# SSH

# 1. Struts2

## 1.1. 原理

### 1.1.1. 1) 客户端（Client）向Action发用一个请求（Request）

2） Container通过web.xml映射请求，并获得控制器（Controller）的名字

3）

容器（Container）调用控制器（StrutsPrepareAndExecuteFilter或FilterDispatcher）,Struts2.1以前调用FilterDispatcher，Struts2.1以后调用StrutsPrepareAndExecuteFilter

4） 控制器（Controller）通过ActionMapper获得Action的信息

5） 控制器（Controller）调用ActionProxy

6） ActionProxy读取struts.xml文件获取action和interceptor stack的信息。

7） ActionProxy把request请求传递给ActionInvocation

8） ActionInvocation依次调用action和interceptor

9） 根据action的配置信息，产生result

10） Result信息返回给ActionInvocation

11） 产生一个HttpServletResponse响应

12） 产生的响应行为发送给客服端。

## 1.2. 请求参数注入的三种方式

### 1.2.1. 属性驱动

在Struts2框架中，表单的提交的数据会自动注入到与Action对象相对应的属性。它与Spring框架中的IoC的注入原理相同，通过Action对象为属性提供setter方法注入

在Action中注入的属性的属性名与表单对应控件的name值相等，且该属性提供getXXX()、setXXX()方法

### 1.2.2. 域驱动DomainDriven

操作领域对象的方法，在Action对象中引用某一实体对象，并且http请求的参数值可以注入到实体对象的属性上(jsp中的表单空间的name属性应该与Action中的对象和bean中的属性对应，格式为 对象.属性 ，例如：user.name)

### 1.2.3. 驱动模型ModelDriven

Struts2的API中，提供了一个名为ModelDriven的接口，Action对象可以通过实现此接口获取指定的实体对象，获取的方式是实现ModelDriven提供的getModel()方法进行获取，其语法格式为： T getModel();：驱动模型必须实例化对象类型的那个成员变量，否则空指针错误

## 1.3. 访问Action中的指定方法

### 1.3.1. 单独为每个方法在xml里面配置Action

### 1.3.2. 动态方法调用

```
<!--启用动态方法调用-->
  <constant name="struts.enable.DynamicMethodInvocation" value="true" />
```

在URL 中 附加 !方法名  来动态调用不同的方法

### 1.3.3. 通配符

如果是2.5或更高版本,请在配置文件的包元素中开启通配符的使用

```
<global-allowed-methods>
      regex:.*
</global-allowed-methods>
```

或者开启SMI

```
<strict-method-invocation="false">
```

action name="test_*" method="{1}"

URL: test_login 即为调用 该Action中的 login 方法;

#### 1.3.4. 直接指定方法名

Action中的 method="login"直接指定了执行login方法,默认执行execute方法

### 1.4. 重定向和转发

#### 1.4.1. dispatcher(默认)

```
<result name="success" type="dispatcher">
  <param name="location">/index.jsp</param>
</result>
```

#### 1.4.2. redirect

重定向到jsp、action、外部网址

```
<result name="success" type="redirect">/index.jsp</result>
<result name="success" type="redirect">/login.do</result>
<result name="success" type="redirect">http://www.baidu.com</result>
```

#### 1.4.3. chain

用于action跳转

```
<action name="action1" class="org.Action1">
  <result name="success" type="chain">action2.do</result>
</action>
<action name="action2" class="org.Action2">
  <result name="success">login.jsp</result>
</action>
```

#### 1.4.4. redirect-action

重定向到另外一个action

```
<result name="success" type="redirect-action">
  <param name="actionName">login.do</param> 重定向action名
  <param name="userId">userId</param>带的参数
</result>
```

1.4.5. 区别:

1.跳转内容dispatcher和redirect跳转的是views（一般是jsp页面）；chain和redirectAction跳转的是一个action。

2.跳转方式dispatcher和chain是服务器端跳转，所以客户端只发起一次请求，产生一个action；redirect和redirectAction是客户端跳转，所以客户端发起两次请求。

1.5. 相对路径/绝对路径

1.5.1. 绝对路径(带有访问协议的路径,比如Http;file:///)

本地绝对路径: file:///

网络绝对路径 :http

1.5.2. 相对路径

以斜杠开头的

前台相对路径

由浏览器解析执行的代码所包含的路径;例如,html,css,js中的路径,以及jsp中静态部分的路径.像html,jsp静态部分中的<img src=""/> <a href=""/> <form action=""/>等;还有css中的img("");js中的window.location.href=""等

前台相对路径的参照路径是web服务器的根路径,即http://127.0.0.1:8080/

后台相对路径

由服务器解析执行的代码及文件中包含的路径.例如,java代码中的路径/jsp文件动态部分的路径/xml文件中的路径(xml文件是要被java代码加载入内存,并由java代码解析的)等

后台相对路径的参照路径是web应用的要路径.如http://127.0.0.1:8080/demo(项目名称)/

demo

```
<!-- 下面的相对路径是带斜杠的前台相对路径，其转变为绝对路径后是
     http://127.0.0.1:8080/test/login.action
     -->
<!--
<form action="/test/login.action" method="post">
    姓名：<input type="text" name="name"/><br>
    年龄：<input type="text" name="age"/><br>
    <input type="submit" value="登录"/>
</form>
```

```
<!-- 要想访问下面的package中定义的action，则需要提交请求的绝对路径应该是：
    http://127.0.0.1:8080/01-primary/test/login.action -->
<package name="demo" namespace="/test" extends="struts-default">
    <action name="login" class="com.bjpowernode.actions.LoginAction">
        <result name="success">/welcome.jsp</result>
    </action>
</package>
```

```
<!-- 下面的相对路径是不带斜杠的相对路径，其参照路径是当前的访问路径，
     而当前的访问路径是：http://127.0.0.1:8080/01-primary/
     所以下面的相对路径转变为绝对路径后，就变为了：
     http://127.0.0.1:8080/01-primary/test/login.acton
 -->
<form action="test/login.action" method="post">
    姓名：<input type="text" name="name"/><br>
    年龄：<input type="text" name="age"/><br>
    <input type="submit" value="登录"/>
</form>
```

不带斜杠开头的

无论是前台相对路径还是后台相对路径,其参考路径都是当前资源的访问路径,而不是当前资源的保存路径(URL中最后一个/之前的为当前资源的访问路径(请求路径),之后的叫作资源名称.)

### 1.5.3. 常见问题

问题:

假设index.jsp中的form表单提交完之后验证失败再次返回到index.jsp(使用action="test/login.action") ,多次失败之后地址栏出现很多个 ...../test/test/.....

原因是:

不带斜杠的相对路径是以当前资源的访问路径为参考,而不是以项目中该index.jsp的路径为参考,所以再验证失败一次之后,项目的当前资源的访问路径发生了变化.

