作者博客：http://jinnianshilongnian.iteye.com/ 欢迎访问

第十章　集成其它 Web 框架

10.1　概述

10.1.1　Spring 和 Web 框架

Spring 框架不仅提供了一套自己的 Web 框架实现，还支持集成第三方 Web 框架（如 Struts1x、Struts2x）。

Spring 实现的 SpringMVC Web 框架将在第十八章详细介绍。

由于现在有很大部分公司在使用第三方 Web 框架，对于并不熟悉 SpringMVC Web 框架的公司，为了充分利用开发人员已掌握的技术并相使用 Spring 的功能，想集成所使用的 Web 框架；由于 Spring 框架的高度可配置和可选择性，因此集成这些第三方 Web 框架是非常简单的。

之所以想把这些第三方 Web 框架集成到 Spring 中，最核心的价值是享受 Spring 的某些强大功能，如一致的数据访问，事务管理，IOC，AOP 等等。

Spring 为所有 Web 框架提供一致的通用配置，从而不管使用什么 Web 框架都使用该通用配置。

10.1.2 通用配置

Spring 对所有 Web 框架抽象出通用配置，以减少重复配置，其中主要有以下配置：

1、Web 环境准备：

1.1、在 spring 项目下创建如图 10-1 目录结构：

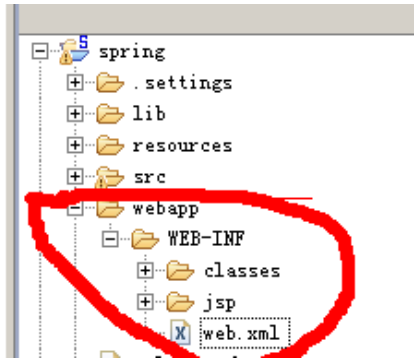


图 10-1 web 目录结构

1.2、右击 spring 项目选择【Propeties】，然后选择【Java Build Path】中的【Source】选项卡，将类输出路径修改为“spring/webapp/WEB-INF/classes”，如图 10-2 所示：

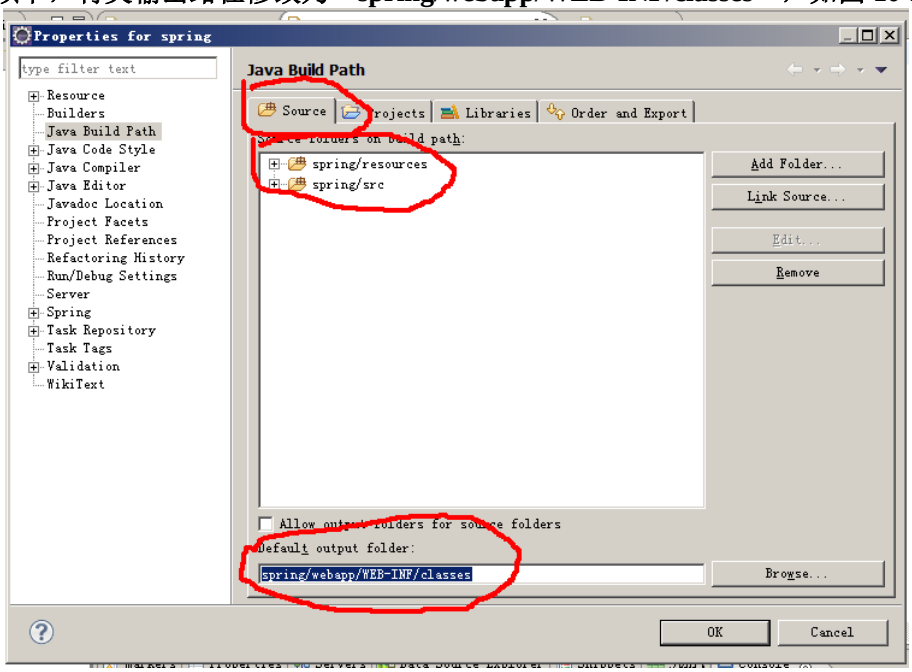


图 10-2 修改类输出路径

1.3、web.xml 初始内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
</web-app>
```

<web-app version="2.4">表示采用 Servlet 2.4 规范的 Web 程序部署描述格式

2、 指定 Web 应用上下文实现：在 Web 环境中，Spring 提供 WebApplicationContext（继承 ApplicationContext）接口用于配置 Web 应用，该接口应该被实现为在 Web 应用程序运行时只读，即在初始化完毕后不能修改 Spring Web 容器（WebApplicationContext），但可能支持重载。

Spring 提供 XmlWebApplicationContext 实现，并在 Web 应用程序中默认使用该实现，可以通过在 web.xml 配置文件中使用如下方式指定：

```
<context-param>
    <param-name>contextClass</param-name>
    <param-value>
        org.springframework.web.context.support.XmlWebApplicationContext
    </param-value>
</context-param>
```

如上指定是可选的，只有当使用其他实现时才需要显示指定。

3、 指定加载文件位置：

前边已经指定了 Spring Web 容器实现，那从什么地方加载配置文件呢？

默认情况下将加载/WEB-INF/applicationContext.xml 配置文件，当然也可以使用如下形式在 web.xml 中定义要加载自定义的配置文件，多个配置文件用“，”分割：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:chapter10/applicationContext-message.xml
    </param-value>
</context-param>
```

通用 Spring 配置文件（resources/chapter10/applicationContext-message.xml）内容如下所示：

```
<bean id="message" class="java.lang.String">
    <constructor-arg index="0" value="Hello Spring"/>
</bean>
```

4、 加载和关闭 Spring Web 容器：

我们已经指定了 Spring Web 容器实现和配置文件，那如何才能让 Spring 使用相应的 Spring Web 容器实现加载配置文件呢？

Spring 使用 ContextLoaderListener 监听器来加载和关闭 Spring Web 容器，即使用如下方式在 web.xml 中指定：

```
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

ContextLoaderListener 监听器将在 Web 应用启动时使用指定的配置文件初始化 Spring Web 容器，在 Web 应用关闭时销毁 Spring Web 容器。

注：监听器是从 Servlet 2.3 才开始支持的，因此如果 Web 应用所运行的环境是 Servlet 2.2 版本则可以使用 ContextLoaderServlet 来完成，但从 Spring3.x 版本之后 ContextLoaderServlet 被移除了。

5、 在 Web 环境中获取 Spring Web 容器:

既然已经定义了 Spring Web 容器，那如何在 Web 中访问呢？Spring 提供如下方式来支持获取 Spring Web 容器（WebApplicationContext）：

```
WebApplicationContextUtils.getWebApplicationContext(servletContext);
或
WebApplicationContextUtils.getRequiredWebApplicationContext(servletContext);
```

如果当前 Web 应用中的 ServletContext 中没有相应的 Spring Web 容器，对于 getWebApplicationContext()方法将返回 null,而 getRequiredWebApplicationContext()方法将抛出异常，建议使用第二种方式，因为缺失 Spring Web 容器而又想获取它，很明显是错误的，应该抛出异常。

6、 通用 jar 包，从下载的 spring-framework-3.0.5.RELEASE-with-docs.zip 中 dist 目录查找如下 jar 包：

```
org.springframework.web-3.0.5.RELEASE.jar
```

此 jar 包为所有 Web 框架所共有，提供 WebApplicationContext 及实现等。

7、Web 服务器选择及测试：

目前比较流行的支持 Servlet 规范的开源 Web 服务器包括 Tomcat、Resin、Jetty 等，Web 服务器有独立运行和嵌入式运行之分，嵌入式 Web 服务器可以在测试用例中运行不依赖于外部环境，因此我们使用嵌入式 Web 服务器。

Jetty 是一个非常轻量级的 Web 服务器，并且提供嵌入式运行支持，在此我们选用 Jetty 作为测试使用的 Web 服务器。

7.1、准备 Jetty 嵌入式 Web 服务器运行需要的 jar 包：

到 http://dist.codehaus.org/jetty/网站下载 jetty-6.1.24，在下载的 jetty-6.1.24.zip 包中拷贝如下 jar 包到项目的 lib/jetty 目录下，并添加到类路径中：

```
lib\jetty-6.1.24.jar                  //核心 jetty 包
lib\jetty-util-6.1.24.jar             //工具 jetty 包
lib\jsp-2.1\ant-1.6.5.jar
lib\jsp-2.1\core-3.1.1.jar
lib\jsp-2.1\jsp-2.1-glassfish-2.1.v20091210.jar     //jsp2.1 支持相关 jar 包
lib\jsp-2.1\jsp-2.1-jetty-6.1.24.jar
lib\jsp-2.1\jsp-api-2.1-glassfish-2.1.v20091210.jar
```

7.2、在单元测试中启动 Web 服务器：

```
package cn.javass.spring.chapter10;
import org.junit.Test;
import org.mortbay.jetty.Server;
import org.mortbay.jetty.webapp.WebAppContext;
public class WebFrameWorkIntegrateTest {
    @Test
    public void testWebFrameWork() throws Exception {
```

```
        Server server = new Server(8080);
        WebAppContext webapp = new WebAppContext();
        webapp.setResourceBase("webapp");
        //webapp.setDescriptor("webapp/WEB-INF/web.xml");
        webapp.setContextPath("/");
        webapp.setClassLoader(Thread.currentThread().getContextClassLoader());
        server.setHandler(webapp);
        server.start();
        server.join();
        //server.stop();
    }
}
```

- **创建内嵌式 Web 服务器：**使用 new Server(8080)新建一个 Jetty 服务器，监听端口为 8080；
- **创建一个 Web 应用：**使用 new WebAppContext()新建一个 Web 应用对象，一个 Web 应用可以认为就是一个 WebAppContext 对象；
- **指定 Web 应用的目录：**使用 webapp.setResourceBase("webapp")指定 Web 应用位于项目根目录下的“webapp”目录下；
- **指定部署描述符：**使用 webapp.setDescriptor("webapp/WEB-INF/web.xml")；此处指定部署描述符为项目根目录下的“webapp/WEB-INF/web.xml”，该步骤是可选的，如果 web.xml 位于 Web 应用的 WEB-INF 下。
- **指定 Web 应用请求上下文：**使用 webapp.setContextPath("/")指定请求上下文为“/”，从而访问该 Web 应用可以使用如“http://localhost:8080/hello.do”形式访问；
- 指定类装载器：因为 Jetty 自带的 ClassLoader 在内嵌环境中对中文路径处理有问题，因此我们使用 Eclispe 的 ClassLoader，即通过“webapp.setClassLoader(Thread.currentThread().getContextClassLoader()) ”指定；
- **启动 Web 服务器：**使用“server.start()”启动并使用“server.join()”保证 Web 服务器一直运行；
- **关闭 Web 服务器：**可以通过某种方式执行“server.stop()”来关闭 Web 服务器；另一种方式是通过【Console】控制台面板的【Terminate】终止按钮关闭，如图 10-3 所示：

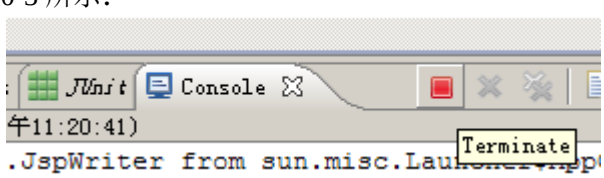


图 10-3 点击红色按钮关闭 Web 服务器

10.2 集成 Struts1.x

10.2.1 概述

Struts1.x 是最早实现 MVC（模型-视图-控制器）模式的 Web 框架之一，其使用非常广泛，虽然目前已经有 Struts2.x 等其他 Web 框架，但仍有很多公司使用 Struts1.x 框架。

集成 Struts1.x 也非常简单，除了通用配置外，有两种方式可以将 Struts1.x 集成到 Spring 中：

- 最简单集成：使用 Spring 提供的 WebApplicationContextUtils 工具类中的获取 Spring Web 容器，然后通过 Spring Web 容器获取 Spring 管理的 Bean；
- Struts1.x 插件集成：利用 Struts1.x 中的插件 ContextLoaderPlugin 来将 Struts1.x 集成到 Spring 中。

接下来让我们首先让我们准备 Struts1x 所需要的 jar 包：

1.1、从下载的 spring-framework-3.0.5.RELEASE-with-docs.zip 中 dist 目录查找如下 jar 包，该 jar 包用于提供集成 struts1.x 所需要的插件实现等：

org.springframework.web.struts-3.0.5.RELEASE.jar

1.2、从下载的 spring-framework-3.0.5.RELEASE-dependencies.zip 中查找如下依赖 jar 包，该组 jar 是 struts1.x 需要的 jar 包：

com.springsource.org.apache.struts-1.2.9.jar	//struts1.2.9 实现包
com.springsource.org.apache.commons.digester-1.8.1.jar	//用于解析 struts 配置文件
com.springsource.org.apache.commons.beanutils-1.8.0.jar	//用于请求参数绑定
com.springsource.javax.servlet-2.5.0.jar	//Servlet 2.5 API
antlr.jar	//语法分析包（已有）
commons-logging.jar	//日志记录组件包（已有）
servlet-api.jar	//Servlet API 包（已有）
jsp-api.jar	//JSP API 包（已有，可选）
commons-validator.jar	**//验证包（可选）**
commons-fileupload.jar	**//文件上传包（可选）**

10.2.2 最简单集成

只使用通用配置，利用 WebApplicationContextUtils 提供的获取 Spring Web 容器方法获

取 Spring Web 容器，然后从 Spring Web 容器获取 Spring 管理的 Bean。

1、 第一个 Action 实现：

```
package cn.javass.spring.chapter10.struts1x.action;
import org.apache.struts.action.Action;
//省略部分 import
public class HelloWorldAction1 extends Action {
    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response) throws Exception {

        WebApplicationContext ctx = WebApplicationContextUtils.
            getRequiredWebApplicationContext(getServlet().getServletContext());
        String message = ctx.getBean("message", String.class);
        request.setAttribute("message",    message);
        return mapping.findForward("hello");
    }
}
```

此 Action 实现非常简单，首先通过 WebApplicationContextUtils 获取 Spring Web 容器，然后从 Spring Web 容器中获取“message”Bean 并将其放到 request 里，最后转到“hello”所代表的 jsp 页面。

2、JSP 页面定义（webapp/WEB-INF/jsp/hello.jsp）：

```
<%@ page language="java" pageEncoding="UTF-8"
    contentType="text/html; charset=UTF-8" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
    <title>Hello World</title>
</head>
<body>
    ${message}
</body>
</html>
```

3、配置文件定义：

3.1、Spring 配置文件定义（resources/chapter10/applicationContext-message.xml）：

在此配置文件中定义我们使用的“message”Bean；

