# JAVAEE应用和开发环境

稳定的性能、良好的开放性、严格的安全性、对异构系统的整合、高并发、高稳定性，选择JAVA EE。

## 1.1 JAVA EE应用概述

### 1.1.1 JAVA EE应用的分层模型

分层：POJO、Dao、业务逻辑层、控制器层、表现层。

## 1.2 轻量级Java EE应用相关技术

Struts2 MVC框架，替代产品Spring MVC与JSF；

# 第二章 JSP/Servlet及相关技术详解

JSP与servlet是java ee规范的2个基本成员，JSP本质上也是Servlet；目前JSP只是单一的表现层技术。

## 2.1 Web应用和web.xml文件

### 2.1.1 构建Web应用

一个web应用的结构是webDemo下包含WEB-INF文件夹与其他的JSP或者包含JSP的文件夹；WEB-INF文件夹下需要包含web.xml、classes文件夹与lib文件夹；classes保存单个.class文件，lib保存JAR；复制到tomcat下的webapps文件夹下可以自动加载。

### 2.1.2 配置描述符web.xml

web.xml称为web应用的配置描述符，很重要，用来配置管理web组件，WEB-INF是特殊的文件夹，WEB容器会包含该文件夹下的内容，客户端无法访问这个文件夹的内容，也可以使用注解的方式配置Web组件；web的大部分组件都在这里配置，在web.xml中需要配置管理JSP、Servlet、Listener、Filter、标签库与JSP属性、JAAS授权认证、资源引用与首页。根元素是<web-app>，metadata-complete(true or false)表明是否会加载注解配置的Web组件。<welcome-file-list>元素指定Web应用的首页，每个web容器（应用服务器）都提供一个系统的web.xml，用于描述所有web应用的共同的配置属性。Tomcat下的conf/web.xml下就是。

## 2.2 JSP的基本原理

JSP本质上是Servlet，使用输出流来输出页面的HTML元素，完全使用Servlet开发效率低下，JSP包含静态部分(HTML)与JAVA(java脚本)动态部分，每个JSP是一个Servlet实例，由Web应用编译成Servlet运行；Servlet接受请求并生成响应，tomcat下生成的Servlet代码在work目录下，会生成init初始化方法、destroy销毁方法与service（响应的方法）3个比较重要的方法，第一次访问时编译。

## 2.3 JSP的4种语法

注释：<!-- -->HTML注释，输出到客户端，<%-- -->JSP注释不会输出到客户端；

声明：<! >声明的变量与方法是Servlet类的成员变量与成员方法，可以加访问修饰符，也可以加static转换成类的属性与方法。

输出：<%=% > 类似print，结尾不加分号；

脚本：<% %>可执行代码，放在service方法里的，控制动态内容的生成，静态内容文本放在out.write里直接输出，而java代码则不变，生成响应执行的时候就生成了静态的HTML文本内容；可以做任何事情，可以声明变量，是方法的变量，所以没有修饰符。

## 2.4 JSP的3个编译指令

JSP指令是通知JSP引擎的消息，不直接输出，每个指令都有默认值；有3个命令；page：针对当前页面的指令，include包含另一个页面（包含源文件在编译），taglib：包含标签库。用法：<@ 指令名字 属性名=”属性值”>。

[language=”java”]声明脚本语言的种类默认值为java，无需设置

[extends=”\*.class”]指定生成的servlet需要继承的父类或者接口

[import=”\*.class”]导入生成的代码中需要用用到类的包，默认导入java.lang.\*,javax.servlet.\*,javax.servlet.jsp.\*,javax.servlet.http.\*

[session=”true|false”]是否需要HTTPSession

[buffer=”size kb”]设定输出缓冲区的大小，即out输出缓冲区的大小，可以为none

[autoFlush=”true|false”]缓冲区溢出时是否需要强制输出，true输出，false时，溢出产生异常

[isThreadsafe=”true|false”]是否线程安全的

[info=””]设置该页面的一段说明，通过Servlet.getServletInfo()可以获取这个值。

[errorPage=”相对路径”]指定错误处理页面，产生异常或者错误转向的页面

[isErrorPage=””]该页面是否是错误处理页面

[contentType=”mimeType[;charset=\*\*\*]”|”text/html;charset=\*\*\*\*”]设定网页的文件格式或者编码字符集