解决: 1.使用base标签: 在jsp的

<head> <base href="<%=basePath%>"> </head>

jsp会自动把base添加到相对路径前面(实质上是把相对路径转化为绝对路径)

2.使用带斜杠开头的相对路径,

action="${pageContext.request.contextPath}/test/login.action"


```
<%
String path = request.getContextPath();
String basePath = request.getScheme()+"://"+request.getServerName()+":"+request.getServerPort()+path+"/";
%>
```

### 1.6. Struts2类型转换

#### 1.6.1. Struts2内默认转换器

其支持的从String类型转换的目标类型如下：

boolean和Boolean：字符串true会转换为布尔类型值true

char和Character：字符串转字符

int和Integer：字符串转整型类型

long和Long：字符串转长整型

float和Float：字符串转单精度浮点型

double和Double：字符串转双精度浮点型

Date：字符串转日期类型，需要字符串满足一定的格式

数组：多个input表单提交给同一个Action的属性，就会构成一个数组传入到该属性中

集合：和数组类似，需要指定了的类型，并且类型不能超出基本数据类型

#### 1.6.2. 基于OGNL的类型转换

对于自定义对象:

就像域驱动一样,需要在jsp页面传递参数的时候指定对象名称

```
<s:textfield name="uesr.username" label="用户名"/>
<s:textfield name="user.age" label="年龄"/>
```

对于list和map集合

```
<s:textfield name="list[0].username" label="用户名"/>
    <s:textfield name="list[0].age" label="年龄"/>
    <s:textfield name="list[1].username" label="用户名"/>
    <s:textfield name="list[1].age" label="年龄"/>
```

list[0].username表示为Action属性list的第一个元素的username传值

```
<s:textfield name="map['1'].username" label="用户名"/>
    <s:textfield name="map['1'].age" label="年龄"/>
    <s:textfield name="map['2'].username" label="用户名"/>
    <s:textfield name="map['2'].age" label="年龄"/>
```

map[‘1’].username表示为Action实例的map属性添加一条信息：key为1，key为1的value值为user的username属性的值为该文本框的值

### 1.6.3. 自定义类型转换

通过重写convertValue的任意一个重载来完成自定义类型转换器:

原理:

首先struts2接到参数，假设得到的参数是Point类型的p=5,8。接下来struts2发现并没有这种东西，转换不成Point,struts2就会去找action对应的包下面有没有一个对应的转换文件，会去找转换文件里面的成员变量里有没有对应的转换器，如果找到一个，就会尝试把接收到的参数传给转换器“转换方法”中的形参value，把要转换的类型传给toType。最后“转换方法”执行得到一个结果Object，结果就是一个new出来的Point

注册类型转换器

1在局部范围内注册一个类型转换器(和Action同包):

ActionName-conversion.properties

文件内容:// 属性名=转换器类的位置

2在全局范围内注册一个类型转换器

xwork-convertion.properties

3使用注解注册一个类型转换器

## 1.7. File Upload

### 1.7.1. 实现步骤

第一步：引入第三方jar包

第二步：JSP页面的form表单中添加个enctype属性，并设置属性值为：multipart/form-data；

input标签：<input type="file" name="uploadImage"/>

第三步：在Action类中添加属性：

private File uploadImage；--得到上传的文件

private String uploadImageContentType；--得到文件的类型

private String uploadImageFileName；--得到文件的名

--并设置这三个属性的setter和getter方法

第四步：编写upload方法（方法名自定义）

（1）：String realpath = ServletActionContext.getServletContext().getRealPath("/images"); --获得文件上传后保存的路径

（2）：File file = new File(realpath)；--根据路径名创建File实例

（3）：--使用org.apache.commons.io.FileUtils工具类实现文件上传

FileUtils.copyFile(uploadImage,new File(file,uploadImageFileName));

### 1.7.2. 多文件上传

在进行多文件上传时，同样可以导入第三方jar包，包括form表单的写法都和上边文件上传相同。不同点是Action类中的写法：三个属性全部改成数组形式的，并编写setter和getter方法。再使用FileUtils工具类进行上传时，只需要循环调用copyFile方法即可，示例：

```
for(int i=0 ;i<uploadImages.length; i++){
   File uploadImage = uploadImages[i];
   FileUtils.copyFile(uploadImage, new File(file, uploadImagesFileName[i]));

}
```

## 1.8. 拦截器（interceptor）

### 1.8.1. 什么是拦截器？

拦截器，在AOP（面向切面编程）中用于在某个方法或字段被访问之前，进行拦截，之后可实现被访问之前或之后，加入某些操作。如：权限的校验。在struts中，拦截器是动态的拦截Action调用的对象，可以实现在一个action执行前后需要另外执行的代码。通俗的讲，拦截器起到拦截action的作用。

### 1.8.2. 拦截器和过滤器

Filter：过滤器，过滤从客户端到服务器发送的请求。

Interceptor：拦截器，拦截的是客户端对action的访问，是更加细化的拦截。struts2的核心都是通过拦截器实现的。

客户端向服务器发送一个action请求，执行核心过滤器（doFilter）方法。在这个方法中，executeAction()方法，在这个方法内部调用dispatcher.servlceAction()；在这个方法内部创建一个action的代理，最终执行的是代理中的execut

e方法，在代理中的execute()方法调用ActionInvocation的invoke方法。在这个方法中递归执行一组拦截器（完成相应的功能），如果没有下一个拦截器，就会执行目标action，根据目标action的返回结果跳转到相应的页面

### 1.8.3. 拦截器的执行流程

客户端向服务器发送一个action请求，执行核心过滤器（doFilter）方法。在这个方法中，executeAction()方法，在这个方法内部调用dispatcher.servlceAction()；在这个方法内部创建一个action的代理，最终执行的是代理中的execute方法，在代理中的execute()方法调用ActionInvocation的invoke方法。在这个方法中递归执行一组拦截器（完成相应的功能），如果没有下一个拦截器，就会执行目标action，根据目标action的返回结果跳转到相应的页面

### 1.8.4. Interceptor接口

**void init():**

该方法在拦截器被调用时立即被执行，它在拦截器的生命周期内只被调用一次，可以在该方法中对相关资源进行初始化配置。

**void destory():**

此方法和init方法相对应，在拦截器实例被销毁前调用，来释放和拦截器相关的资源。在拦截器的声明周期内也是执行一次。

**String intercept(ActionInvocation invocation)：**

拦截器的核心方法。在此方法中编写真正执行拦截功能的代码，实现具体的拦截操作。它返回的一个字符串类型的数据作为逻辑视图，struts2的配置文件中的<result name="