```
<bean id="message" class="java.lang.String">
    <constructor-arg index="0" value="Hello Spring"/>
</bean>
```

3.2、struts 配置文件定义（resources/chapter10/struts1x/struts-config.xml）：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
          "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
<struts-config>
  <action-mappings>
    <action path="/hello"
        type="cn.javass.spring.chapter10.struts1x.action.HelloWorldAction1">
      <forward name="hello" path="/WEB-INF/jsp/hello.jsp"/>
    </action>
  </action-mappings>
</struts-config>
```

3.3、web.xml 部署描述符文件定义（webapp/WEB-INF/web.xml）添加如下内容：

```
    <!-- Struts1.x前端控制器配置开始     -->
    <servlet>
        <servlet-name>hello</servlet-name>
        <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>

        <init-param>
            <param-name>config</param-name>
            <param-value>
                /WEB-INF/classes/chapter10/struts1x/struts-config.xml
            </param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>hello</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
    <!-- Struts1.x前端控制器配置结束     -->
```

Struts1.x 前端控制器配置了 ActionServlet 前端控制器，其拦截以.do 开头的请求，Strut 配置文件通过初始化参数“config”来指定，如果不知道“config”参数则默认加载的配置文件为“/WEB-INF/ struts-config.xml”。

4、执行测试：在 Web 浏览器中输入 http://localhost:8080/hello.do 可以看到“Hello Spring”信息说明测试正常。

有朋友想问，我不想使用这种方式，我想在独立环境内测试，没关系，您只需将 spring/lib 目录拷贝到 spring/webapp/WEB-INF/下，然后将 webapp 拷贝到如 tomcat 中即可运行，尝试一下吧。

Spring 还提供 ActionSupport 类来简化获取 WebApplicationContext，Spring 为所有标准 Action 类及子类提供如下支持类，即在相应 Action 类后边加上 Support 后缀：

- ActionSupport
- DispatchActionSupport
- LookupDispatchActionSupport
- MappingDispatchActionSupport

具体使用方式如下：

1、Action 定义

```
package cn.javass.spring.chapter10.struts1x.action;
//省略 import
public class HelloWorldAction2 extends ActionSupport {
    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response) throws Exception {
        WebApplicationContext ctx = getWebApplicationContext();
        String message = ctx.getBean("message", String.class);
        request.setAttribute("message", message);
        return mapping.findForward("hello");
    }
}
```

和第一个示例唯一不同的是直接调用 **getWebApplicationContext()**即可获得 Spring Web 容器。

2、修改 Struts 配置文件（resources/chapter10/struts1x/struts-config.xml）添加如下 Action 定义：

```
<action path="/hello2"
        type="cn.javass.spring.chapter10.struts1x.action.HelloWorldAction2">
    <forward name="hello" path="/WEB-INF/jsp/hello.jsp"/>
</action>
```

3、启动嵌入式 Web 服务器并在 Web 浏览器中输入 http://localhost:8080/hello2.do 可以看到“Hello Spring”信息说明 Struts1 集成成功。

这种集成方式好吗？而且这种方式算是集成吗？直接获取 Spring Web 容器然后从该 Spring Web 容器中获取 Bean，暂且看作是集成吧，这种集成对于简单操作可以接受，但更

复杂的注入呢？接下来让我们学习使用 Struts 插件进行集成。

10.2.2 Struts1.x 插件集成

Struts 插件集成使用 ContextLoaderPlugin 类，该类用于为 ActionServlet 加载 Spring 配置文件。

1、在 Struts 配置文件（resources/chapter10/struts1x/struts-config.xml）中配置插件：

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
  <set-property property="contextClass"
       value="org.springframework.web.context.support.XmlWebApplicationContext"/>
  <set-property property="contextConfigLocation"
       value="/WEB-INF/hello-servlet.xml"/>
  <set-property property="namespace" value="hello"/>
</plug-in>
```

- **contextClass：**可选，用于指定 WebApplicationContext 实现类，默认是 XmlWebApplicationContext；
- **contextConfigLocation：**指定 Spring 配置文件位置，如果我们的 ActionServlet 在 web.xml 里面通过 <servlet-name>hello</servlet-name>指定名字为“hello”，且没有指定 contextConfigLocation，则默认 Spring 配置文件是 /WEB-INF/hello-servlet.xml；
- **namespace：**因为默认使用 ActionServlet 在 web.xml 定义中的 Servlet 的名字，因此如果想要使用其他名字可以使用该变量指定，如指定“hello”，将加载的 Spring 配置文件为/WEB-INF/hello-servlet.xml；

由于我们的 ActionServlet 在 web.xml 中的名字为 hello，而我们的配置文件在 /WEB-INF/hello-servlet.xml，因此 contextConfigLocation 和 namespace 可以不指定，因此最简单配置如下：

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn"/>
```

通用配置的 Spring Web 容器将作为 ContextLoaderPlugin 中创建的 Spring Web 容器的父容器存在，然而可以省略通用配置而直接在 struts 配置文件中通过 ContextLoaderPlugin 插件指定所有配置文件。

插件已经配置了，那如何定义 Action、配置 Action、配置 Spring 管理 Bean 呢，即如何真正集成 Spring+Struts1x 呢？使用插件方式时 Action 将在 Spring 中配置而不是在 Struts 中配置了，Spring 目前提供以下两种方式：

- 将 Struts 配置文件中的<action>的 type 属性指定为 DelegatingActionProxy，然后在 Spring 中配置同名的 Spring 管理的 Action Bean；

- 使用 Spring 提供的 DelegatingRequestProcessor 重载 Struts 默认的 RequestProcessor 来从 Spring 容器中查找同名的 Spring 管理的 Action Bean。

看懂了吗？好像没怎么看懂，那就直接上代码，有代码有真相。

2、定义 Action 实现，由于 Action 将在 Spring 中配置，因此 message 可以使用依赖注入方式了：

```
package cn.javass.spring.chapter10.struts1x.action;
//省略
public class HelloWorldAction3 extends Action {
    private String message;
    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response) throws Exception {

        request.setAttribute("message", message);
        return mapping.findForward("hello");
    }
    public void setMessage(String message) {//有 setter 方法，大家是否想到 setter 注入
        this.message = message;
    }
}
```

3、DelegatingActionProxy 方式与 Spring 集成配置：

3.1、在 Struts 配置文件（resources/chapter10/struts1x/struts-config.xml）中进行 Action 定义：

```
<action path="/hello3"
        type="org.springframework.web.struts.DelegatingActionProxy">
    <forward name="hello" path="/WEB-INF/jsp/hello.jsp"/>
</action>
```

3.2、在 Spring 配置文件（webapp/WEB-INF/hello-servlet.xml）中定义 Action 对应的 Bean：

```
<bean name="/hello3"
        class="cn.javass.spring.chapter10.struts1x.action.HelloWorldAction3">
    <property name="message" ref="message"/>
</bean>
```

3.3、启动嵌入式 Web 服务器并在 Web 浏览器中输入 http://localhost:8080/hello3.do 可

以看到“Hello Spring”信息说明测试正常。

从以上配置中可以看出：

- Struts 配置文件中<action>标签的 path 属性和 Spring 配置文件的 name 属性应该完全一样，否则错误；
- Struts 通过 **DelegatingActionProxy** 去到 Spring Web 容器中查找同名的 Action Bean；

很简单吧，DelegatingActionProxy 是个代理 Action，其实现了 Action 类，其内部帮我们查找相应的 Spring 管理 Action Bean 并把请求转发给这个真实的 Action。

4、DelegatingRequestProcessor 方式与 Spring 集成：

4.1、首先要替换掉 Struts 默认的 RequestProcessor，在 Struts 配置文件（resources/chapter10/struts1x/struts-config.xml）中添加如下配置：

```
<controller>
    <set-property property="processorClass"
        value="org.springframework.web.struts.DelegatingRequestProcessor"/>
</controller>
```

4.2、在 Struts 配置文件（resources/chapter10/struts1x/struts-config.xml）中进行 Action 定义：

```
<action path="/hello4"
        type=" cn.javass.spring.chapter10.struts1x.action.HelloWorldAction3">
    <forward name="hello" path="/WEB-INF/jsp/hello.jsp"/>
</action>
```

或更简单形式：

```
<action path="/hello4">
    <forward name="hello" path="/WEB-INF/jsp/hello.jsp"/>
</action>
```

4.3、在 Spring 配置文件（webapp/WEB-INF/hello-servlet.xml）中定义 Action 对应的 Bean：

```
<bean name="/hello4"
        class="cn.javass.spring.chapter10.struts1x.action.HelloWorldAction3">
    <property name="message" ref="message"/>
</bean>
```

4.4、启动嵌入式 Web 服务器并在 Web 浏览器中输入 http://localhost:8080/hello4.do 可以看到“Hello Spring”信息说明 Struts1 集成成功。

从以上配置中可以看出：

- Struts 配置文件中<action>标签的 path 属性和 Spring 配置文件的 name 属性应该完全一样，否则错误；
- Struts 通过 **DelegatingRequestProcessor** 去到 Spring Web 容器中查找同名的 Action Bean；

很简单吧，只是由 **DelegatingRequestProcessor** 去帮我们查找相应的 Action Bean，但没有代理 Action 了，所以推荐使用该方式。

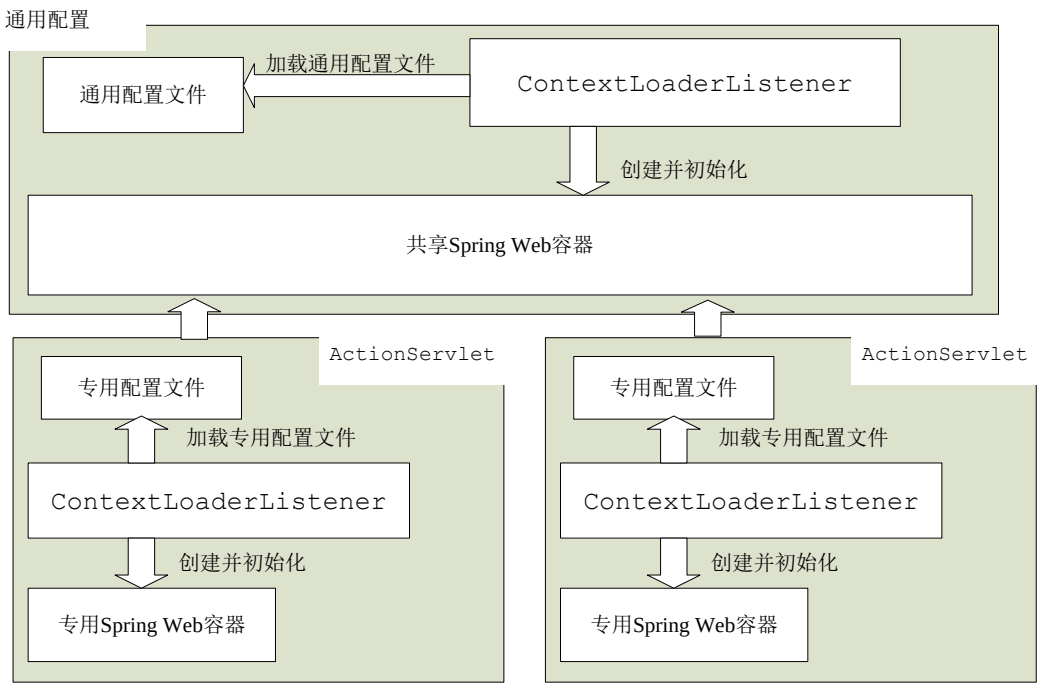


图 10-4 共享及专用 Spring Web 容器

Struts1x 与 Spring 集成到此就完成了，在集成时需要注意一下几点：

- 推荐使用 ContextLoaderPlugin+DelegatingRequestProcessor 方式集成；
- 当有多个 Struts 模块时建议在通用配置部分配置通用部分，因为通用配置在正在 Web 容器中是可共享的，而在各个 Struts 模块配置文件中配置是不可共享的，因此不推荐直接使用 ContextLoaderPlugin 中为每个模块都指定所有配置，因为 **ContextLoaderPlugin 加载的 Spring 容器只对当前的 ActionServlet 有效对其他 ActionServlet 无效，**如图 10-4 所示。

、10.3 集成 Struts2.x

10.3.1 概述

Struts2 前身是 WebWork，核心并没有改变，其实就是把 WebWork 改名为 struts2，与 Struts1 一点关系没有。

Struts2 中通过 ObjectFactory 接口实现创建及获取 Action 实例，类似于 Spring 的 IoC 容器，所以 Action 实例可以由 ObjectFactory 实现来管理，因此集成 Spring 的关键点就是如何创建 ObjectFactory 实现来从 Spring 容器中获取相应的 Action Bean。

Struts2 提供一个默认的 ObjectFactory 接口实现 StrutsSpringObjectFactory，该类用于根据 Struts2 配置文件中相应 Bean 信息从 Spring 容器中获取相应的 Action。

因此 Struts2.x 与 Spring 集成需要使用 StrutsSpringObjectFactory 类作为中介者。

接下来让我们首先让我们准备 Struts2x 所需要的 jar 包

准备 Struts2.x 需要的 jar 包，到 Struts 官网 http://struts.apache.org/下载 struts-2.2.1.1 版本，拷贝如下 jar 包到项目的 lib 目录下并添加到类路径：

```
lib\struts2-core-2.2.1.1.jar                 //核心 struts2 包
lib\xwork-core-2.2.1.1.jar                   //命令框架包，独立于 Web 环境，为 Struts2
                                             //提供核心功能的支持包
lib\freemarker-2.3.16.jar                    //提供模板化 UI 标签及视图技术支持
lib\ognl-3.0.jar                             //对象图导航工具包，类似于 SpEL
lib\ struts2-spring-plugin-2.2.1.1.jar       //集成 Spring 的插件包
lib\commons-logging-1.0.4.jar                //日志记录组件包（已有）
lib\commons-fileupload-1.2.1.jar             //用于支持文件上传的包
```

10.3.2 使用 ObjectFactory 集成

1、Struts2.x 的 Action 实现：

```
package cn.javass.spring.chapter10.struts2x.action;
import org.apache.struts2.ServletActionContext;
import com.opensymphony.xwork2.ActionSupport;
public class HelloWorldAction extends ActionSupport {
    private String message;
    @Override
    public String execute() throws Exception {
        ServletActionContext.getRequest().setAttribute("message", message);
        return "hello";
    }
    public void setMessage(String message) {//setter 注入
        this.message = message;
    }
}
```

