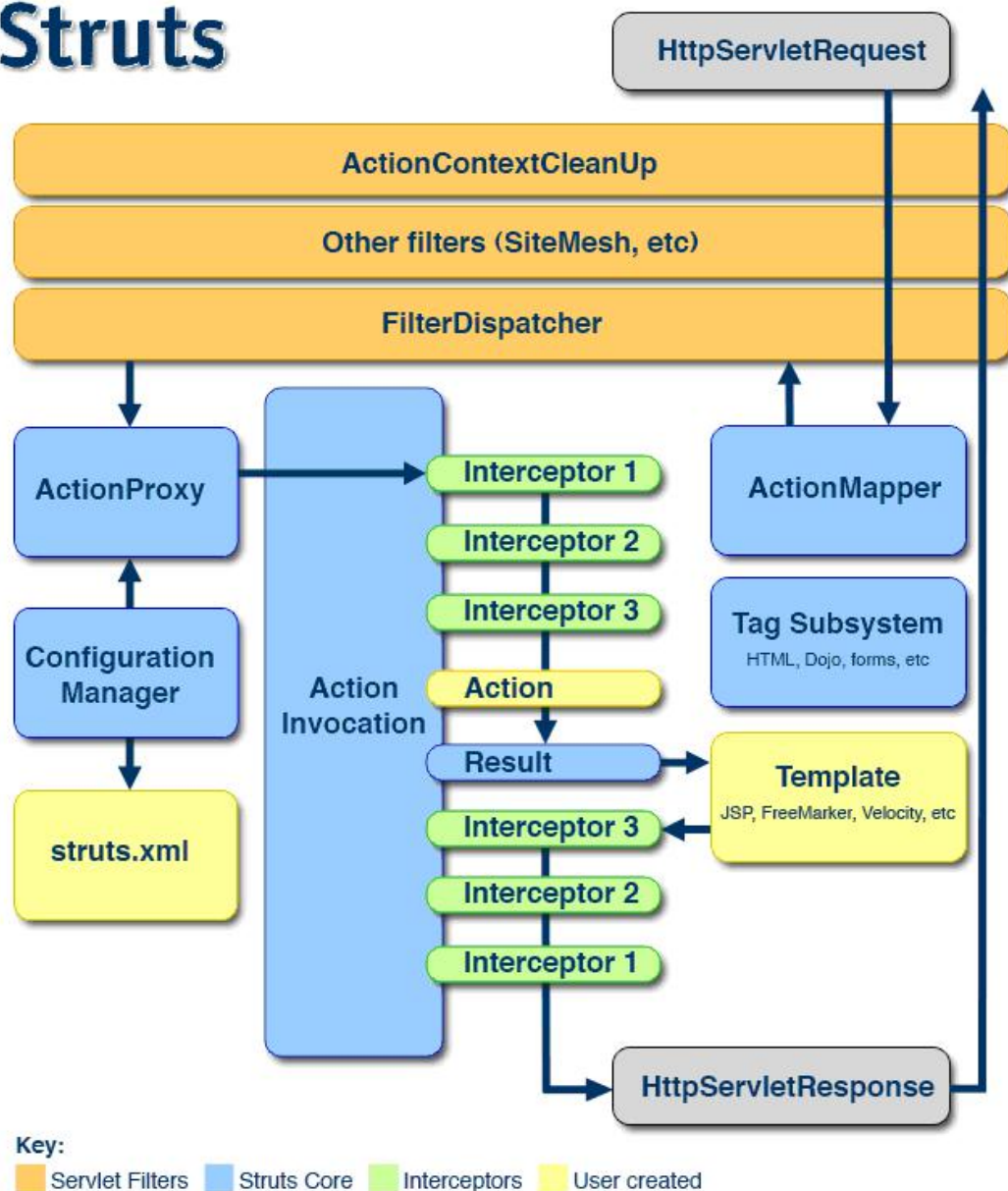
由上图，可知，Struts 视图组件的主要职责为接收控制器组件的数据请求，显示数据模型同时将用户输入的数据或请求传给控制器组件。

Struts 视图组件分为三类，分别为自定义标签库，JSP，以及 ActionForm。

Struts 1 只能支持 JSP 作为视图资源，虽然也可以使用自定义的标签库，但是配置比较复杂，学习成本太高，而且已经算是过时的技术了。

而 Struts2 中的视图组件已经不仅仅是这三类了，Struts 2 的进步之处就是可以使用其他视图技术，如 FreeMarker、Velocity 等。具体看下图的 Struts2 体系结构图。

Struts



一旦 Action 执行完毕，ActionInvocation 负责根据 struts.xml 中的配置找到对应的返回结果。返回结果通常是（但不总是，也可能是另外的一个 Action 链）一个需要被表示的 JSP 或者 FreeMarker 的模版。在表示的过程中可以使用 Struts2 框架中继承的标签。在这个过程中需要涉及到 ActionMapper。

在 struts.xml 配置文件中，每一个 Action 定义都有 name 和 class 属性，同时还要指定 result 元素。result 元素指定了逻辑视图名称和实际视图的对应关系。每个 result 都有一个 type 属性。

关于标签：

Struts2 中将所有的标签都定义在一个 s 标签库里，比如：<s:form>/<s:submit>，使用标签需要先导入标签库，代码为：<%@taglib prefix="s" uri="/struts-tags"%> 其中 uri 就是 struts2 标签库的 URL，而 prefix 属性值是该标签库的前缀。这些标签或用于生成 html 元素标签，或用于数据访问，控制逻辑。在数据访问上，可以使用 OGNL 表达式语言。

参考文献

以上内容的理论知识点或技术点如果参考了网上或印刷制品，请在这里罗列出来

[1] java 中通过 xsl 将 xml 数据转换为 html 格式字符串：

http://blog.csdn.net/hu_shengyang/article/details/8679036

[2] Java 将 XML 和 XSL 转换成 HTML：<http://lanqiaoyeyu.iteye.com/blog/1169623>

[3] 4.Struts 视图组件：

http://wenku.baidu.com/link?url=g9tcVVnEKtpva4_vFyUR_ukfGf1B93zmho93EzCtYiRjezFEQaa8tDbi1CzFjlh0QB_a2zkTvrKDL061nXkqvLA6kDWc5zPKq8qHk4mRucS