">name对应的名字。每拦截一个动作请求，该方法就被执行一次。

### 1.8.5. 拦截器的配置

```
<!-- 定义拦截器 -->
<interceptors>
```

```
<!-- name属性用来指定拦截器的名称，class属性用来指定拦截器的实现类 -->
<interceptor name="interceptorDemo1"
class="com.ebao.interceptor.InterceptorDemo1"/>
<!--
demo2中的param标签用来指定参数，其中paramName为参数名，paramValu
e为参数值 -->
<interceptor name="interceptorDemo2"
class="com.ebao.interceptor.InterceptorDemo2">
<param name="paramName">paramValue</param>
</interceptor>
<//interceptor>
<action name="userAction_*" class="com.ebao.shop.action.UserAction"
method="{1}">
<result name="login">/login.jsp</result>
<!--
action中引用拦截器（一旦引入自定义拦截器，struts2的默认拦截器栈中的拦
截器就不执行了，不过仍需要其中的拦截器，需要再次引入） -->
<interceptor-ref name="defaultStack"/>
<interceptor-ref name="interceptorDemo1"/>
<interceptor-ref name="interceptorDemo2"/>
</action>
```

定义拦截器栈

概述：

实际开发中，经常在Action执行前同时执行多个拦截器，这时，可以将多个拦截器组成一个拦截器栈。使用时可以将栈中的多个拦截器当成一个整体栏引用。

```
<!-- 拦截器栈的定义如下 -->
<interceptor-stack name="myInterceptorStack">
<!-- 默认栈的引用 -->
```

```
<interceptor-ref name="defaultStack"/>
<!-- 自定义栈的引用 -->
<interceptor-ref name="interceptorDemo1"/>
<interceptor-ref name="interceptorDemo2"/>
</interceptor-stack>
<!--
自定义拦截器栈的引用，自定义栈中一般含默认栈的引用了，无需再次引用 -->
<interceptor-ref name="defaultStack"/>
```

## 1.9. OGNL

### 1.9.1. 基本语法

它也是一种表达式语言 - 通常用于表单输入字段名称和JSP标记。

OGNL表达式用于将Java端数据属性绑定到基于文本的视图层中的字符串。

OGNL

表达式都基于当前对象的上下文来完成求值运算，链的前面部分的结果将作为后面求值的上下文

例如:

```
ame.toCharArray()[0].numericValue.toString()
```

上述表达式的求值步骤：

1提取根 (root) 对象的 name 属性。

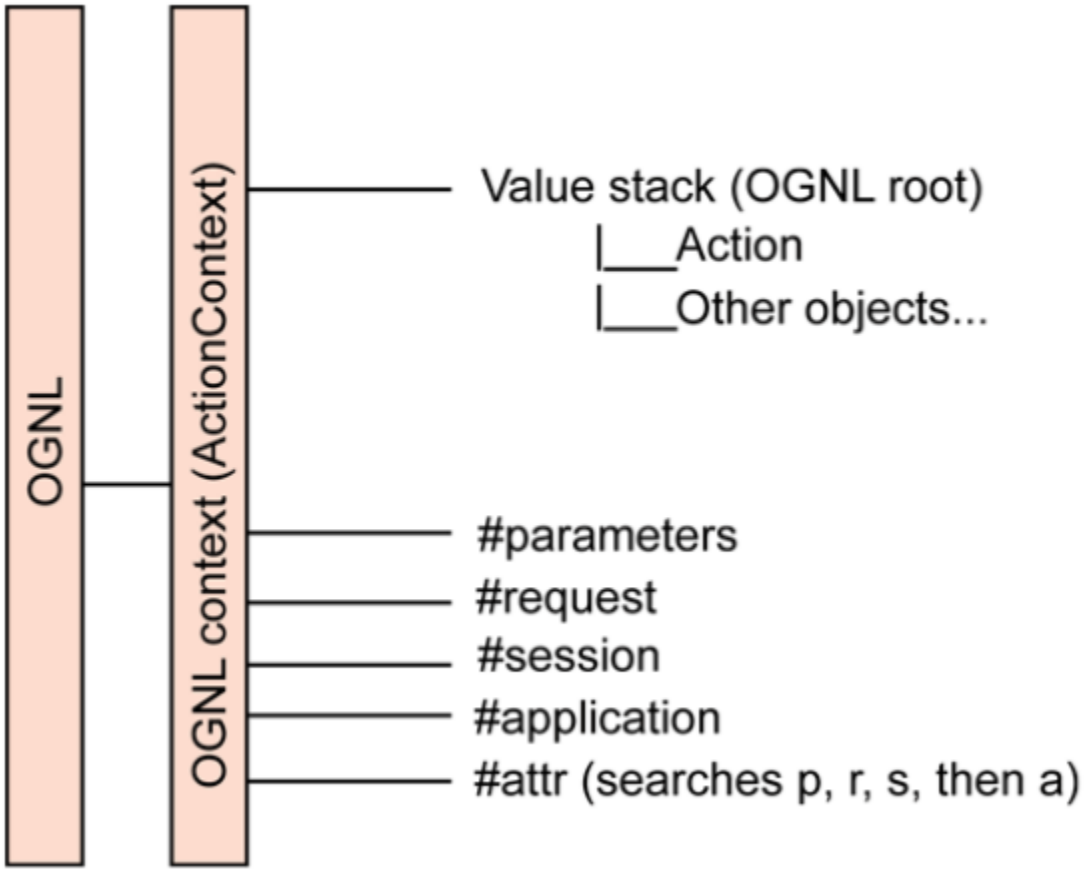
2调用上一步返回的结果字符串的 toCharArray() 方法。

3提取返回的结果数组的第一个字符。

4获取字符的 numericValue 属性，该字符是一个 Character 对象，Character 类有一个 getNumericValue() 方法。

调用结果 Integer 对象的 toString() 方法。

### 1.9.2. Context




**ValueStack原理**

**ValueStack允许推入多个bean(即为Action)，并根据它来评估动态EL表达式。在计算表达式时，将从堆栈中向下搜索堆栈，从最新的对象推送到最早的对象，查找具有给定属性的getter或setter的bean或给定名称的方法（取决于表达式是评估）**

### 1.9.3. OGNL访问静态成员

**struts2**

**中默认提供了一些访问静态成员的方式，但是默认是关闭的，所以应该在struts2的配置文件中先设置**

**<constant name="struts.ognl.allowStaticMethodAccess" value="true"/>**

1 访问静态方法

<s:property value="@com.netsdar.song.manager.util.AppUtil@getName()"/>

2 访问静态常量

<s:property value="@com.netsdar.song.manager.util.AppUtil@Index_Title"/>

### 1.9.4. 问题

题一：#,%{},$符号

在Struts2标签属性中经常会出现"#"或者"%{}"的符号出现，OGNL上下文中有且仅有一个根对象。Struts2为我们定义了许多明明对象，他们分别是"ValueStack"，"Parameters"，"Session"，"Request"，"Appliction"，"Attr"，其中"ValueStack"被设置为上下文的根对象。访问非根对象必须加上"#"号，这就是出现"#"的原因。Struts2中的标的处理类，并不是所有都将标签的属性作为OGNL表达式来看待，有时候我们需要设置动态地值，则必须告诉标签的处理类该字符串按照OGNL表达式来处理，%{}符号的作用就是告诉标签的处理类将它包含的字符串按照OGNL表达式处理。