2、JSP 页面定义，使用 Struts1x 中定义的 JSP 页面“webapp/WEB-INF/jsp/hello.jsp”；

3、Spring 一般配置文件定义（resources/chapter10/applicationContext-message.xml）：

在此配置文件中定义我们使用的“message”Bean；

```
<bean id="message" class="java.lang.String">
    <constructor-arg index="0" value="Hello Spring"/>
</bean>
```

4、Spring Action 配置文件定义（resources/chapter10/hello-servlet.xml）：

```
<bean name="helloAction"
      class="cn.javass.spring.chapter10.struts2x.action.HelloWorldAction"
      scope="prototype">
    <property name="message" ref="message"/>
</bean>
```

Struts2 的 Action 在 Spring 中配置，而且应该是 prototype，因为 Struts2 的 Action 是有状态的，定义在 Spring 中，那 Struts 如何找到该 Action 呢？

5、struts2 配置文件定义（resources/chapter10/struts2x/struts.xml）：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <constant name="struts.objectFactory"
              value="org.apache.struts2.spring.StrutsSpringObjectFactory"/>
    <constant name="struts.devMode" value="true"/>
    <package name="default" extends="struts-default">
        <action name="hello" class="helloAction">
            <result name="hello" >/WEB-INF/jsp/hello.jsp</result>
        </action>
    </package>
</struts>
```

- **struts.objectFactory**：通过在 Struts 配置文件中使用常量属性 **struts.objectFactory** 来定义 Struts 将要使用的 ObjectFactory 实现，此处因为需要从 Spring 容器中获取 Action 对象，因此需要使用 StrutsSpringObjectFactory 来集成 Spring；
- <action name="hello" **class="helloAction"**>：StrutsSpringObjectFactory 对象工厂将根据<action>标签的 class 属性去 Spring 容器中查找同名的 Action Bean；即本例中将到 Spring 容器中查找名为 helloAction 的 Bean。

6、web.xml 部署描述符文件定义（webapp/WEB-INF/web.xml）：

6.1、由于 Struts2 只能使用通用配置，因此需要在通用配置中加入 Spring Action 配置文件（chapter10/struts2x/struts2x-servlet.xml）：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:chapter10/applicationContext-message.xml,
        classpath:chapter10/struts2x/struts2x-servlet.xml
    </param-value>
</context-param>
```

Struts2 只能在通用配置中指定所有 Spring 配置文件，并没有如 Struts1 自己指定 Spring 配置文件的实现。

6.2、Strut2 前端控制器定义，在 web.xml 中添加如下配置：

```
<!-- Struts2.x前端控制器配置开始     -->
<filter>
    <filter-name>struts2x</filter-name>
    <filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
    <init-param>
        <param-name>config</param-name>
        <param-value>
            struts-default.xml,struts-plugin.xml,chapter10/struts2x/struts.xml
        </param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>struts2x</filter-name>
    <url-pattern>*.action</url-pattern>
</filter-mapping>
<!-- Struts2.x 前端控制器配置结束     -->
```

- **FilterDispatcher**：Struts2 前端控制器为 FilterDispatcher，是 Filter 实现，不是

Servlet；

- **config**：通过初始化参数 config 指定配置文件为 struts-default.xml，struts-plugin.xml，chapter10/struts2x/struts.xml；如果不知道该参数则默认加载 struts-default.xml，struts-plugin.xml，struts.xml(位于 webapp/WEB-INF/classes 下)；显示指定时需要将 struts-default.xml，struts-plugin.xml 也添加上。
- ***.action**：将拦截以“.action”结尾的 HTTP 请求；
- **struts2x**：FilterDispatcher 前端控制器的名字为 struts2x，因此相应的 Spring 配置文件名为 struts2x-servlet.xml。

7、执行测试，在 Web 浏览器中输入 http://localhost:8080/hello.action 可以看到“Hello Spring”信息说明 Struts2 集成成功。

集成 Strut2 也是非常简单，在此我们总结一下吧：

- **配置文件位置：**
  - Struts 配置文件默认加载“struts-default.xml,struts-plugin.xml, struts.xml”，其中 struts-default.xml 和 struts-plugin.xml 是 Struts 自带的，而 struts.xml 是我们指定的，默认位于 webapp/WEB-INF/classes 下；
  - 如果需要将配置文件放到其他位置，需要在 web.xml 的<filter>标签下，使用初始化参数 config 指定，如“struts-default.xml，struts-plugin.xml，chapter10/struts2x/struts.xml”，其中“struts-default.xml 和 struts-plugin.xml”是不可省略的，默认相对路径是类路径。
- **集成关键 ObjectFactory**：在 Struts 配置文件或属性文件中使用如下配置知道使用 StrutsSpringObjectFactory 来获取 Action 实例：

在 struts.xml 中指定：

```
<constant name="struts.objectFactory"
          value="org.apache.struts2.spring.StrutsSpringObjectFactory"/>
```

或在 struts.properties 文件（webapp/WEB-INF/classes/）中：

```
struts.objectFactory=org.apache.struts2.spring.StrutsSpringObjectFactory
```

- **集成关键 Action 定义：**

StrutsSpringObjectFactory 将根据 Struts2 配置文件中的<action class="">标签的 classes 属性名字去到 Spring 配置文件中查找同名的 Bean 定义，这也是集成的关键。

```
<action name="hello" class="helloAction">
    ……
</action>                          通过同名来集成
```

```
<bean name="helloAction" class="Action类全限定名" scope="prototype">
    ……
</bean>
```

- **Spring 配置文件中 Action 定义：**由于 Struts2 的 Action 是有状态的，因此应该将 Bean 定义为 prototype。

如图 10-5，Sturt2 与 Spring 集成的关键就是 StrutsSpringObjectFactory，注意图只是说明 Struts 与 Spring 如何通过中介者 StrutsSpringObjectFactory 来实现集成，不能代表实际的类交互。

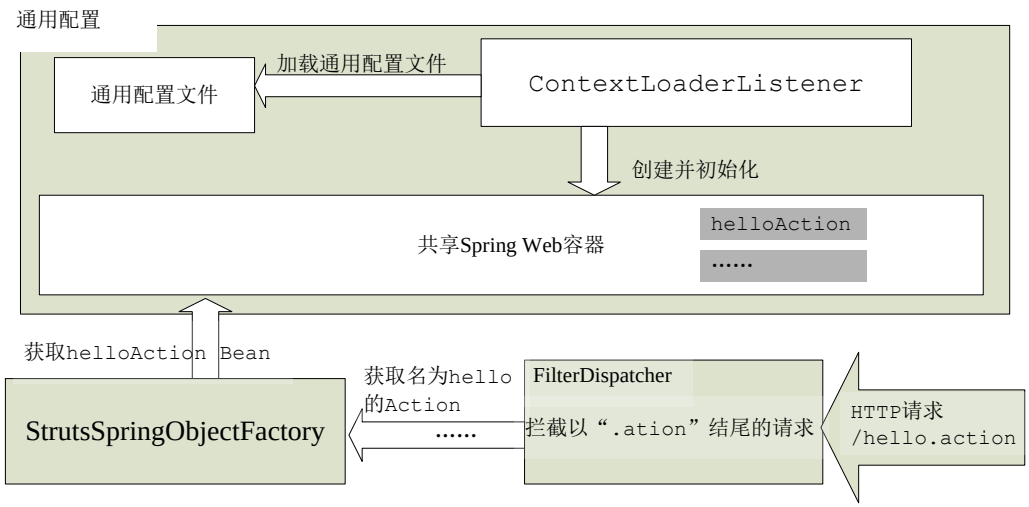


图 10-5 Strut2 与 Spring 集成

10.4 集成 JSF

10.4.1 概述

JSF（JavaServer Faces）框架是 Java EE 标准之一，是一个基于组件及事件驱动的 Web 框架，JSF 只是一个标准（规范），目前有很多厂家实现，如 Oracle 的默认标准实现 Mojarra、Apache 的 MyFaces、Jboss 的 RichFaces 等。

本示例将使用 Oracle 标准实现 Mojarra，请到官网 http://javaserverfaces.java.net/下载最新的 JSF 实现。

JSF 目前有 JSF1.1、JSF1.2、JSF2 版本实现。

Spring 集成 JSF 有三种种方式：

- **最简单集成：**使用 FacesContextUtils 工具类的 getWebApplicationContext 方法，类似于 Struts1x 中的最简单实现；
- **VariableResolver 实现：**Spring 提供 javax.faces.el.VariableResolver 的两种实

现 DelegatingVariableResolver 和 SpringBeanVariableResolver，此方式适用于 JSF1.1、JSF1.2 及 JSF2，但在 JSF1.2 和 JSF2 中不推荐使用该方式，而是使用第三种集成方式；

- **ELResolver 实现：**Spring 提供 javax.el.ELResolver （Unified EL）实现 SpringBeanFacesELResolver 用于集成 JSF1.2 和 JSF2。

接下来让我们首先让我们准备 JSF 所需要的 jar 包：
首先准备 JSF 所依赖的包：

```
commons-digester.jar                    //必须，已有
commons-collections.jar                 //必须，已有
commons-beanutils.jar                   //必须，已有
jsp-api.jar                             //必须，已有
servlet-api.jar                         //必须，已有
jstl.jar                                //可选
standard.jar                            //可选
```

准备 JSF 包，到 http://javaserverfaces.java.net/下载相应版本的 Mojarra 实现，如下载 JSF1.2 实现 mojarra-1.2_15-b01-FCS-binary.zip，拷贝如下 jar 包到类路径：

```
lib\jsf-api.jar                         //JSF 规范接口包
lib\jsf-impl.jar                        //JSF 规范实现包
```

10.4.2 最简单集成

类似于 Struts1x 中的最简单集成，Spring 集成 JSF 也提供类似的工具类 FacesContextUtils，使用如下方式获取 WebApplicationContext：

```
WebApplicationContext ctx =
    FacesContextUtils.getWebApplicationContext(FacesContext.getCurrentInstance());
```

当然我们不推荐这种方式，而是推荐使用接下来介绍的另外两种方式。

10.4.2 使用 VariableResolver 实现集成

Spring 提供 javax.faces.el.VariableResolver 的两种实现 DelegatingVariableResolver 和 SpringBeanVariableResolver，其都是 Spring 与 JSF 集成的中介者，此方式适用于 JSF1.1、JSF1.2 及 JSF2：

- DelegatingVariableResolver：首先委托给 JSF 默认 VariableResolver 实现去查找 JSF 管理 Bean，如果找不到再委托给 Spring 容器去查找 Spring 管理 Bean；

- SpringBeanVariableResolver：其与 DelegatingVariableResolver 查找正好相反，首先委托给 Spring 容器去查找 Spring 管理 Bean，如果找不到再委托给 JSF 默认 VariableResolver 实现去查找 JSF 管理 Bean。

接下来看一下如何在 JSF 中集成 Spring 吧（本示例使用 JSF1.2，其他版本的直接替换 jar 包即可）：

1、JSF 管理 Bean（Managed Bean）实现：

```
package cn.javass.spring.chapter10.jsf;
public class HelloBean {
    private String message;
    public void setMessage(String message) {
        this.message = message;
    }
    public String getMessage() {
        return message;
    }
}
```

2、JSF 配置文件定义（resources/chapter10/jsf/faces-config.xml）：

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config version="1.2" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd">

    <application>
        <variable-resolver>
            org.springframework.web.jsf.DelegatingVariableResolver
        </variable-resolver>
    </application>

    <managed-bean>
        <managed-bean-name>helloBean</managed-bean-name>
        <managed-bean-class>
            cn.javass.spring.chapter10.jsf.HelloBean
        </managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
        <managed-property>
            <property-name>message</property-name>
            <value>#{message}</value>
        </managed-property>
```

- **与 Spring 集成：**通过<variable-resolver>标签来指定集成 Spring 的中介者 DelegatingVariableResolver；
- **注入 Spring 管理 Bean：**通过<managed-property>标签的**<value>#{message}</value>**注入 Spring 管理 Bean“message”。

4、JSP 页面定义（webapp/hello-jsf.jsp）：

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html;
charset=UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
<html>
<head>
    <title>Hello World</title>
</head>
<body>
    <h:outputText value="#{helloBean.message}"/>
</body>
</html>
</f:view>
```

5、JSF 前端控制器定义，在 web.xml 中添加如下配置：

指定 JSF 配置文件位置，通过 javax.faces.CONFIG_FILES 上下文初始化参数指定 JSF 配置文件位置，多个可用“，”分割，如果不指定该参数则默认加载的配置文件为“/WEB-INF/ faces-config.xml”：

```
<!-- JSF配置文件开始 -->
<context-param>
    <param-name>javax.faces.CONFIG_FILES</param-name>
    <param-value>
        /WEB-INF/classes/chapter10/jsf/faces-config-jsf1x.xml
    </param-value>
</context-param>
<!-- JSF配置文件结束 -->
```

前端控制器定义：使用 FacesServlet 作为 JSF 的前端控制器，其拦截以“.jsf”结尾的 HTTP 请求：

```
<!-- jsf前端控制器配置开始     -->
<servlet>
      <servlet-name>jsf</servlet-name>
      <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
</servlet>
<servlet-mapping>
      <servlet-name>jsf</servlet-name>
    <url-pattern>*.jsf</url-pattern>
</servlet-mapping>
```

7、执行测试，在 Web 浏览器中输入 http://localhost:8080/hello-jsf.jsp 可以看到“Hello Spring”信息说明 JSF 集成成功。

自此，JSF 集成 Spring 已经成功，在此可以把 DelegatingVariableResolver 替换为 SpringBeanVariableResolver，其只有在查找相应依赖时顺序是正好相反的，其他完全一样。

如果您的项目使用 JSF1.2 或 JSF2，推荐使用 SpringBeanFacesELResolver，因为其实标准的 Unified EL 实现，而且 VariableResolver 接口已经被注释为@Deprecated，表示可能在以后的版本中去掉该接口。

10.4.2 使用 ELResolver 实现集成

JSF1.2 之前，JSP 和 JSF 各个使用自己的一套表达式语言（EL Language），即如 JSF 使用 VariableResolver 实现来解析 JSF EL 表达式，而从 JSF1.2 和 JSP2.1 开始使用 Unified EL，从而统一了表达式语言。