[pageEncoding=””]生成网页的编码字符集HTML字符集（缓冲区生成的文本）

@include也是静态包含页面，直接将页面包含到本页面，把包含页面的编译指令也包含进来，形成一个完整的JSP（body内容显示），再编译成Servlet。

## 2.5 JSP的7个动作指令

编译指令通知JSP引擎处理，在编译成Servlet时起作用，动作指令在运行时起作用，可替换成JSP脚本，是JSP脚本的标准化写法。

### 2.5.1 forward指令

<jsp:forward>：执行页面转向，将请求处理转发到下一个页面，请求地址不会变，页面内容被forward页面响应代替，而且转发时请求参数也一并传过去，并且可以使用<jsp:param>传递额外的参数，并没有向新页面发送请求，只是使用新页面Servlet来对用户生成响应，用法：

重定向一个HTML文件，JSP文件，或者是容器中的Servlet。

<jsp:forward page={"relativeURL" | "<%= expression %>"} />或者

<jsp:forward page={"relativeURL" | "<%= expression %>"} >

<jsp:param name="parameterName" value="{parameterValue | <%= expression %>}" />

</jsp:forward>

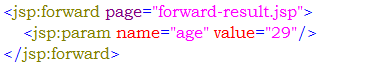
例子

<jsp:forward page="/servlet/login" />

<jsp:forward page="/servlet/login">

<jsp:param name="username" value="jsmith" />

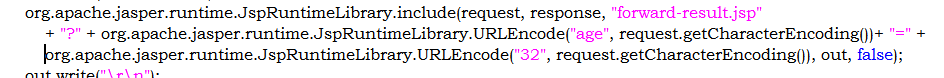
</jsp:forward>



### 2.5.2 include指令

<jsp:include>：动态include指令，不会导入页面的编译指令，导入body部分内容，可以使用<jsp:param>加参数；<jsp:include page="{relativeURL | <%= expression%>}" flush="true" />或者<jsp:include page="{relativeURL | <%= expression %>}" flush="true" ><jsp:param name="parameterName" value="{parameterValue | <%= expression %>}" /></jsp:include>

在Servlet中编译如下：



与静态导入的不同：动态只是使用include方法导入页面，静态融入成一个jsp；静态导入编译命令起作用，动态只导入body内容，动态可以加额外的参数；与forward指令非常像，forward使用目标页面完全替换响应，include只是插入响应。

### 2.5.3 useBean、setProperty、getProperty指令

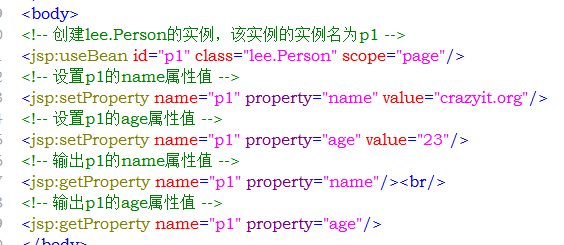
<jsp:useBean> setProperty与getProperty都是应用JAVAbean的，usebean实例化一个java类，可以指定作用范围；setProperty与getProperty作用该实例的属性，其实是调用的javabean的set与get方法。

<jsp:useBean id="id" class=”classname”scope="page|request|session|application" />。

<jsp:setProperty name=”beanName” property=”propertyName” value=”value”/>

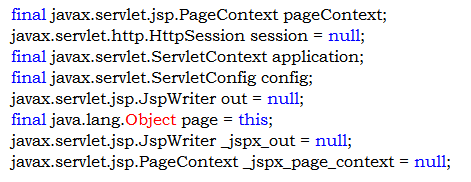
<jsp:getProperty name=”” property=”propertyName”/>

name是指java bean的属性名，



## 2.6 JSP脚本中的9个内置对象

jsp的内置对象是Servlet API接口的实例，JSP对它们进行了默认初始化，在\_jspservic()方法里创建，方便开发者直接调用这些对象。Request与response是service方法的形参，调用方法是初始化这2个对象，其余对象为方法局部对象，