"$"符号用于XML文件中用于获取动态值，与%{}作用类似。

问题二：%{}符号的影响

Struts2的标签几十几百个，要记住哪一个标签的处理类将标签的属性作为OGNL表达式是一件很困难的事情，在不清楚处理类的处理方式时怎么办，%{}对于标签处理类来说，若处理类将属性值作为普通字符串则%{}符号包含的字符串当做OGNL表达式，若处理类将属性值作为OGNL表达式来处理，则直接忽略%{}符号。换句话说，不清楚处理方式的话，可以都使用%{}符号。

例如:

OGNL表达式"username"表示了从根对象ValueStack中取出属性username的值。它会从栈顶到栈底遍历ValueStack，直到找某一个Action中的"username"属性。

## 1.10. Struts Tag

### 1.10.1. 条件标签:用于执行基本的条件流转

```
<s:if test="#request.username=='ebao'">test</s:if>
<selseif test="#request.username=='class'">test0</s:elseif>
```

### 1.10.2. 迭代标签：用于遍历集合（java.util.Collection）或者枚举值（java.util.Iterator）类型的对象，value属性表示集合或枚举对象

```
<s:iterator value="userList" status="user">
姓名：<s:property value="user.userName" />
年龄：<s:property value="user.age" />
</s:iterator>
```

### 1.10.3. 属性标签：用以输出value属性的值，并拥有一个default属性,在value对象不存在时显示。escape属性为true,来输出原始的HTML文本

```
<s:property value="getText('some.key')">
```

## 1.11. Ajax

### 1.11.1. 必要的包:

Struts2 Json

6个jar包，缺一不可。

commons-lang.jar；

json-lib-2.3-jdk15.jar；

struts2-json-plugin-2.2.3.jar；

ezmorph-1.0.1.jar；

commons-beanutils-1.9.2.jar；

commons-collections-3.1.jar

这个依赖必须要指定jdk版本<classifier>jdk15</classifier>

```
<dependency>
   <groupId>net.sf.json-lib</groupId>
   <artifactId>json-lib</artifactId>
   <version>2.4</version>
   <classifier>jdk15</classifier>
</dependency>
```

### 1.11.2. 要求

1.需要传递到Action 的数据:

```
data : {"id":2},  // 要传递的数据
```

2.在action中用属性模型的方式即可获取id值

3.在struts中

A:package 继承 json-default

B: <result name="selGoodsByIdAjax" type="json"> //一定要写返回类型 type="json"

```
        <param name="root">result</param> //固定写法,name="root"
    </result>
```

4. JSONObject json=JSONObject.fromObject(goods);

System.out.println(json+"这是json对象");

this.result = json.toString();//需要把json对象转化为字符串传到前台页面

5.在前台页面取到result之后 需要用 var obj = eval("("+result+")");//对该数据进行格式化 {"name":"value"}

timestamp 转化为json类型数据后如下,需要在js中修改其格式显示在前台页面

```
"createTime":{"date":8,"hours":14,"seconds":36,"month":7,"nanos":803000000
,"timezoneOffset":-
480,"year":118,"minutes":58,"time":1533711516803,"day":3}
```

```
function
json2TimeStamp(milliseconds){//需要传递过来一个Timestamp的毫秒数:nanos
    var datetime = new Date();
    datetime.setTime(milliseconds);
    var year=datetime.getFullYear();
//月份重0开始，所以要加1，当小于10月时，为了显示2位的月份，所以补0
        var month = datetime.getMonth() + 1 < 10 ? "0" + (datetime.getMonth() +
1) : datetime.getMonth() + 1;
    var date = datetime.getDate() < 10 ? "0" + datetime.getDate() :
datetime.getDate();
    var hour = datetime.getHours()< 10 ? "0" + datetime.getHours() :
datetime.getHours();
    var minute = datetime.getMinutes()< 10 ? "0" + datetime.getMinutes()
: datetime.getMinutes();
    var second = datetime.getSeconds()< 10 ? "0" + datetime.getSeconds() :
datetime.getSeconds();
    return year + "-" + month + "-" + date+"
"+hour+":"+minute+":"+second;
    }
```

### 1.11.3. DEMO

Action

```
private String result;
private User user;
public String ajax() {
    User u = new User();
    u.setId(111);
    u.setUserName("aaa");
    this.setUser(u);
    JSONObject json=JSONObject.fromObject(user);
```

```
        System.out.println(json+"这是json对象");
        this.result = json.toString();
        return "Ajax";
      }
```

struts.xml

```
  <package name="json" extends="json-default">
        <action name="SelectId" class="com.struts.action.UserJsonAction"
        method = "ajax">
          <result name="Ajax" type="json">
             <param name="root">result</param>
          </result>
        </action>
  </package>
```

JSP

```
  function get(){
        var id1 = null;
        //var params=$("a").serialize();//序列化要传递的数据
        $.ajax({
          url : "SelectId",//请求地址
          dataType : "json",//数据格式
          type : "post",//请求方式
          async : false,//是否异步请求
          //data : params,  // 要传递的数据
          success : function(result)
          {  //如何发送成功
            alert(result);
            var obj = eval("("+result+")");//转化为Json数据格式
```

```
            id1 = obj.id;
            alert(obj.id);
          }});
      return id1;
    }
```

### 1.12. Restful

#### 1.12.1. 网络上的所有事物都可被抽象为资源（Resource）。

每个资源都有一个唯一的资源标识符（Resource Identifier）。

同一资源具有多种表现形式。

使用标准方法操作资源。

通过缓存来提高性能。

对资源的各种操作不会改变资源标识符。

所有的操作都是无状态的（Stateless）

#### 1.12.2. REST 系统由使用 URI（Uniform Resource Identifier，即统一资源标识符）命名的资源组成。由于 REST 对资源使用了基于 URI 的统一命名，因此这些信息就自然地暴露出来了，从而避免“信息地窖”的不良后果。与 URI 相关的概念还有 URL，URL 是 Uniform Resource Locator，也就是统一资源定位符的意思。其中 http://www.crazyit.org 就是一个统一资源定位符，URL 是 URI 的子集。简而言之：每个 URL 都是 URI，但不是每个 URI 都是 URL。

## 2. Hibernate

### 2.1. *.hbm.xml

#### 2.1.1. 映射文件（xxx.hbm.xml）用来把PO与数据库中的数据表、PO之间的关系与数据表之间的关系，以及PO的属性与表字段一一映射起来，它是Hibernate的核心文件。

PO:Persistent

Objects，即持久化对象，它可以是普通的JavaBean，惟一特殊的是它们与（仅一个）Session相关联。JavaBean在Hibernate中存在三种状态：临时状态（transient）、持久化状态（persistent）和脱管状态（detached）。当一个JavaBean对象在内存中孤立存在不与数据库中的数据有任何关联关系时，那么这个JavaBean对象就称为临时对象（TransientObject）；当它与一个Session相关联时，就变成持久化对象（Persistent