因此集成 JSF1.2+可以通过实现 Unified EL 来完成集成，即 Spring 提供 ELResolver 接口实现 SpringBeanFacesELResolver 用于集成使用。

类似于 VariableResolver 实现，通过 SpringBeanFacesELResolver 集成首先将从 Spring 容器中查找相应的 Spring 管理 Bean，如果没找到再通过默认的 JSF ELResolver 实现查找 JSF 管理 Bean。

接下来看一下示例一下吧：

1、 添加 Unified EL 所需要的 jar 包：

```
el-api.jar                                          //Unified EL 规范接口包
```

由于在 Jetty 中已经包含了该 api，因此该步骤可选。

2、 修改 JSF 配置文件（resources/chapter10/jsf/faces-config.xml）：

将如下配置

```
<variable-resolver>
    org.springframework.web.jsf.DelegatingVariableResolver
</variable-resolver>
```

修改为：

```
<el-resolver>
    org.springframework.web.jsf.el.SpringBeanFacesELResolver
</el-resolver>
```

3、执行测试，在 Web 浏览器中输入 http://localhost:8080/hello-jsf.jsp 可以看到“Hello Spring”信息说明 JSF 集成成功。

自此 JSF 与 Spring 集成就算结束了，是不是也很简单。

第十一章　SSH 集成开发

11.1　概述

11.1.1　功能概述

本节将通过介绍一个积分商城系统来演示如何使用 SSH 集成进行开发。

积分商城一般是购物网站的子模块，提供一些礼品或商品用于奖励老用户或使用积分来折换成现金，如图 11-1 所示。

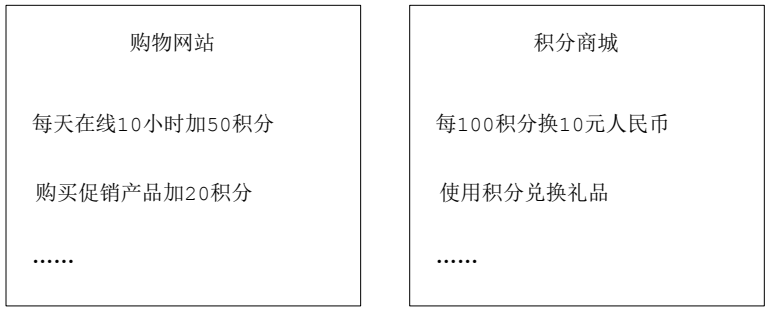


图 11-1 购物网站与积分商城

积分商城功能点：

- **后台管理**
  - **交易管理模块：**用于查看积分交易历史；
  - **商品管理模块：**用于 CRUD 积分兑换商品；
  - **日报或月报：**用于发送给运营人员每日积分兑换情况，一般通过 email 发送；
  - **……**
- **前台展示**
  - **商品展示：**展示给用户可以使用积分兑换的商品；
  - **支付模块：**用户成功兑换商品后扣除用户相应积分
  - **添加积分模块：**提供接口用于其他产品赠送积分使用，如每天在线 10 小时赠送 50 积分，购买相应商品增加相应积分；
  - **订单管理模块：**订单管理模块可以使用现有购物平台的订单管理。

购物平台、用户系统及积分商城交互如图 11-2 所示，其中用户系统负责用户登录，购物平台是购物网站核心，积分商城用于用户使用积分购买商品。

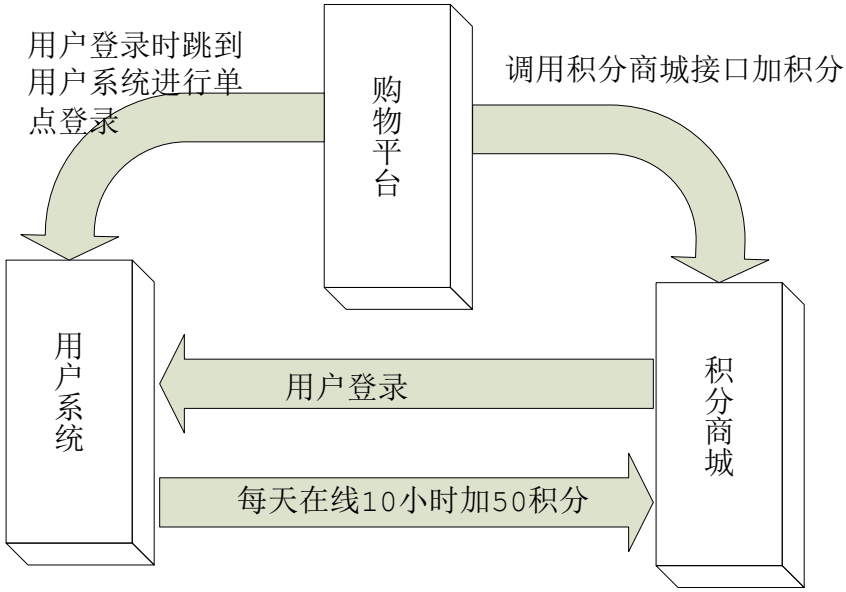


图 11-2 购物平台、用户系统及积分商城交互

由于积分商城也是很复杂，由于篇幅原因不打算完全介绍，只介绍其中一个模块——商品（兑换码）管理及购买，该模块主要提供给用户使用积分兑换一些优惠券或虚拟物品（如移动充值卡）等等。

11.1.2 技术选型

由于本节是关于 SSH 集成的，因此选用技术如下：

- 平台：Java EE；
- 运行环境：Windows XP，JDK1.6；
- 编辑器：Eclipse3.6 + SpringSource Tool Suite；
- Web 容器：tomcat 6.0.20；
- 数据库：mysql 5.4.3；
- 框架：Struts2.0.14、Spring3.0.5、Hibernate3.6.0.Final；
- 日志记录：log4j 1.2.15；
- 数据库连接池：proxool 0.9.1；
- 视图技术：JSP 2.0。

技术选定了，应该考虑平台架构了，这关系到项目的成功与否。

11.1.3 系统架构

积分商城系统架构也将采用经典的三层架构，如图 11-3 所示：

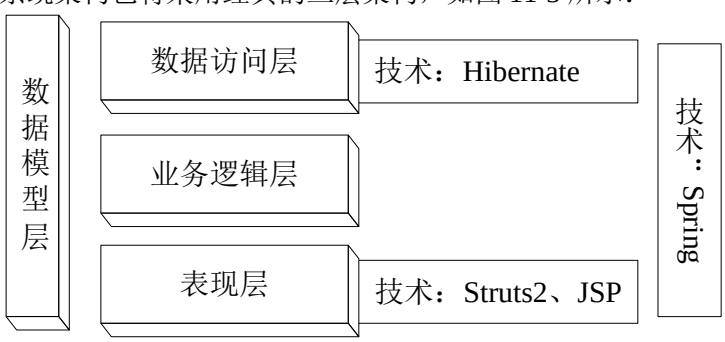


图 11-3 三层架构

分层的目的是约束层次边界，每层的职责和目标应明确和单一，每层专注自己的事情，不要跨越分层边界，具体每层功能如下：

- 数据访问层：封装底层数据库或文件系统访问细节，从而对业务逻辑层提供一致的接口，使业务逻辑层不关心底层细节；
- 业务逻辑层：专注于业务逻辑实现，不关心底层如何访问，并在该层实现如声明式事务管理，组装分页对象；
- 表现层：应该非常轻量级及非常“薄（功能非常少，几乎全是委托）”，拦

截用户请求并响应，表现层数据验证，负责根据请求委托给业务逻辑层进行业务处理，本层不实现任何业务逻辑，且提供用户交互界面；

- 数据模型层：数据模型定义，提供给各层使用，不应该算作三层架构中的某一层，因为数据模型可使用其他对象（如 Map）代替之。

系统架构已选定，在此我们进行优化一下，因为在进行基于 SSH 的三层架构进行开发时通常会有一些通用功能、如通用 DAO、通用 Service、通用 Action、通用翻页等等，因此我们再进行开发时都是基于通用功能进行的，能节省不少开发时间，从而可以使用这些节约的时间干自己想干的事情，如图 10-4 所示。

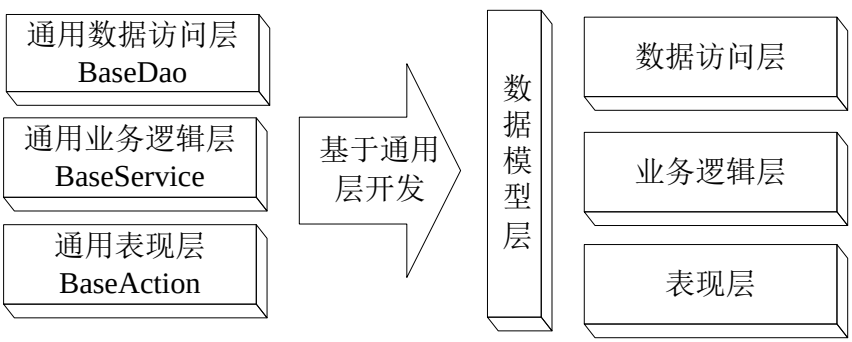


图 10-4 基于通用层的三层架构

11.1.4 项目搭建

1、创建动态 web 工程：

通过【File】>【New】>【other】>【Web】>【Dynamic Web Project】创建一个 Web 工程，如图 11-5 所示；

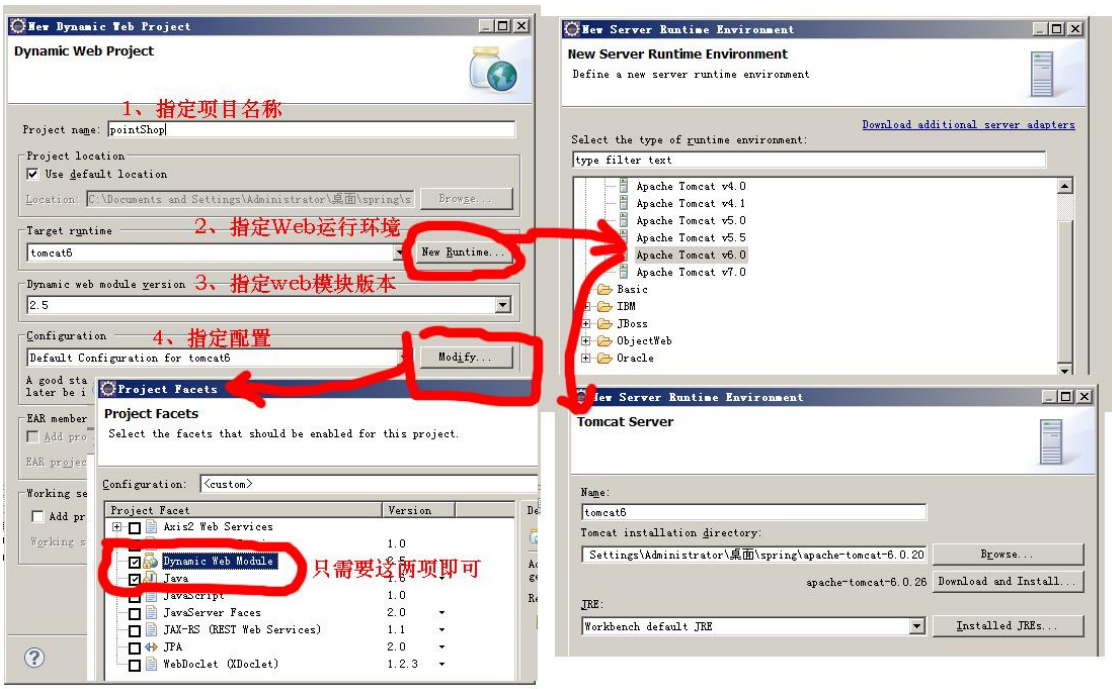
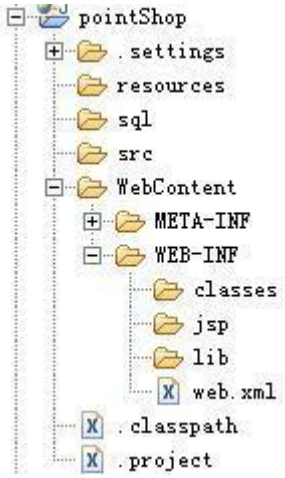


图 11-5 Web 工程配置

2、 项目结构，如图 11-6 所示：



resources：放置配置文件；

sql：放置 DDL 或 DML 语句，如数据库创建语句；

src：java 文件位置；

WebContent：web 项目根目录；

classes：存放编译好的 class 文件；

jsp：放置 jsp 视图文件；

lib 放置 jar 包；

web.xml：Web 项目部署描述符文件。

图 11-6 项目结构

3、项目属性修改：

3.1、字符编码修改，如图 11-7 所示，在实际项目中一定要统一字符编码：

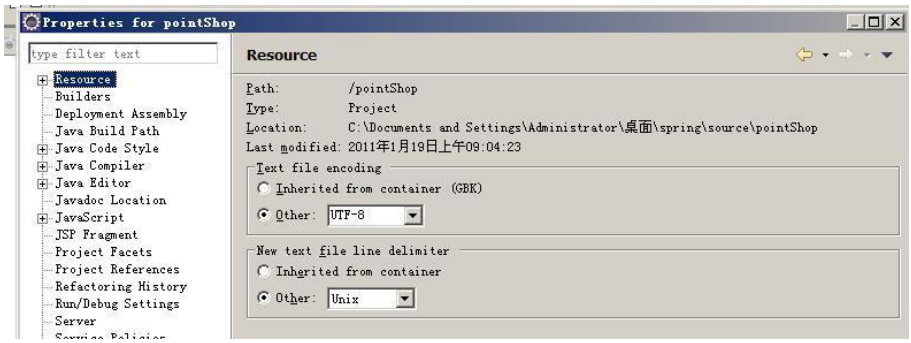


图 11-7 修改项目字符编码

3.2、类路径输出修改，如图 11-8，将类路径输出改为/WEB-INF/classes 下：

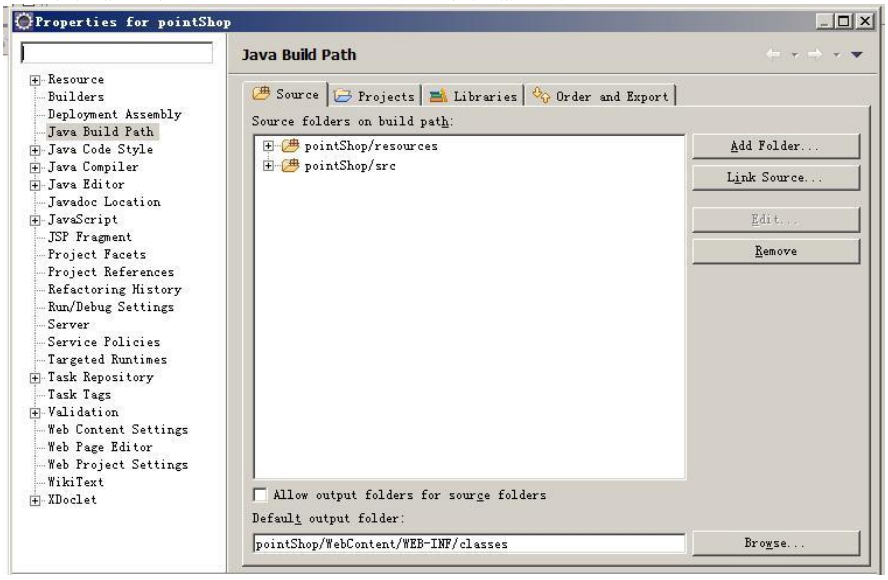


图 11-8 类路径修改

4、准备 jar 包：

4.1、Spring 项目依赖包，到下载的 spring-framework-3.0.5.RELEASE-with-docs.zip 中拷贝如下 jar 包：

dist\org.springframework.aop-3.0.5.RELEASE.jar
dist\org.springframework.asm-3.0.5.RELEASE.jar
dist\org.springframework.beans-3.0.5.RELEASE.jar
dist\org.springframework.context-3.0.5.RELEASE.jar
dist\org.springframework.core-3.0.5.RELEASE.jar
dist\org.springframework.expression-3.0.5.RELEASE.jar
dist\org.springframework.jdbc-3.0.5.RELEASE.jar
dist\org.springframework.orm-3.0.5.RELEASE.jar
dist\org.springframework.transaction-3.0.5.RELEASE.jar
dist\org.springframework.web-3.0.5.RELEASE.jar