Object）；在这个Session被关闭的同时，这个对象也会脱离持久化状态，变成脱管对象（Detached

Object），可以被应用程序的任何层自由使用，例如，可用做与表示层打交道的数据传输对象（Data transfer Object）。

2.1.2. 1 类和表的映射

class标签有一般用到两个属性，一个是name，一个是table，分别对应的是类名和数据表名.还有一个schema属性,用来指定数据表所属用户

2.1.3. 2 主键的映射

id标签中的name属性对应的是实体类中唯一标示id，column属性对应数据表中的主键id，length可设置id字段的长度。

generator标签设置主键生成策略。

1.assigned:

<generator class="assigned" />

主键由外部程序负责生成;特点:可以跨数据库，人为控制主键生成，应尽量避免

2.increment

<generator class="increment" />

由Hibernate从数据库中取出主键的最大值（每个session只取1次），以该

值为基础，每次增量为1，在内存中生成主键，不依赖于底层的数据库，因此可以跨数据库。特点：跨数据库，不适合多进程并发更新数据库，适合单一进程访问数据库，不能用于群集环境。

**3.hilo**

```
<generator class="hilo">
  <param name="table">hibernate_hilo</param>指定保存hi值的表名
  <param name="column">next_hi</param>指定保存hi值的列名
  <param name="max_lo">100</param> 指定低位的最大值
</generator>
```

hilo（高低位方式high low）是hibernate中最常用的一种生成方式，需要一张额外的表保存hi的值。保存hi值的表至少有一条记录（只与第一条记录有关），否则会出现错误。可以跨数据库。

**4.native**

```
  <generator class="assigned" />//或者自己指定seq
  <generator class="native"> <param name="sequence">T_SEQ</param>
</generator>
```

hibernate 将根据底层数据库适配器dialect的定义,对于oracle自动采用Sequence生成主键;在这种情形下，hibernate会默认使用名为hibernate_sequence的序列

**5.sequence**

```
  <param name="sequence">T_SEQ</param>
```

**6.uuid**

```
  <generator class="uuid" />
```

Hibernate在保存对象时，生成一个UUID字符串作为主键，保证了唯一性，但其并无任何业务逻辑意义，只能作为主键

### 2.1.4. 3.属性和字段的映射

property标签中，name对应实体类中的属性，

type对应的是该属性在数据库中字段（列）的类型，如果不填，默认为属性name对应属性的类型，如：数据库表字段名、长度、类型都和属性password保持一致，

not-null对应数据库中的不为空，非空约束，

unique对应的是唯一性，唯一约束，

column对应的是数据库表对应的字段名，如果不填，该字段默认为属性name对应属性的名字，如：该字段名是password，是birthday，是salary。

### 2.1.5. 4.关系的映射

一对一

使用one-to-one标签

```
<one-to-one name="idCard" constrained="true"/>
  <!--constrained="true"表示person引用了idCard的主键作为外键-->
    <!-- one-to-one指示hibernate如何加载其关联对象，默认根据主键加载
      也就是拿到关系字段值，根据对端的主键来加载关联对象
    -->
```

也可以在在many-to-one标签中，设置unique：标识唯一性，这样many-to-one就变成了唯一的。

一对多(一个部门多个员工为例(Emp,Dept))

Dept:

```
<set name="emps" table="EMP" insverse="true" lazy="true">
  <key column="DEPTID"></key>
  <one-to-many class="com.entity.Emp">
</set>
```

name对应部门dept关联关系集合的属性名，table对应关联集合元素对应的表名，inverse为true表示放弃维护关系能力，lazy表示控制关联数据延迟加

**载，key标签中column对应关联表（t_emp）指向本表的外键字段名，one-to-many标签中的class对应关联集合中元素的类型。**

**EMP:**

```
<many-to-one name="dept" class="com.entity.Dept">
  <column name="DEPTID">
</many-to-one>
```

**many-to-one标签中name对应的是关联关系的属性名，class对应关联属性类型，column：本实体类（Emp）对应表（t_emp）中指向关联表（t_dept）中的外键字段名。**

**多对多(多个老师Teacher对应多个学生Student)**

**数据库模型要尽量符合第三范式,即属性不能传递依赖于主属性(,如果某一属性依赖于其他非主键属性，而其他非主键属性又依赖于主键，那么这个属性就是间接依赖于主键，这被称作传递依赖于主属性。)**

**实现多对多的关系需要通过建立中间表来实现多对多关系**

**Student**

```
<set name="teas" table="t_stu_tea" fetch="join">
  <key><column name="stu_id"  /></key>
  <many-to-many column="tea_id" class="com.entity.Teacher"></many-to-many>
</set>
```

**set标签表名当前实体对应多个数据的关系，其中name对应关联关系的属性名，table是对应关系的表名，也就是中间表，fetch为join表示左外连接，key标签中的column是当前表（t_student）在中间表中的外键字段名，many-to-many中的column属性是关联表（t_teacher）在中间表中的外键字段名，class对应关联属性类型.**

**Teacher**

```
<set name="stus" table="t_stu_tea" inverse="true">
  <key ><column name="tea_id"  /></key>
  <many-to-many column="stu_id" class="com.entity.Student"></many-to-many>
</set>
```

**set标签表示当前实体对应多个数据的关系，其中name对应关联关系的属性名，table是对应关系的表名，也就是中间表，inverse为true表示放弃维护关系能力，key标签中的column是当前表（t_teacher）在中间表的外键字段名，many-to-many中的column属性是关联表（t_student）在中间表的外键字段名，class对应关联属性类型。**

## 2.2. hibernate.cfg.xml

### 2.2.1. <!-- 连接数据库的基本参数 -->

```
<property name="hibernate.connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
<property name="hibernate.connection.url">jdbc:oracle:thin:@localhost:1521:orcl</property>
<property name="hibernate.connection.username">C##SHOP</property>
<property name="hibernate.connection.password">123456</property>
<!-- 配置Hibernate的方言 -->
<property name="hibernate.dialect">org.hibernate.dialect.Oracle12cDialect</property>
<!-- 可选配置================ -->
<!-- 打印SQL -->
<property name="hibernate.show_sql">true</property>
```

```
<!-- 格式化SQL -->
<property name="hibernate.format_sql">true</property>
<!-- 自动建表 -->
<property name="hibernate.hbm2ddl.auto">update</property>

<!-- 注册mapping -->
<mapping resource="com/struts/model/Goods.hbm.xml" />
```

#### 2.2.2. C3P0

```
<!--C3P0 数据源(数据库连接池) -->
  <property name="hibernate.connection.provider_class">
    org.hibernate.c3p0.internal.C3P0ConnectionProvider
  </property>
```

### 2.3. Hibernate的操作

#### 2.3.1. session获取

```
Hibernate5
private static SessionFactory sessionFactory;
  static SessionFactory buildSessionFactory(){
    StandardServiceRegistry ssr =
        new StandardServiceRegistryBuilder().configure().build();
    sessionFactory =
        new MetadataSources(ssr).buildMetadata().buildSessionFactory();
    return sessionFactory;
  }
  public static SessionFactory getSessionFactory(){
    return (sessionFactory==null ? buildSessionFactory() : sessionFactory);
  }
  public static Session openSession(){
```

```
    return getSessionFactory().openSession();
  }
```

**Hibernate3**

```
private static final ThreadLocal<Session> threadLocal = new
ThreadLocal<Session>();//ThreadLocal对象
  private static SessionFactory sessionFactory = null;//SessionFactory对象
  static {
    try {
    // 加载Hibernate配置文件
    Configuration cfg = new Configuration().configure();
    sessionFactory = cfg.buildSessionFactory();
  } catch (Exception e) {
    System.out.println(e.getMessage());
    System.err.println("创建会话工厂失败");
    e.printStackTrace();} }
//获取Session
  public static Session getSession() throws HibernateException {
    Session session = (Session) threadLocal.get();
  if (session == null || !session.isOpen()) {
    if (sessionFactory == null) {
      rebuildSessionFactory();}
    session = (sessionFactory != null) ? sessionFactory.openSession(): null;
    threadLocal.set(session);}return session;}
//重建会话工厂
public static void rebuildSessionFactory() {
  try {
    // 加载Hibernate配置文件
    Configuration cfg = new Configuration().configure();
    sessionFactory = cfg.buildSessionFactory();
```

```
    } catch (Exception e) {
        System.err.println("创建会话工厂失败");
        e.printStackTrace();}}
  //获取SessionFactory对象
  public static SessionFactory getSessionFactory() {
        return sessionFactory;}
  //关闭Session
    public static void closeSession() throws HibernateException {
        Session session = (Session) threadLocal.get();
        threadLocal.set(null);
        if (session != null) {
            session.close();//关闭Session}}
```

### 2.3.2. session方法

1、保存方法

Serializable save(Object o)；

2、查询方法

T get(Class c，Serializable id)；

T load(Class c，Serializable id)；

3、修改方法

void update(Object o);

4、删除方法

void delete(Object o)；

5、保存或更新

void saveOrUpdate(Object o)；

get和load方法的区别：

get：

1、采用的是立即加载，执行到此代码时，就会马上发送SQL语句去查询。

**2、查询返回的是真实对象本身。**

**3、查询一个对象找不到时返回null。**

**load：**

**1、采用的是延时加载，（lazy懒加载），执行到此代码时，不会立即发送SQL语句，仅当真正用到这个对象时才去发送SQL语句。**

**2、查询后返回的是代理对象。**

**3、查询一个对象找不到的时候，返回异常信息。**

### 2.3.3. 1/2/3:获取session对象

**4、手动开启事务**

```
Transaction ts = session.beginTransaction();
```

**5、数据库操作（增删改查等）**

**6、提交事务**

```
ts.commit();
```

**7、释放资源**

```
session.close();
```

**示例：**

```
//保存
User u1 = new User();
session.save(u1);
ts.commit();
//更新
User u2 = session.get(User.class, 3);
u2.setAddress("江苏省无锡市);
session.update(u2);
ts.commit();
//批量更新
Query queryUpdate = session.createQuery("update User set userName=? where
```

```
userName=?");
queryUpdate.setParameter(0, "123");
queryUpdate.setParameter(1, "456");
queryUpdate.executeUpdate();
//删除
User u3 = session.get(User.class, 1);
session.delete(u3);
ts.commit();
//批量删除
Query queryDelete = session.createQuery("delete User where userName = ?");
queryDelete.setParameter(0, "123");
queryDelete.executeUpdate();
```

## 2.4. HQL

### 2.4.1. 占位符

1.hql中可以使用别名的方式来查询，格式是 :xxx

通过setParameter来设置别名

2.基于 ? 的参数化形式(

jdbc的setParameter的下标从1开始，hql的下标从0开始)

3.JPA占位符的方式

```
String hql = "select a from Apple a where a.color=?2 a.weight>?5";
query.setParameter("2", "red");
query.setParameter("5", "10");
```

### 2.4.2. 分页查询

通过setFirstResult(0).setMaxResults(10)可以设置分页查询，相当于offset和pagesize

**procedure实现:**

```
create or replace procedure showPage(
    pageNo in number, --要显示哪一页
    pageSize in number, --每一页显示多少
    rowsCount out number, --总共多少条记录
    pageCount out number, --能分多少页
    cur out sys_refcursor --查询结果
)
as
begin
 select count(empno) into rowsCount from emp;
 if mod (rowsCount,pageSize) != 0 then
   pageCount :=round(rowsCount/pageSize)+1 ;
 else
   pageCount := rowsCount/pageSize;
 end if;
 open cur for select empno,ename from (select rownum rn,a.* from emp a)
    where  rn>= ((pageNo-1)*pageSize)+1 and rn<=pageNo*pageSize;
end;
```

### 2.4.3. in 进行列表查询

```
List<Student> stus = (List<Student>)session.createQuery("select stu from
Student stu where stu.room.id in (:room) and stu.sex like
:sex").setParameterList("room", new Integer[]{1, 2}).setParameter("sex",
"%女%") .list();
```

如果一个参数通过别名来传入，一个是通过 ?

的方式来传入的话，那么通过别名的**hql**语句以及参数设置语句要放在 ?

的后面，不然**hibernate**会报错。如果都是使用 别名

来设置参数，则无先后顺序

### 2.4.4. 左外连和右外连查询

```
List<Object[]> stus = (List<Object[]>)session.createQuery
("select room.name, count(stu.room.id) from Student stu right join stu.room
room group by room.id").list();
```

### 2.4.5. group having

在hql中不能通过给查询出来的字段设置别名，别名只能设置在from 后面

```
List<Object[]> stus = (List<Object[]>)session.createQuery("select special.name,
count(stu.room.special.id) from Student stu right join stu.room.special special
group by special.id having count(stu.room.special.id)>150") .list();//
```

查询出人数大于150个人的专业

## 2.5. 原生SQL

### 2.5.1. private Session s =null;

```
@Before
public void init() {
  this.s = HibernateUtil5.openSession();
}
@Test
public void a () {
  String sql = "select *  from GOODS";
  Query<Goods> query =
       s.createNativeQuery(sql,Goods.class)
        .setFirstResult(1).setMaxResults(2);
  System.out.println(query.getResultList());
}
@SuppressWarnings("unchecked")
```

```
@Test
public void b () {
  String sql = "select ID  from GOODS";

//当查询的结果并非完整的列的时候,不能s.createNativeQuery(sql,Goods.class
);
  Query<Goods> query = s.createNativeQuery(sql);
  System.out.println(query.getResultList());
}

@Test
public void c () {
  String sql = "select *  from GOODS where id  = :l_id";
  Query<Goods> query =
      s.createNativeQuery(sql,Goods.class).setParameter("l_id", 1);
  System.out.println(query.getResultList());
}
```