4.1、Spring 及其他项目依赖包，到 spring-framework-3.0.5.RELEASE-dependencies.zip 中拷贝如下 jar 吧：

```
com.springsource.net.sf.cglib-2.2.0.jar
com.springsource.org.aopalliance-1.0.0.jar
com.springsource.org.apache.commons.beanutils-1.8.0.jar
com.springsource.org.apache.commons.collections-3.2.1.jar
com.springsource.org.apache.commons.digester-1.8.1.jar
com.springsource.org.apache.commons.logging-1.1.1.jar
com.springsource.org.apache.log4j-1.2.15.jar
com.springsource.org.apache.taglibs.standard-1.1.2.jar
com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar
```

4.2、Hibernate 依赖包，到 hibernate-distribution-3.6.0.Final.zip 中拷贝如下 jar 包：

```
hibernate3.jar
lib\jpa\hibernate-jpa-2.0-api-1.0.0.Final.jar
lib\required\dom4j-1.6.1.jar
lib\required\javassist-3.12.0.GA.jar
lib\required\jta-1.1.jar
lib\ required \slf4j-api-1.6.1.jar
lib\required\antlr-2.7.6.jar
```

4.3、数据库连接池依赖包，到 proxool-0.9.1.zip 中拷贝如下 jar 包：

```
lib\proxool-0.9.1.jar
lib\proxool-cglib.jar
```

4.4、准备 mysql JDBC 连接依赖包：

```
mysql-connector-java-5.1.10.jar
```

4.5、slf4j 依赖包准备，到下载的 slf4j-1.6.1.zip 包中拷贝如下 jar 包：

```
slf4j-log4j12-1.6.1.jar
```

4.6、Strut2 依赖包，到 struts-2.2.1.1.zip 中拷贝如下 jar 包：

```
lib\struts2-core-2.2.1.1.jar
lib\xwork-core-2.2.1.1.jar
lib\freemarker-2.3.16.jar
lib\ognl-3.0.jar
lib\struts2-spring-plugin-2.2.1.1.jar
lib\commons-fileupload-1.2.1.jar
```

jar 包终于准备完了，是不是很头疼啊，在此推荐使用 maven 进行依赖管理，无需拷贝这么多 jar 包，而是通过配置方式来指定使用的依赖，具体 maven 知识请到官方网站 http://maven.apache.org/了解。

11.2 实现通用层

11.2.1 功能概述

通过抽象通用的功能，从而复用，减少重复工作：

- 对于一些通用的常量使用一个专门的常量类进行定义；
- 对于视图分页，也应该抽象出来，如 JSP 做出 JSP 标签；
- 通用的数据层代码，如通用的 CRUD，减少重复劳动，节约时间；
- 通用的业务逻辑层代码，如通用的 CRUD，减少重复劳动，节约时间；
- 通用的表现层代码，同样用于减少重复，并提供更好的代码结构规范。

11.2.2 通用的常量定义

目标：在一个常量类中定义通用的常量的好处是如果需要修改这些常量值只需在一个地方修改即可，变的地方只有一处而不是多处。

如默认分页大小如果在多处硬编码定义为 10，突然发生变故需要将默认分页大小 10 为 5，怎么办？如果当初我们提取出来放在一个通用的常量类中是不是只有一处变动。

```
package cn.javass.commons;
public class Constants {
    public static final int DEFAULT_PAGE_SIZE = 5;   //默认分页大小
    public static final String DEFAULT_PAGE_NAME = "page";
    public static final String CONTEXT_PATH = "ctx";
}
```

如上代码定义了通用的常量，如默认分页大小。

11.2.2 通用分页功能

分页功能是项目中必不可少的功能，因此通用分页功能十分有必要，有了通用的分页功能，即有了规范，从而保证视图页面的干净并节约了开发时间。

1、 分页对象定义，用于存放是否有上一页或下一页、当前页记录、当前页码、分页上下文，该对象是分页中必不可少对象，一般在业务逻辑层组装 Page 对象，然后传送到表现层展示，然后通用的分页标签使用该对象来决定如何显示分页：

```
package cn.javass.commons.pagination;
import java.util.Collections;
import java.util.List;
public class Page<E> {/** 表示分页中的一页。*/
    private boolean hasPre; //是否有上一页
    private boolean hasNext; //是否有下一页
    private List<E> items; //当前页包含的记录列表
    private int index; //当前页页码(起始为 1)
    //省略 setter
    public int getIndex() {
        return this.index;
    }
    public boolean isHasPre() {
        return this.hasPre;
    }
    public boolean isHasNext() {
        return this.hasNext;
    }
    public List<E> getItems() {
        return this.items == null ? Collections.<E>emptyList() : this.items;
    }
}
```

2、 **分页标签实现**，将使用 Page 对象数据决定如何展示分页，如图 11-9 和 11-10 所示：

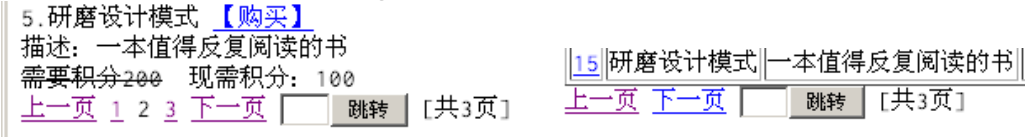


图 11-9 通用分页标签实现

图 11-9 和 11-10 展示了两种分页展示策略，由于分页标签和集成 SSH 没多大关系且不是必须的并由于篇幅问题不再列出分页标签源代码，有兴趣的朋友请参考 cn.javass.commons.pagination.NavigationTag 类文件。

11.2.3 通用数据访问层

目标：通过抽象实现最基本的 CURD 操作，提高代码复用，可变部分按需实现。

1、通用数据访问层接口定义

```
package cn.javass.commons.dao;
import java.io.Serializable;
import java.util.List;
public interface IBaseDao<M extends Serializable, PK extends Serializable> {
    public void save(M model);// 保存模型对象
    public void saveOrUpdate(M model);// 保存或更新模型对象
    public void update(M model);// 更新模型对象
    public void merge(M model);// 合并模型对象状态到底层会话
    public void delete(PK id);// 根据主键删除模型对象
    public M get(PK id);// 根据主键获取模型对象
    public int countAll();//统计模型对象对应数据库表中的记录数
    public List<M> listAll();//查询所有模型对象
    public List<M> listAll(int pn, int pageSize);// 分页获取所有模型对象
}
```

通用 DAO 接口定义了如 CRUD 通用操作，而可变的（如查询所有已发布的接口，即有条件查询等）需要在相应 DAO 接口中定义，并通过泛型“M”指定数据模型类型和“PK”指定数据模型主键类型。

2、通用数据访问层 DAO 实现

此处使用 Hibernate 实现，即实现是可变的，对业务逻辑层只提供面向接口编程，从而隐藏数据访问层实现细节。

实现时首先通过反射获取数据模型类型信息，并根据这些信息获取 Hibernate 对应的数据模型的实体名，再根据实体名组装出通用的查询和统计记录的 HQL，从而达到同样目的。

注意我们为什么把实现生成 HQL 时放到 init 方法中而不是构造器中呢？因为 SessionFactory 是通过 setter 注入，setter 注入晚于构造器注入，因此在构造器中使用 SessionFactory 会是 null，因此放到 init 方法中，并在 Spring 配置文件中指定初始化方法为 init 来完成生成 HQL。

```
package cn.javass.commons.dao.hibernate;
//为节省篇幅省略 import
public abstract class BaseHibernateDao<M extends Serializable, PK extends Serializable>
    extends HibernateDaoSupport implements IBaseDao<M, PK> {
    private Class<M> entityClass;
    private String HQL_LIST_ALL;
    private String HQL_COUNT_ALL;
    @SuppressWarnings("unchecked")
    public void init() {//通过初始化方法在依赖注入完毕时生成HQL
        //1、通过反射获取注解“M”（即模型对象）的类类型
        this.entityClass =
            (Class<M>) ((ParameterizedType) getClass().getGenericSuperclass()).
                getActualTypeArguments()[0];
        //2、得到模型对象的实体名
        String entityName = getSessionFactory().getClassMetadata(this.entityClass).
```

```
public void merge(M model) {
    getHibernateTemplate().merge(model);
}
public void delete(PK id) {
    getHibernateTemplate().delete(this.get(id));
}
public M get(PK id) {
    return getHibernateTemplate().get(this.entityClass, id);
}
public int countAll() {
    Number total = unique(getCountAllHql());
    return total.intValue();
}
public List<M> listAll() {
    return list(getListAllHql());
}
public List<M> listAll(int pn, int pageSize) {
```

```
                Query query = session.createQuery(hql);
                if (paramlist != null) {
                    for (int i = 0; i < paramlist.length; i++) {
                        query.setParameter(i, paramlist[i]);//设置占位符参数
                    }
                }
                if (pn > -1 && pageSize > -1) {//分页处理
                    query.setMaxResults(pageSize);//设置将获取的最大记录数
                    int start = PageUtil.getPageStart(pn, pageSize);
                    if(start != 0) {
                        query.setFirstResult(start);//设置记录开始位置
                    }
                }
                return query.list();
            }
        });
    }
```

通用 DAO 实现代码相当长，但麻烦一次，以后有了这套通用代码将会让工作很轻松，该通用 DAO 还有其他便利方法因为本示例不需要且由于篇幅原因没加上，请参考源代码。

11.2.4 通用业务逻辑层

目标：实现通用的业务逻辑操作，将常用操作封装提高复用，可变部分同样按需实现。

1、通用业务逻辑层接口定义

```
package cn.javass.commons.service;
//由于篇幅问题省略 import
public interface IBaseService<M extends Serializable, PK extends Serializable> {
    public M save(M model); //保存模型对象
    public void saveOrUpdate(M model);// 保存或更新模型对象
    public void update(M model);// 更新模型对象
    public void merge(M model);// 合并模型对象状态
    public void delete(PK id);// 删除模型对象
    public M get(PK id);// 根据主键获取模型对象
    public int countAll();//统计模型对象对应数据库表中的记录数
```

3、 通用业务逻辑层接口实现

通用业务逻辑层通过将通用的持久化操作委托给 DAO 层来实现通用的数据模型 CRUD 等操作。

通过通用的 setDao 方法注入通用 DAO 实现，在各 Service 实现时可以通过强制转型获取各转型后的 DAO。

```
package cn.javass.commons.service.impl;
//由于篇幅问题省略 import
public abstract class BaseServiceImpl<M extends Serializable, PK extends Serializable>
    implements IBaseService<M, PK> {
    protected IBaseDao<M, PK> dao;
    public void setDao(IBaseDao<M, PK> dao) {//需要依赖注入
        this.dao = dao;
    }
    public IBaseDao<M, PK> getDao() {
        return this.dao;
    }
```

```
    public M save(M model) {
        getDao().save(model);
        return model;
    }
    public void merge(M model) {
        getDao().merge(model);
    }
    public void saveOrUpdate(M model) {
        getDao().saveOrUpdate(model);
    }
    public void update(M model) {
        getDao().update(model);
    }
    public void delete(PK id) {
        getDao().delete(id);
    }
    public void deleteObject(M model) {
```

11.2.6 通用表现层

目标：规约化常见请求和响应操作，将常见的 CURD 规约化，采用规约编程提供开发效率，减少重复劳动。

Struts2 常见规约编程：

- **通用字段驱动注入：**如分页字段一般使用“pn”或“page”来指定当前分页页码参数名，通过 Struts2 的字段驱动注入实现分页页码获取的通用化；
- **通用 Result：**对于 CURD 操作完全可以提取公共的 Result 名字，然后在 Strust2 配置文件中进行规约配置；
- **数据模型属性名：**在页面展示中，如新增和修改需要向值栈或请求中设置数据模型，在此我们定义统一的数据模型名如“model”，这样在项目组中形成约定，大家只要按照约定来能提高开发效率；

- **分页对象属性名：**与数据模型属性名同理，在此我们指定为“page”；
- **便利方法：**如获取值栈、请求等可以提供公司内部需要的便利方法。

1、通用表现层 Action 实现：

```
package cn.javass.commons.web.action;
import cn.javass.commons.Constants;
//省略 import
public class BaseAction extends ActionSupport {
    /** 通用 Result */
    public static final String LIST = "list";
    public static final String REDIRECT = "redirect";
    public static final String ADD = "add";
    /** 模型对象属性名*/
    public static final String MODEL = "model";
    /** 列表模型对象属性名*/
    public static final String PAGE = Constants.DEFAULT_PAGE_NAME;
    public static final int DEFAULT_PAGE_SIZE =
        Constants.DEFAULT_PAGE_SIZE;
    private int pn = 1;   /** 页码，默认为 1 */
    //省略 pn 的 getter 和 setter，自己补上
    public ActionContext getActionContext() {
        return ActionContext.getContext();
    }
    public ValueStack getValueStack() {//获取值栈的便利方法
        return getActionContext().getValueStack();
    }
}
```