# 3. Spring

## 3.1. IoC

### 3.1.1. IoC（Inversion of Contril）：控制反转

它使程序组件之间尽量形成一种松耦合的结构，开发者在使用类的实例之前，需要先创建对象的实例。只不过Spring将创建实例的任务交给了IoC容器。简单来说，就是将对象的创建权反转给了Spring。

DI：依赖注入

前提必须有IoC环境，Spring管理某个类时将类的依赖的属性注入进来。

1、setter注入

基于javaBean的setter方法为属性赋值

2、构造器注入

基于构造方法为属性赋值，容器通过调用类的构造方法，将其所需的依赖关系注入其中。

构造器注入的额好处是，实例化对象的同时就完成了属性的初始化。

## 3.2. AOP

## 3.3. Spring的配置

### 3.3.1. Bean标签的id和name属性：

**id**：使用了约束中的唯一性约束，不能出现重复值，并且不能出现特殊字符

**name**：没有使用唯一性约束（理论上可以出现重复的，但实际开发中不能出现）

**Bean的作用域的配置：**

**scope**：Bean的作用范围

singleton：默认的，Spring会采用单例模式创建对象，在整个Spring IoC容器中，此作用域中的Bean只生成一个实例

prototype：多例模式，每次都会创建一个新的对象

request：应用在web项目中，Spring创建这个类以后，将这个类存入request范围中，相应的还有session范围

global session：每个全局的HTTP Session对应一个Bean实例，仅在使用porlet context的时候有效

## 3.4. Spring的Bean管理（XML方式）

### 3.4.1. Spring的Bean实例化方式（了解）

1、无参构造方法的方式（默认）

类中需要存在无参构造方法

示例：

```
public calss Bean{
    private String name;
```

```
    public Bean(){}
}
<bean id="bean" class="com.ebao.pojo.Bean"/>
```

静态工厂实例化方式

2、静态工厂实例化方式

需要编写Bean的静态工厂

示例：

```
public class BeanFactory{
  public static Bean getBean(){
    return new Bean();
  }
}
<bean id="bean" class="com.ebao.factory.BeanFactory" factory-
method="getBean"/>
```

3、实例工厂实例化的方式

工厂类：

```
public class BeanFactory{
    public Bean getBean(){
      return new Bean();
    }
}
<bean id="bean" class="com.ebao.factory.BeanFactory" factory-
method="getBean"/>
```

## 3.5. Spring的属性注入

### 3.5.1. 1、构造方法方式的属性注入

配置文件：

```
<bean id="bean" class="com.ebao.Bean">
    <constructor-arg nam="属性名" value="属性值"/>
</bean>
```

**2、setter方法的属性注入**

配置文件：

```
<bean id="bean" class="com.ebao.Bean">
   <property nam="属性名" value="属性值"/>
</bean>
```

setter方法注入对象类型的属性

```
<bean id="bean" class="com.ebao.Bean">
   <property nam="普通属性名" value="属性值"/>
   <property nam="类实例属性名" ref="属性值"/>
</bean>
```

# 4. SSH整合

## 4.1. 依赖JAR

### 4.1.1. 数据库:

**ojdbc7;**

**c3p0;**

**mchange-commons-java**

### 4.1.2. Spring:(4.3.18)

**spring-core**

**spring-beans**

**spring-web**

**spring-context**

**spring-tx**

**spring-test**

**spring-webmvc**

**spring-jdbc**

**spring-aop**

**4.1.3. Hibernate:(5.3.3)**

**hibernate-c3p0:**

**hibernate-core**

**4.1.4. Struts(2.5.16)**

**struts2-core**

**Struts Json**

**json-lib 2.4;**

**struts2-json-plugin 2.5.16;**

**net.sf.ezmorph 1.0.6;**

**commons-beanutils 1.9.3;**

**4.1.5. 整合:**

**struts2-spring-plugin 2.5.16;**

**spring-orm;**

**4.2. web.xml**

**4.2.1. <!-- spring监听器配置 -->**

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath:applicationContext.xml</param-value>
</context-param>
<listener>
  <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

**4.2.2. <!-- STRUTS 2.5 -->**

```
<filter>
  <filter-name>struts-prepare</filter-name>
  <filter-class>org.apache.struts2.dispatcher.filter.StrutsPrepareFilter</filter-
```

```
class>
  </filter>

  <filter>
    <filter-name>struts-execute</filter-name>
    <filter-class>org.apache.struts2.dispatcher.filter.StrutsExecuteFilter</filter-
class>
  </filter>

  <filter-mapping>
    <filter-name>struts-prepare</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

  <filter-mapping>
    <filter-name>struts-execute</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
```

### 4.3. ApplicationContext.xml(spring开启注解)

#### 4.3.1. <?xml version="1.0" encoding="UTF-8"?>

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:p="http://www.springframework.org/schema/p"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="
      http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop-4.3.xsd
      http://www.springframework.org/schema/beans
```

```
        http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-4.3.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-4.3.xsd"
  default-autowire="byName" default-lazy-init="true">
<!-- spring配置默认default-lazy-init为false，
     当配置为true时sping不会再去加载整个对象实例图，
     大大减少了初始化的时间，减少了spring的启动速度。
     default-autowire="byName"(通过名字自动装配)
-->
```

**4.3.2.**

```
<context:component-scan
    base-package="com.ssh.dao.impl,com.ssh.service.impl,com.ssh.action">
</context:component-scan>
//开启自定义扫描的包
```

**4.3.3.**

```
<bean id="mySessionFactory"
    class="org.springframework.orm.hibernate5.LocalSessionFactoryBean"
    scope="prototype">
    <property name="configLocation"
      value="classpath:hibernate.cfg.xml" />
</bean>
//通过 hibernate.cfg.xml来获取 hibernate session
```

**4.3.4.**

```
<!-- 配置事务管理器， -->
  <bean id="transactionManager"
    class="org.springframework.orm.hibernate4.HibernateTransactionManager">
    <property name="sessionFactory" ref="mySessionFactory" />
  </bean>
<!-- 开启注解事务 -->
```

```
    <tx:annotation-driven
      transaction-manager="transactionManager" />
```

### 4.4. ApplicationContext.xml(配置装配)

#### 4.4.1. <?xml version="1.0" encoding="UTF-8"?>

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
   http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
   http://www.springframework.org/schema/aop
   http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
   http://www.springframework.org/schema/tx
   http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
   http://www.springframework.org/schema/context
   http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <bean name="productActionBean"
class="com.how2java.action.ProductAction">
      <property name="productService" ref="productServiceImpl" />
    </bean>
    <bean name="productServiceImpl"
class="com.how2java.service.impl.ProductServiceImpl">
      <property name="productDAO" ref="productDAOImpl" />
    </bean>
    <bean name="productDAOImpl"
class="com.how2java.dao.impl.ProductDAOImpl">
      <property name="sessionFactory" ref="sf" />
```

```
  </bean>
  <bean name="sf"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="ds" />
    <property name="mappingResources">
      <list>
        <value>com/how2java/pojo/Product.hbm.xml</value>
      </list>
    </property>
    <property name="schemaUpdate">
      <value>true</value>
    </property>

    <property name="hibernateProperties">
      <value>
        hibernate.dialect=org.hibernate.dialect.MySQLDialect
        hibernate.show_sql=true
        hbm2ddl.auto=update
      </value>
    </property>
  </bean>
  <bean name="ds"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <property name="url"
value="jdbc:mysql://localhost:3306/ssh?characterEncoding=UTF-8" />
    <property name="username" value="root" />
    <property name="password" value="123456" />
  </bean>
</beans>
```