2、通用表现层 JSP 视图实现：

将视图展示的通用部分抽象出来减少页面设计的工作量。

2.1、通用 JSP 页头文件（WEB-INF/jsp/common/inc/header.jsp）：

此处实现比较简单，实际中可能包含如菜单等信息，对于可变部分使用请求参数来获取，从而保证了可变与不可变分离，如标题使用“${param.title}”来获取。

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html;
charset=UTF-8"%>
<!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=UTF-8">
    <title>${param.title}</title>
</head>
<body>
```

2.2、通用 JSP 页尾文件（WEB-INF/jsp/common/inc/footer.jsp）：

此处比较简单，实际中可能包含公司版权等信息。

```
</body>
</html>
```

2.3、通用 JSP 标签定义文件（WEB-INF/jsp/common/inc/tld.jsp）：

在一处定义所有标签，避免标签定义使代码变得凌乱，且如果有多个页面需要新增或删除标签即费事又费力。

```
<%@taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<%@taglib prefix="s" uri="/struts-tags" %
```

2.3、通用错误 JSP 文件（WEB-INF/jsp/common/error.jsp）：

当系统遇到错误或异常时应该跳到该页面来显示统一的错误信息并可能在该页保存异常信息。

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html;
charset=UTF-8"%>
<jsp:include page="inc/header.jsp"/>
失败或遇到异常！
<jsp:include page="inc/footer.jsp"/>
```

2.4、通用正确 JSP 文件（WEB-INF/jsp/common/error.jsp）：

对于执行成功的操作可以使用通用的页面表示，可变部分同样可以使用可变的请求参数传入。

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html;
charset=UTF-8"%>
<jsp:include page="inc/header.jsp"/>
成功！
<jsp:include page="inc/footer.jsp"/>
```

3、通用设置 web 环境上下文路径拦截器：

用于设置当前 web 项目的上下文路径，即可以在 JSP 页面使用“${ctx}”获取当前上下文路径。

```
package cn.javass.commons.web.filter;
//省略 import
/** 用户设置当前 web 环境上下文，用于方便如 JSP 页面使用 */
public class ContextPathFilter implements Filter {
    @Override
    public void init(FilterConfig config) throws ServletException {
```

11.2.7 通用配置文件

目标：通用化某些常用且不可变的配置文件，同样目标是提高复用，减少工作量。

1、Spring 资源配置文件（resources/applicationContext-resources.xml）：

定义如配置元数据替换 Bean、数据源 Bean 等通用的 Bean。

```
<bean class=
      "org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
      <property name="locations">
            <list>
                  <value>classpath:resources.properties</value>
            </list>
      </property>
</bean>
<bean id="dataSource"
    class="org.springframework.jdbc.datasource.LazyConnectionDataSourceProxy">
      <property name="targetDataSource">
            <bean class="org.logicalcobwebs.proxool.ProxoolDataSource">
                  <property name="driver" value="${db.driver.class}" />
                  <property name="driverUrl" value="${db.url}" />
                  <property name="user" value="${db.username}" />
                  <property name="password" value="${db.password}" />
```

通过通用化如数据源来提高复用，对可变的如数据库驱动、URL、用户名等采用替换配置元数据形式进行配置，具体配置含义请参考【7.5 集成 Spring JDBC 及最佳实践】。

2、替换配置元数据的资源文件（resources/resources.properties）：

定义替换配置元数据键值对用于替换 Spring 配置文件中可变的配置元数据。

```
#数据库连接池属性
proxool.maxConnCount=10
proxool.minConnCount=5
proxool.statistics=1m,15m,1h,1d
proxool.simultaneousBuildThrottle=30
```

```
proxool.trace=false
db.driver.class=com.mysql.jdbc.Driver
db.url=jdbc:mysql://localhost:3306/point_shop?useUnicode=true&c
haracterEncoding=utf8
db.username=root
```

3、通用 Struts2 配置文件（WEB-INF/struts.xml）：

由于是要集成 Spring，因此需要使用 StrutsSpringObjectFactory，我们需要在 action 名字中出现“/”因此定义 struts.enable.SlashesInActionNames=true。

在此还定义了“custom-default”包继承 struts-default 包，且是抽象的，在包里定义了如全局结果集全局异常映射。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
      "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
      "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
```

4、通用 log4j 日志记录配置文件（resources/log4j.xml）：

可以配置基本的 log4j 配置文件然后在其他地方通过拷贝来定制需要的日志记录配置。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
    <!-- Appenders -->
    <appender name="console" class="org.apache.log4j.ConsoleAppender">
        <param name="Target" value="System.out" />
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%-5p: %c - %m%n" />
        </layout>
    </appender>
    <!-- Root Logger -->
    <root>
        <priority value="DEBUG" />
        <appender-ref ref="console" />
    </root>
</log4j:configuration>
```

4、通用 web.xml 配置文件定义（WEB-INF/web.xml）：

定义如通用的集成配置、设置 web 环境上下文过滤器、字符过滤器（防止乱码）、通用的 Web 框架拦截器（如 Struts2 的）等等，从而可以通过拷贝复用。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.5"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <!-- 通用配置开始    -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            classpath:applicationContext-resources.xml
        </param-value>
    </context-param>
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
```

```
    </listener>
    <!-- 通用配置结束    -->
    <!-- 设置web环境上下文（方便JSP页面获取）开始    -->
    <filter>
        <filter-name>Set Context Path</filter-name>
        <filter-class>cn.javass.commons.web.filter.ContextPathFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>Set Context Path</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <!-- 设置web环境上下文（方便JSP页面获取）结束    -->
    <!-- 字符编码过滤器（防止乱码）开始    -->
    <filter>
        <filter-name>Set Character Encoding</filter-name>
        <filter-class>
```

```
    <filter-mapping>
        <filter-name>struts2Filter</filter-name>
        <url-pattern>*.action</url-pattern>
    </filter-mapping>
    <!-- Struts2.x前端控制器配置结束    -->
</web-app>
```

11.3 实现积分商城层

11.3.1 概述

积分商城是基于通用层之上进行开发，这样我们能减少很多重复的劳动，加快项目开发进度。

11.3.2 实现数据模型层

1、商品表，定义了如商品名称、简介、原需积分、现需积分等，其中是否发布表示只有发布（true）了的商品才会在前台删除，是否已删除表示不会物理删除，商品不应该物理删除，而是逻辑删除，版本属性用于防止并发更新。

```
package cn.javass.point.model;
/** 商品表 */
@Entity
@Table(name = "tb_goods")
public class GoodsModel implements java.io.Serializable {
    /** 主键 */
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", length = 10)
    private int id;
    /** 商品名称 */
    @Column(name = "name", nullable = false, length = 100)
    private String name;
    /** 商品简介 */
    @Column(name = "description", nullable = false, length = 100)
    private String description;
    /** 原需积分 */
    @Column(name = "original_point", nullable = false, length = 10)
    private int originalPoint;
```

```
/** 现需积分 */
    @Column(name = "now_point", nullable = false, length = 10)
    private int nowPoint;
    /** 是否发布，只有发布的在前台显示 */
    @Column(name = "published", nullable = false)
    private boolean published;
    /** 是否删除，商品不会被物理删除的 */
    @Column(name = "is_delete", nullable = false)
    private boolean deleted;
    /** 版本 */
    @Version   @Column(name = "version", nullable = false, length = 10)
```

2、商品兑换码表，定义了兑换码、兑换码所属商品（兑换码和商品直接是多对一关系）、购买人、购买时间、是否已经购买（防止一个兑换码多个用户兑换）、版本。

```
package cn.javass.point.model;
import java.util.Date;
//省略部分 import
/** 商品兑换码表 */
@Entity
@Table(name = "tb_goods_code")
public class GoodsCodeModel implements java.io.Serializable {
    /** 主键 */
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", length = 10)
    private int id;
    /** 所属商品 */
    @ManyToOne
    private GoodsModel goods;
    /** 兑换码*/
    @Column(name = "code", nullable = false, length = 100)
    private String code;
    /** 兑换人,实际环境中应该和用户表进行对应*/
    @Column(name = "username", nullable = true, length = 100)
    private String username;
    /** 兑换时间*/
    @Column(name = "exchange_time")
    private Date exchangeTime;
    /** 是否已经兑换*/
    @Column(name = "exchanged")
    private boolean exchanged = false;
    /** 版本 */
    @Version
    @Column(name = "version", nullable = false, length = 10)
    private int version;
    //省略 getter 和 setter、hashCode 及 equals，实现请参考源代码
```

4、 **商品表及商品兑换码表之间关系，**即一个商品有多个兑换码，如图 11-10 所示：

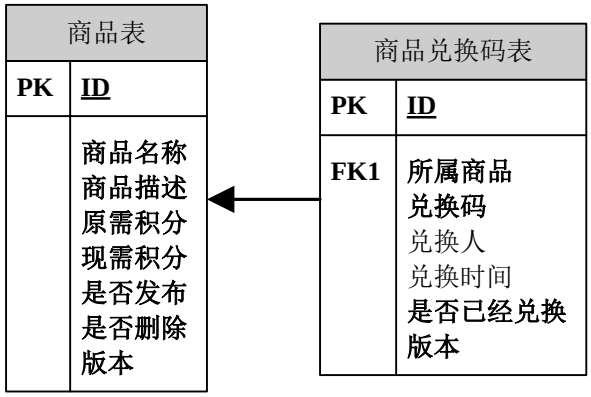


图 11-10 商品表及商品兑换码表之间关系

5、 创建数据库及表结构的 SQL 语句文件（sql/ pointShop_schema.sql）：

```
CREATE DATABASE IF NOT EXISTS `point_shop`
      DEFAULT CHARACTER SET 'utf8';
USE `point_shop`;
DROP TABLE IF EXISTS `tb_goods_code`;
DROP TABLE IF EXISTS `tb_goods`;
-- ----------------------------
-- Table structure for  商品表
-- ----------------------------
CREATE TABLE `tb_goods` (
   `id` int(10) unsigned NOT NULL AUTO_INCREMENT COMMENT '商品id',
   `name` varchar(100) NOT NULL COMMENT '商品名称',
   `description` varchar(100) NOT NULL COMMENT '商品简介',
```

```
   `original_point` int(10) unsigned NOT NULL    COMMENT '原需积分',
   `now_point` int(10) unsigned NOT NULL    COMMENT '现需积分',
   `published` bool NOT NULL    COMMENT '是否发布',
   `is_delete` bool NOT NULL DEFAULT false COMMENT '是否删除',
   `version` int(10) unsigned NOT NULL DEFAULT 0 COMMENT '版本',
   PRIMARY KEY (`id`),
   INDEX(`name`),
   INDEX(`published`)
   )ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='商品表';
-- ----------------------------
-- Table structure for  商品兑换码表
```

Mysql 数据库引擎应该使用 InnoDB，如果使用 MyISM 将不支持事务。

11.3.3 实现数据访问层

数据访问层只涉及与底层数据库或文件系统等打交道，不会涉及业务逻辑，一定注意层次边界，不要在数据访问层实现业务逻辑。

商品模块的应该实现如下功能：

- 继承通用数据访问层的 CRUD 功能；
- 分页查询所有已发布的商品
- 统计所有已发布的商品；

商品兑换码模块的应该实现如下功能：

- 继承通用数据访问层的 CRUD 功能；
- 根据商品 ID 分页查询该商品的兑换码
- 根据商品 ID 统计该商品的兑换码记录数；
- 根据商品 ID 获取一个还没有兑换的商品兑换码

1、商品及商品兑换码 DAO 接口定义：

商品及商品兑换码 DAO 接口定义直接继承 IBaseDao，无需在这些接口中定义重复的 CRUD 方法了，并通过泛型指定数据模型类及主键类型。

```
package cn.javass.point.dao;
//省略 import
/** 商品模型对象的 DAO 接口 */
public interface IGoodsDao extends IBaseDao<GoodsModel, Integer> {
    /** 分页查询所有已发布的商品*/
    List<GoodsModel> listAllPublished(int pn);
```

```
package cn.javass.point.dao;
//省略 import
/** 商品兑换码模型对象的 DAO 接口 */
public interface IGoodsCodeDao extends IBaseDao<GoodsCodeModel, Integer> {
    /** 根据商品 ID 统计该商品的兑换码记录数*/
    public int countAllByGoods(int goodsId);
    /** 根据商品ID查询该商品的兑换码列表*/
    public List<GoodsCodeModel> listAllByGoods(int pn, int goodsId);
    /** 根据商品ID获取一个还没有兑换的商品兑换码 */
    public GoodsCodeModel getOneNotExchanged(int goodsId);
}
```

2、 商品及商品兑换码 DAO 接口实现定义：

DAO 接口实现定义都非常简单，对于 CRUD 实现直接从 BaseHibernateDao 继承即可，无需再定义重复的 CRUD 实现了，并通过泛型指定数据模型类及主键类型。

```
package cn.javass.point.dao.hibernate;
//省略 import
public class GoodsHibernateDao extends BaseHibernateDao<GoodsModel, Integer>
    implements IGoodsDao {
    @Override  //覆盖掉父类的 delete 方法，不进行物理删除
    public void delete(Integer id) {
        GoodsModel goods = get(id);
        goods.setDeleted(true);
        update(goods);
    }
    @Override //覆盖掉父类的 getCountAllHql 方法，查询不包括逻辑删除的记录
    protected String getCountAllHql() {
        return super.getCountAllHql() + " where deleted=false";
    }
    @Override //覆盖掉父类的 getListAllHql 方法，查询不包括逻辑删除的记录
    protected String getListAllHql() {
        return super.getListAllHql() + " where deleted=false";
    }
```

```
package cn.javass.point.dao.hibernate;
//省略import
public class GoodsCodeHibernateDao extends
      BaseHibernateDao<GoodsCodeModel, Integer> implements IGoodsCodeDao {
      @Override   //根据商品ID查询该商品的兑换码
      public List<GoodsCodeModel> listAllByGoods(int pn, int goodsId) {
            final String hql = getListAllHql() + " where goods.id = ?";
            return list(hql, pn, Constants.DEFAULT_PAGE_SIZE , goodsId);
      }
      @Override //根据商品ID统计该商品的兑换码数量
      public int countAllByGoods(int goodsId) {
            final String hql = getCountAllHql() + " where goods.id = ?";
            Number result = unique(hql, goodsId);
            return result.intValue();
      }
```

```
     @Override //获取一个指定商品的没有被兑换的兑换码
      public GoodsCodeModel getOneNotExchanged(int goodsId) {
            String hql = getListAllHql() + " where goods.id = ? and exchanged=false";
            List<GoodsCodeModel> result = list(hql, goodsId);
            if(CollectionUtils.isEmpty(result)) {
                  return null;
            }
            return result.get(0);
      }
}
```