### 4.5. Hibernate.cfg.xml

#### 4.5.1. <?xml version="1.0" encoding="UTF-8"?>

```
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
   <session-factory>
      <property name="hibernate.connection.username">C##SHOP</property>
      <property name="hibernate.connection.password">123</property>
      <property
name="hibernate.connection.driver_class">oracle.jdbc.driver.OracleDriver</prop
erty>
      <property
name="hibernate.connection.url">jdbc:oracle:thin:@localhost:1521:orcl</propert
y>
      <property
name="hibernate.dialect">org.hibernate.dialect.OracleDialect</property>
      <!--C3P0数据 -->
      <property name="hibernate.connection.provider_class">
         org.hibernate.c3p0.internal.C3P0ConnectionProvider
      </property>
      <!-- 显示SQL -->
      <property name="hibernate.show_sql">false</property>
      <!-- 格式化SQL -->
      <!-- <property name="hibernate.format_sql">true</property> -->
      <!-- 注册mapping -->
      <mapping resource="com/ssh/entity/Goods.hbm.xml" />
      <mapping resource="com/ssh/entity/Category.hbm.xml" />
      <mapping resource="com/ssh/entity/User.hbm.xml" />
```

```
    </session-factory>
</hibernate-configuration>
```

### 4.6. Goods.hbm.xml

#### 4.6.1. <?xml version="1.0"?>

```
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<!-- Generated 2018-6-4 15:57:54 by Hibernate Tools 3.5.0.Final -->
<hibernate-mapping>
  <class name="com.ssh.entity.Goods" table="GOODS"
    schema="C##SHOP">
    <id name="id" type="java.lang.Integer">
      <column name="ID"></column>
      <generator class="sequence">
        <param name="sequence_name">HIBERNATE_SEQUENCE</param>
      </generator>
    </id>
    <property name="goodsName" type="java.lang.String">
      <column name="GOODS_NAME"></column>
    </property>
    <property name="goodsDesc" type="java.lang.String">
      <column name="GOODS_DESC"></column>
    </property>
    <property name="goodsPrice" type="java.lang.Double">
      <column name="GOODS_PRICE"></column>
    </property>
    <property name="goodsImg" type="java.lang.String">
      <column name="GOODS_IMG"></column>
    </property>
    <property name="storage" type="java.lang.Integer">
```

```
        <column name="STORAGE"></column>
      </property>
      <property name="createTime" type="java.sql.Timestamp">
        <column name="CREATE_TIME"></column>
      </property>
      <many-to-one name="category"
        class="com.ssh.entity.Category">
        <column name="CATEGORY_ID" />
      </many-to-one>
  </class>
  </hibernate-mapping>
```

### 4.7. struts.xml

#### 4.7.1. Struts 管理 Action

```
  <?xml version="1.0" encoding="UTF-8" ?>
  <!DOCTYPE struts PUBLIC
      "-//Apache Software Foundation//DTD Struts Configuration 2.5//EN"
      "http://struts.apache.org/dtds/struts-2.5.dtd">
  <struts>

    <package name="rz" extends="struts-default" strict-method-
  invocation="false">

      <action name="hello_*" class="com.ssh.action.GoodsAction" method="{1}" >
        <result name="ok">/WEB-INF/views/success.jsp</result>
      </action>
      <action name="user_*" class="com.ssh.action.UserAction" method="{1}" >
        <result name="success" type="chain">hello_index</result>
        <result name="error">/index.jsp</result>
      </action>
```

```
  </package>
</struts>
```

#### 4.7.2. Spring 管理 Action

```
<constant name="struts.objectFactory" value="spring" />
  <package name="rz" extends="struts-default" strict-method-
invocation="false">
    <action name="user_*" class="spring_userAction" method="{1}" >
      <result name="success" type="chain">hello_index</result>
      <result name="error">/index.jsp</result>
    </action>
    <action name="hello_*" class="spring_GoodsAction" method="{1}" >
      <result name="ok">/WEB-INF/views/success.jsp</result>
    </action>
  </package>
```

ApplicationContext.xml

```
  <bean id="spring_userAction" class="com.ssh.action.UserAction"/>
  <bean id="spring_GoodsAction" class="com.ssh.action.GoodsAction"/>
```

### 4.8. Dao/Service/Action

#### 4.8.1. Dao:(DaoImpl)

```
  @Autowired
  public SessionFactory sessionFactory;
```

#### 4.8.2. Service:(ServiceImpl)

```
  @Autowired
  private GoodsDao goodsDao;
```

#### 4.8.3. Action:

```
@Autowired
private GoodsService goodsService;
```

### 4.9. JUnit

#### 4.9.1. 测试Dao层

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
private static ApplicationContext context = null;
  SessionFactory sessionFactory = null;
  @Before
  public void init() throws Exception {
    context = new ClassPathXmlApplicationContext("applicationContext.xml");
    sessionFactory = (SessionFactory) context.getBean("mySessionFactory");
  }
  @Test
  public void a() {
    Session s= sessionFactory.openSession();
    System.out.println(s.createQuery("from Goods").list().toString());;
  }
```

#### 4.9.2. 测试Service层

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
private static ApplicationContext context = null;
  GoodsService goodsService  =null;
  UserService userService = null;
  @Before
  public void init() throws Exception {
    context = new ClassPathXmlApplicationContext("applicationContext.xml");
```

```
        goodsService = context.getBean(GoodsService.class);
        userService = context.getBean(UserService.class);
    }
    @Test
    public void a () {
        System.out.println(goodsService.getAllGoods());
    }
    @Test
    public void user() {
        User user = new User();
        user.setUserName("123");
        user.setUserPass("123");
        System.out.println(userService.findByNameAndPwd(user));
    }
```

#### 4.9.3. Spring-test整合Junit

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)//表示整合JUnit4进行测试 (4.12)
@ContextConfiguration(locations={"classpath:applicationContext.xml"})//加载s
pring配置文件
public class BaseTest {
    @Test
    public void a () {
        System.out.println("a");
    }
}
```

```
public class JunitSpring extends BaseTest{
   @Autowired
   private SessionFactory mySessionFactory ;
   @Test
   public void a() {
      Session s= mySessionFactory.openSession();
      System.out.println(s.createQuery("from Goods").list().toString());;
   }
}
```