3 、 Spring DAO 层 配 置 文 件 （ resources/cn/javass/point/dao/applicationContext-hibernate.xml）：

DAO 配置文件中定义 Hibernate 的 SessionFactory、事务管理器和 DAO 实现。

```
<bean id="sessionFactory" class="
    org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/><!-- 1、指定数据源 -->
    <property name="annotatedClasses">                <!-- 2、指定注解类 -->
        <list>
            <value>cn.javass.point.model.GoodsModel</value>
            <value>cn.javass.point.model.GoodsCodeModel</value>
        </list>
    </property>
    <property name="hibernateProperties"><!-- 3、指定Hibernate属性 -->
        <props>
            <prop key="hibernate.dialect">${hibernate.dialect}</prop>
            <prop key="hibernate.show_sql">${hibernate.show_sql}</prop>
            <prop key="hibernate.format_sql">${hibernate.format_sql}</prop>
            <prop key="hibernate.hbm2ddl.auto">${hibernate.hbm2ddl.auto}</prop>
        </props>
    </property>
</bean>
<bean id="txManager"
      class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

```
<bean id="abstractDao" abstract="true" init-method="init">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
<bean id="goodsDao"
      class="cn.javass.point.dao.hibernate.GoodsHibernateDao"
      parent="abstractDao"/>
<bean id="goodsCodeDao"
      class="cn.javass.point.dao.hibernate.GoodsCodeHibernateDao"
      parent="abstractDao"/>
```

4、修改替换配置元数据的资源文件（resources/resources.properties），添加 Hibernate 属性相关：

```
#Hibernate属性
hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect
hibernate.hbm2ddl.auto=none
hibernate.show_sql=false
hibernate.format_sql=true
```

11.3.4 实现业务逻辑层

业务逻辑层实现业务逻辑，即系统中最复杂、最核心的功能，不应该在业务逻辑层出现如数据库访问等底层代码，对于这些操作应委托给数据访问层实现，从而保证业务逻辑层的独立性和可复用性，并应该在业务逻辑层组装分页对象。

商品模块应该实现如下功能：

- CURD 操作，直接委托给通用业务逻辑层；
- 根据页码查询所有已发布的商品的分页对象，即查询指定页的记录，这是和数据访问层不同的；

商品兑换码模块应该实现如下功能：

- CURD 操作，直接委托给通用业务逻辑层；
- 根据页码和商品 Id 查询查询所有商品兑换码分页对象,即查询指定页的记录；
- 新增指定商品的兑换码，用于对指定商品添加兑换码；
- 购买指定商品兑换码操作，用户根据商品购买该商品的兑换码，如果指定商品的兑换码没有了将抛出没有兑换码异常 NotCodeException；

1、商品及商品兑换码 Service 接口定义：

接口定义时，对于 CRUD 直接继承 IBaseService 即可，无需再在这些接口中定义重复的 CRUD 方法了，并通过泛型指定数据模型类及数据模型的主键。

```
package cn.javass.point.service;
//省略 import
public interface IGoodsService extends IBaseService<GoodsModel, Integer> {
    /**根据页码查询所有已发布的商品的分页对象*/
    Page<GoodsModel> listAllPublished(int pn);
}
```

```
package cn.javass.point.service;
//省略 import
public interface IGoodsCodeService extends IBaseService<GoodsCodeModel, Integer> {
    /** 根据页码和商品 Id 查询查询所有商品兑换码分页对象*/
    public Page<GoodsCodeModel> listAllByGoods(int pn, int goodsId);
    /** 新增指定商品的兑换码*/
    public void save(int goodsId, String[] codes);
    /** 购买指定商品兑换码 */
    GoodsCodeModel buy(String username, int goodsId) throws NotCodeException ;
}
```

2、NotCodeException 异常定义，表示指定商品的兑换码已经全部被兑换了，没有剩余的兑换码了：

```
package cn.javass.point.exception;
/** 购买失败异常,表示没有足够的兑换码 */
public class NotCodeException extends RuntimeException {
}
```

NotCodeException 异常类实现 RuntimeException，当需要更多信息时可以在异常中定义，异常比硬编码错误代码（如-1 表示没有足够的兑换码）更好理解。

3、商品及商品兑换码 Service 接口实现定义：

接口实现时，CRUD 实现直接从 BaseServcice 继承即可，无需再在这些专有实现中定义重复的 CRUD 实现了，并通过泛型指定数据模型类及数据模型的主键。

```
package cn.javass.point.service.impl;
//省略 import
public class GoodsServiceImpl extends BaseServiceImpl<GoodsModel, Integer>
    implements IGoodsService {
    @Override
    public Page<GoodsModel> listAllPublished(int pn) {
        int count = getGoodsDao().countAllPublished();
        List<GoodsModel> items = getGoodsDao().listAllPublished(pn);
        return PageUtil.getPage(count, pn, items, Constants.DEFAULT_PAGE_SIZE);
    }
    IGoodsDao getGoodsDao() {//将通用 DAO 转型
        return (IGoodsDao) getDao();
    }
}
```

```
package cn.javass.point.service.impl;
//省略 import
public class GoodsCodeServiceImpl extends
    BaseServiceImpl<GoodsCodeModel, Integer> implements IGoodsCodeService {
    private IGoodsService goodsService;
    public void setGoodsService(IGoodsService goodsService) {//注入 IGoodsService
        this.goodsService = goodsService;
        }
    private IGoodsCodeDao getGoodsCodeDao() {//将注入的通用 DAO 转型
        return (IGoodsCodeDao) getDao();
    }
    @Override
    public Page<GoodsCodeModel> listAllByGoods(int pn, int goodsId) {
        Integer count = getGoodsCodeDao().countAllByGoods(goodsId);
        List<GoodsCodeModel> items =
            getGoodsCodeDao().listAllByGoods(pn, goodsId);
        return PageUtil.getPage(count, pn, items, Constants.DEFAULT_PAGE_SIZE);
    }
    @Override
    public void save(int goodsId, String[] codes) {
        GoodsModel goods = goodsService.get(goodsId);
        for(String code : codes) {
            if(StringUtils.hasText(code)) {
                GoodsCodeModel goodsCode = new GoodsCodeModel();
                goodsCode.setCode(code);
                goodsCode.setGoods(goods);
            save(goodsCode);
            }
        }
    }
    @Override
    public GoodsCodeModel buy(String username, int goodsId)
        throws NotCodeException {
        //1、实际实现时要验证用户积分是否充足
        //2、其他逻辑判断
        //3、实际实现时要记录交易记录开始
        GoodsCodeModel goodsCode =
            getGoodsCodeDao().getOneNotExchanged(goodsId);
```

```
            if(goodsCode == null) {
                //3、实际实现时要记录交易记录失败
                throw new NotCodeException();
                //目前只抛出一个异常，还可能比如并发购买情况
            }
            goodsCode.setExchanged(true);
            goodsCode.setExchangeTime(new Date());
            goodsCode.setUsername(username);
            save(goodsCode);
            //3、实际实现时要记录交易记录成功
            return goodsCode;
        }
}
```

save 方法和 buy 方法实现并不是最优的，save 方法中如果兑换码有上千个怎么办？这时就需要批处理了，通过批处理比如 20 条一提交数据库来提高性能。buy 方法就要考虑多个用户同时购买同一个兑换码如何处理？

交易历史一定要记录，从交易开始到交易结束（不管成功与否）一定要记录用于当客户投诉时查询相应数据。

4、Spring Service 层配置文件（resources/cn/javass/point/service/applicationContext-service.xml）：

Service 层配置文件定义了事务和 Service 实现。

```
<tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
        <tx:method name="save*" propagation="REQUIRED" />
        <tx:method name="add*" propagation="REQUIRED" />
        <tx:method name="create*" propagation="REQUIRED" />
        <tx:method name="insert*" propagation="REQUIRED" />
        <tx:method name="update*" propagation="REQUIRED" />
        <tx:method name="del*" propagation="REQUIRED" />
        <tx:method name="remove*" propagation="REQUIRED" />
        <tx:method name="buy*" propagation="REQUIRED" />
        <tx:method name="count*" propagation="SUPPORTS" read-only="true" />
        <tx:method name="find*" propagation="SUPPORTS" read-only="true" />
        <tx:method name="list*" propagation="SUPPORTS" read-only="true" />
        <tx:method name="*" propagation="SUPPORTS" read-only="true" />
    </tx:attributes>
</tx:advice>
```

```
<aop:config>
    <aop:pointcut id="txPointcut"
                  expression="execution(* cn.javass.point.service.*.*(..))" />
    <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut" />
</aop:config>
<bean id="goodsService"
     class="cn.javass.point.service.impl.GoodsServiceImpl">
    <property name="dao" ref="goodsDao"/>
</bean>
<bean id="goodsCodeService"
     class="cn.javass.point.service.impl.GoodsCodeServiceImpl">
    <property name="dao" ref="goodsCodeDao"/>
    <property name="goodsService" ref="goodsService"/>
</bean>
```

11.3.5 实现表现层

表现层显示页面展示和交互，应该支持多种视图技术（如 JSP、Velocity），表现层实现不应该实现诸如业务逻辑层功能，只负责调用业务逻辑层查找数据模型并委托给相应的视图进行展示数据模型。

积分商城分为前台和后台，前台负责与客户进行交互，如购买商品；后台是负责商品及商品兑换码维护的，只应该管理员有权限操作。

后台模块：

- 商品管理模块：负责商品的维护，包括列表、新增、修改、删除、查询所有商品兑换码功能；
- 商品兑换码管理模块：包括列表、新增、删除所有兑换码操作；

前台模块：只有已发布商品展示，用户购买指定商品时，如果购买成功则给用户发送兑换码，购买失败给用户错误提示。

表现层 Action 实现时一般使用如下规约编程：

- **Action 方法定义：**使用如 list 方法表示展示列表，doAdd 方法表示去新增页面，add 方法表示提交新增页面的结果并委托给 Service 层进行处理；
- **结果定义：**如使用“list”结果表示到展示列表页面，“add”结果去新增页面等等；
- **参数设置：**一般使用如“model”表示数据模型，使用“page”表示分页对象。

1、集成 Struts2 和 Spring 配置：

1.1、Spring Action 配置文件：即 Action 将从 Spring 容器中获取，前台和后台配置文件应该分开以便好管理；

- 后台 Action 配置文件 resources/cn/javass/web/pointShop-admin-servlet.xml；
- 前台 Action 配置文件 resources/cn/javass/web/pointShop-front-servlet.xml；

1.2、Struts 配置文件定义（resources/struts.xml）：

为了提高开发效率和采用规约编程，我们将使用模式匹配通配符来定义 action。对于管理后台和前台应该分开，URL 模式将类似于/{module}/{action}/{method}.action：

- module 即模块名如 admin，action 即 action 前缀名，如后台的“GoodsAction”可以使用“goods”，method 即 Action 中的方法名如“list”。
- 可以在 Struts 配置文件中使用{1}访问第一个通配符匹配的结果，以此类推；
- Reuslt 也采用规约编程，即只有符合规律的放置 jsp 文件才会匹配到，如 Result 为“/WEB-INF/jsp/admin/{1}/list.jsp”，而 URL 为/goods/list.action 结果将为“/WEB-INF/jsp/admin/goods/list.jsp”。

```
<package name="admin" extends="custom-default" namespace="/admin">
        <action name="*/*" class="/admin/{1}Action" method="{2}">
            <result name="redirect" type="redirect">/admin/{1}/list.action</result>
            <result name="list">/WEB-INF/jsp/admin/{1}/list.jsp</result>
            <result name="add">/WEB-INF/jsp/admin/{1}/add.jsp</result>
        </action>
    </package>
```

在此我们继承了“**custom-default**”包来支持 action 名字中允许“/”。

如“/admin/goods/list.action”将调用 cn.javass.point.web.admin.action.GoodsAction 的 list 方法。

```
<package name="front" extends="custom-default">
        <action name="*/*" class="/front/{1}Action" method="{2}">
            <result name="redirect" type="redirect">/{1}/list.action</result>
            <result name="list">/WEB-INF/jsp/front/{1}/list.jsp</result>
            <result name="add">/WEB-INF/jsp/front/{1}/add.jsp</result>
            <result name="buyResult">/WEB-INF/jsp/front/{1}/buyResult.jsp</result>
        </action>
    </package>
```

如“/goods/list.action”将调用 cn.javass.point.web.front.action.GoodsAction 的 list 方法。

1.3、web.xml 配置：将 Spring 配置文件加上；

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
```

```
            classpath:applicationContext-resources.xml,
            classpath:cn/javass/point/dao/applicationContext-hibernate.xml,
            classpath:cn/javass/point/service/applicationContext-service.xml,
            classpath:cn/javass/point/web/pointShop-admin-servlet.xml,
            classpath:cn/javass/point/web/pointShop-front-servlet.xml
        </param-value>
</context-param>
```

2、后台商品管理模块

商品管理模块实现商品的 CRUD，本示例只演示新增，删除和更新由于篇幅问题留作练习。

2.1、Action 实现

```
package cn.javass.point.web.admin.action;
//省略 import
public class GoodsAction extends BaseAction {
    public String list() {//列表、展示所有商品（包括未发布的）
        getValueStack().set(PAGE, goodsService.listAll(getPn()));
        return LIST;
    }
    public String doAdd() {//到新增页面
        goods = new GoodsModel();
        getValueStack().set(MODEL, goods);
        return ADD;
    }
    public String add() {//保存新增模型对象
        goodsService.save(goods);
        return REDIRECT;
    }
    //字段驱动数据填充
    private int id = -1;     //前台提交的商品 ID
    private GoodsModel goods; //前台提交的商品模型对象
    //省略字段驱动数据的 getter 和 setter
    //依赖注入 Service
    private IGoodsService goodsService;
    //省略依赖注入的 getter 和 setter
}
```

2.2、Spring 配置文件定义（resources/cn/javass/web/pointShop-admin-servlet.xml）：

```
<bean name="/admin/goodsAction"
      class="cn.javass.point.web.admin.action.GoodsAction" scope="prototype">
    <property name="goodsService" ref="goodsService"/>
</bean>
```

2.3、JSP 实现商品列表页面（WEB-INF/jsp/admin/goods/list.jsp）

查询所有商品，通过迭代“page.items”（Page 对象的 items 属性中存放着分页列表数据）来显示商品列表，在最后应该有分页标签（请参考源代码，示例无），如类似于“<my:page url="${ctx}/admin/goods/list.action"/>”来定义分页元素，。

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html;
charset=UTF-8"%>
<%@ include file="../../common/inc/tld.jsp"%>
<jsp:include page="../../common/inc/header.jsp">
    <jsp:param name="title" value="商品管理-商品列表"/>
</jsp:include>
<a href="${ctx}/admin/goods/doAdd.action">新增</a><br/>
<table border="1">
    <tr>
        <th>ID</th>
        <th>商品名称</th>
        <th>商品描述</th>
        <th>原需积分</th>
        <th>现需积分</th>
        <th>是否已发布</th>
        <th></th>
        <th></th>
        <th></th>
    </tr>
  <s:iterator value="page.items">
    <tr>
        <td><a href="${ctx}/admin/goods/toUpdate.action?id=<s:property
value='id'/>"><s:property value="id"/></a></td>
        <td><s:property value="name"/></td>
        <td><s:property value="description"/></td>
        <td><s:property value="originalPoint"/></td>
        <td><s:property value="nowPoint"/></td>
        <td><s:property value="published"/></td>
```

```
        <td>更新</td> <td>删除</td>
        <td><a href="${ctx}/admin/goodsCode/list.action?goodsId=<s:property
value='id'/>">查看Code码</a></td>
    </tr>
    </s:iterator>
</table>
<jsp:include page="../../common/inc/footer.jsp"/>
```

右击“pointShop”项目选择【Run As】>【Run On Server】启动 Tomcat 服务器，在浏览器中输入“http://localhost:8080/pointShop/admin/goods/list.action”将显示图 11-11 界面。

新增

ID	商品名称	商品描述	原需积分	现需积分	是否已发布			
6	研磨设计模式	一本值得反复阅读的书	200	100	true	更新	删除	查看兑换码
7	研磨设计模式	一本值得反复阅读的书	200	100	true	更新	删除	查看兑换码
8	研磨设计模式	一本值得反复阅读的书	200	100	true	更新	删除	查看兑换码
9	研磨设计模式	一本值得反复阅读的书	200	100	true	更新	删除	查看兑换码

图 11-11 后台商品列表页面

2.4、JSP 实现商品新增页面（WEB-INF/jsp/admin/goods/add.jsp）

表单提交到/admin/goods/add.action 即 cn.javass.point.web.admin.action.GoodsAction 的 add 方法。并将参数绑定到 goods 属性上，在此我们没有进行数据验证，在实际项目中页面中和 Action 中都要进行数据验证。

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html;
charset=UTF-8" %>
<%@ include file="../../common/inc/tld.jsp"%>
<jsp:include page="../../common/inc/header.jsp">
    <jsp:param name="title" value="商品管理-新增"/>
</jsp:include>
<s:fielderror cssStyle="color:red"/>
<s:form action="/admin/goods/add.action" method="POST" acceptcharset="UTF-8" >
<s:token/>
<table border="1">
    <s:hidden name="goods.id" value="%{model.id}"/>
    <s:hidden name="goods.version" value="%{model.version}"/>
    <tr>
        <s:textfield label="商品名称" name="goods.name"
                value="%{model.name}" required="true"/>
    </tr>
```

```
    <tr>
        <s:textarea label="商品简介" name="goods.description"
                value="%{model.description}" required="true" cols="20" rows="3"/>
    </tr>
    <tr>
        <s:textfield label="原需积分" name="goods.originalPoint"
                value="%{model.originalPoint}" required="true"/>
    </tr>
    <tr>
        <s:textfield label="现需积分" name="goods.nowPoint"
                value="%{model.nowPoint}" required="true"/>
    </tr>
    <tr>
        <s:radio label="是否发布" name="goods.published"
                list="#{true:'发布',false:'不发布'}" value="%{model.published}" />
    </tr>
    <tr>
       <td><input name="submit" type="submit" value="新增"/></td>
    <td>
          <input name="cancel" type="button"
           onclick="javascript:window.location.href='${ctx}/admin/goods/list.action'"
       value="取消"/>
    </td>
    </tr>
</table>
</s:form>
<jsp:include page="../../common/inc/footer.jsp"/>
```

右击“pointShop”选择【Run As】>【Run On Server】启动 Tomcat 服务器，在商品列表页面单间【新增】按钮将显示图 11-11 界面。

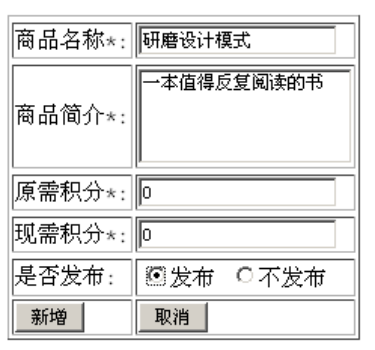


图 11-12 后台商品新增页面

3、后台兑换码管理

提供根据商品 ID 查询兑换码列表及新增兑换码操作，兑换码通过文本框输入多个，使用换行分割。

3.1、Action 实现

```
package cn.javass.point.web.admin.action;
//省略 import
public class GoodsCodeAction extends BaseAction {
    public String list() {
        getValueStack().set(MODEL, goodsService.get(goodsId));
        getValueStack().set(PAGE,
            goodsCodeService.listAllByGoods(getPn(), goodsId));
        return LIST;
    }
    public String doAdd() {
        getValueStack().set(MODEL, goodsService.get(goodsId));
        return ADD;
    }
    public String add() {
        String[] codes = splitCodes();
        goodsCodeService.save(goodsId, codes);
        return list();
    }
    private String[] splitCodes() {//将根据换行分割 code 码
        if(codes == null) {
            return new String[0];
        }
        return codes.split("\r"); //简单起见不考虑“\n”
    }
    //字段驱动数据填充
    private int id = -1;          //前台提交的商品兑换码 ID
    private int goodsId = -1; //前台提交的商品 ID
    private String codes;//前台提交的兑换码，由换行分割
    private GoodsCodeModel goodsCode; //前台提交的商品兑换码模型对象
```

```
    private IGoodsService goodsService;
    //省略依赖注入的 getter 和 setter
}
```

3.2、Spring 配置文件定义（resources/cn/javass/web/pointShop-admin-servlet.xml）：

```
<bean name="/admin/goodsCodeAction"
        class="cn.javass.point.web.admin.action.GoodsCodeAction" scope="prototype">
    <property name="goodsService" ref="goodsService"/>
    <property name="goodsCodeService" ref="goodsCodeService"/>
</bean>
```

3.3、JSP 实现商品兑换码列表页面（WEB-INF/jsp/admin/goodsCode/list.jsp）

商品兑换码列表页面时将展示相应商品的兑换码。

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html;
charset=UTF-8"%>
<%@ include file="../../common/inc/tld.jsp"%>
<jsp:include page="../../common/inc/header.jsp">
    <jsp:param name="title" value="商品管理-商品Code码列表"/>
</jsp:include>
<a href="${ctx}/admin/goodsCode/doAdd.action?goodsId=${model.id}">新增</a>|
<a href="${ctx}/admin/goods/list.action">返回商品列表</a><br/>
<table border="1">
    <tr>
        <th>ID</th>
        <th>所属商品</th>
        <th>兑换码</th>
        <th>购买人</th>
        <th>兑换时间</th>
        <th>是否已经兑换</th>
        <th></th>
    </tr>
    <s:iterator value="page.items">
    <tr>
        <td><s:property value="id"/></td>
```

```
            <td><s:property value="exchanged"/></td>
            <td>删除</td>
        </tr>
        </s:iterator>
</table>
<jsp:include page="../../common/inc/footer.jsp"/>
```

右击“pointShop”选择【Run As】>【Run On Server】启动 Web 服务器，在浏览器中输入“http://localhost:8080/pointShop/admin/goods/list.action”，然后在指定商品后边点击【查看兑换码】将显示图 11-12 界面。

新增 | 返回商品列表

ID	所属商品	兑换码	购买人	兑换时间	是否已经兑换	
1	研磨设计模式	12234443232			false	删除
2	研磨设计模式	342342342342			false	删除
3	研磨设计模式	3423424234234			false	删除

图 11-12 商品兑换码列表

3.4、JSP 实现商品兑换码新增页面（WEB-INF/jsp/admin/goodsCode/add.jsp）

用于新增指定商品的兑换码。

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html;
charset=UTF-8"%>
<%@ include file="../../common/inc/tld.jsp"%>
<jsp:include page="../../common/inc/header.jsp">
    <jsp:param name="title" value="用户管理-新增"/>
</jsp:include>
<s:fielderror cssStyle="color:red"/>
<s:form action="/admin/goodsCode/add.action" method="POST"
acceptcharset="UTF-8">
<s:token/>
<s:hidden name="goodsId" value="%{model.id}" />
<table border="1">
    <tr>
        <s:textfield label="所属商品" name="model.name" readonly="true"/>
    </tr>
```

```
    <tr>
        <td><input name="submit" type="submit" value="新增"/></td>
        <td><input name="cancel" type="button"
onclick="javascript:window.location.href='${ctx}/admin/goodsCode/list.action?goodsId=
<s:property value='%{model.id}'/>'" value="取消"/></td>
    </tr>
</table>
</s:form>
<jsp:include page="../../common/inc/footer.jsp"/>
```

右击“pointShop”选择【Run As】>【Run On Server】启动 Tomcat 服务器，在商品兑换码列表中单击【新增】按钮将显示图 11-13 界面。

图 11-13 兑换码新增页面

4、前台商品展示及购买模块：

前台商品展示提供商品展示及购买页面，购买时应考虑是否有足够兑换码等，此处错误消息使用硬编码，应该考虑使用国际化支持，请参考【】学习国际化。

4.1、Action 实现

```
package cn.javass.point.web.front.action;
//省略 import
public class GoodsAction extends BaseAction {
    private static final String BUY_RESULT = "buyResult";
    public String list() {
        getValueStack().set(PAGE, goodsService.listAllPublished(getPn()));
        return LIST;
    }
    public String buy() {
        String username = "test";
        GoodsCodeModel goodsCode = null;
```

```
            this.addActionError("没有足够的兑换码了");
            return BUY_RESULT;
        } catch (Exception e) {
            e.printStackTrace();
            this.addActionError("未知错误");
            return BUY_RESULT;
        }
        this.addActionMessage("购买成功，您的兑换码为 :"+ goodsCode.getCode());
        getValueStack().set(MODEL, goodsCode);
        return BUY_RESULT;
    }
    //字段驱动数据填充
    private int goodsId;
    //省略字段驱动数据的 getter 和 setter
    //依赖注入 Service
    IGoodsService goodsService;
    IGoodsCodeService goodsCodeService;
    //省略依赖注入的 getter 和 setter
}
```

4.2、**Spring 配置文件定义（resources/cn/javass/web/pointShop-front-servlet.xml）：**

```
<bean name="/front/goodsAction"
      class="cn.javass.point.web.front.action.GoodsAction" scope="prototype">
    <property name="goodsService" ref="goodsService"/>
    <property name="goodsCodeService" ref="goodsCodeService"/>
</bean>
```

4.3、JSP 实现前台商品展示及购买页面（WEB-INF/jsp/ goods/list.jsp）

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html;
charset=UTF-8"%>
<%@ include file="../../common/inc/tld.jsp"%>
<jsp:include page="../../common/inc/header.jsp">
    <jsp:param name="title" value="积分商城-商品列表"/>
</jsp:include>
```

```
    描述：<s:property value="description"/><br/>
    <S>需要积分<s:property value="originalPoint"/></S>  现需积分：
<B><s:property value="nowPoint"/></B><br/>
</s:iterator>
<jsp:include page="../../common/inc/footer.jsp"/>
```

右击“pointShop”选择【Run As】>【Run On Server】启动 Web 服务器，在浏览器中输入 **http://localhost:8080/pointShop/goods/list.action** 将显示图 11-14 界面。

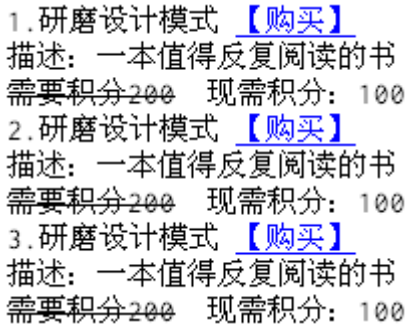


图 11-14 前台商品展示即购买页面

在前台商品展示即购买页面中点击购买，如果库存中还有兑换码，将购买成功，否则购买失败。

4.3、商品购买结果页面（WEB-INF/jsp/admin/goods/buyResult.jsp）

购买成功将通过“<s:actionmessage/>”标签显示成功信息并将兑换码显示给用户，购买失败将通过“<s:actionerror/>”标签提示如积分不足或兑换码没有了等错误信息。

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html;
charset=UTF-8"%>
<%@ include file="../../common/inc/tld.jsp"%>
<jsp:include page="../../common/inc/header.jsp">
    <jsp:param name="title" value="积分商城-购买结果"/>
</jsp:include>
<s:actionerror/>
<s:actionmessage/>
<jsp:include page="../../common/inc/footer.jsp"/>
```

在商品展示及购买列表购买成功或失败将显示图 11-15 或图 11-16 界面。

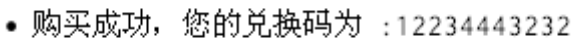


图 11-15 购买成功页面

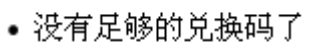


图 11-16 购买失败页面

到此 SSH 集成已经结束，集成 SSH 是非常简单的，但开发流程及开发思想是关键。

我们整个开发过程是首先抽象和提取通用的模块和代码，这样可以复用减少开发时间，其次是基于通用层开发不可预测部分（即可变部分），因为每个项目的功能是不一样的。在开发过程中还集中将重复内容提取到一处这样方便以后修改。