他们都是jspservice里面的形参或者局部变量，可以直接在jsp页面的脚本与输出语句中使用。

### 2.6.1 application对象

附言：web服务器步骤：启动单独的线程、使用IO流读取用户请求的二进制数据流、从请求数据中解析参数、处理用户请求、生成响应数据、使用IO流向客户端发送数据；service方法完成3，4，5步。

Application对象：ServletContext的实例，每个Web应用只有一个，代表web应用本身，类似于一个Map；可用于页面之间交换信息，或者访问Web应用的配置信息；JSP，servlet都是web服务器调用的，启动单独线程、使用IO流读取用户请求数据，从请求中解析参数，处理用户请求、生成响应，使用IO流发送数据。相互之间交互存难题，所以使用application、sesssion、request、page作用域解决，这4个类似于map结构，application对整个Web有效，session对本次会话有效，request对本次请求有效，page对当前页面有效；application可以直接在JSP页面、Servlet中使用，获取配置参数主要是web.xml的context-param元素的内容，每个<context-param>配置一个参数，这些参数对整个Web应用都是有效的。主要的方法：getAttribute(String attName)，setAttribute(String attName, String attValue)，设置参数，getInitParameter(String paramName)。

### 2.6.2 config对象

Config对象：servletConfig的实例，代表当前JSP的配置信息，JSP不需要在web.xml中配置，所以基本没有配置信息，Servlet需要在web.xml配置，所以需要用到config对象，可以使用配置Servlet的方法配置JSP，效果是一样的。

可以使用getInitParameter(String paramName)获取参数值。



必须要使用这个配置文件规定的路径访问JSP才能通过config获取到initparam，直接访问JSP获取不到配置参数。

### 2.6.3 exception对象

Exception对象：是Throwable的实例，代表JSP脚本产生的的错误或者异常对象，只有isErrorPage为true时才会在Servlet类中中生成这个对象，在产生错误页面的Servlet的jspservice方法中的catch模块里面，会forword到errorPage页面生成的servlet，errorPage页面的service方法里面通过request可以获取这个异常生成的exception对象。

### 2.6.4 out对象

Out对象：页面输出流。与<%=>的作用一样。

### 2.6.5 PageContext对象

PageContext对象：代表页面上下文，用于页面间数据交互；可以访问很多作用域里面的数据，可以通过方法获取其他所有8个对象，常用的方法有getServletContext()获取application对象与getServletConfig()获取config对象、getRequest()获取request对象、getResponse()获取response对象、getSession()获取session对象，可以访问request、session、page与application中的所有数据，getAttribute(String name,int scope)可以获取指定范围内的数据，getAttribute(String name)获得是page对象里的变量值，还存在所有对应的set方法。

### 2.6.6 request对象

Request对象：request封装用户的一次请求，所有的请求参数都在Request对象中，包含请求头（浏览器控制）与请求参数（开发人员控制）2个部分，是Httpservletrequest接口的实例；主要的方法如下：

String getParameter(String paramName) 获取paramName请求参数的值；

String[] getParameterValues(String paramName)获取paramName请求参数的值;

Map getParameterMap()获取所有请求参数名和参数值所组成的Map对象；

Enumeration getParameterNames() 获取所有请求参数的名字。

提供很多获取参数的方法，也提供了很多方法获取请求头。GET方式请求，数据直接加在浏览器地址栏后面，数据不能超过2KB；POST方式请求，数据量大；只有有name属性的表单域才会生成请求参数；多个域有同一个name，生成多个值的请求参数；使用request对象里的参数时，必须首先设置request输入参数的解析字符，否则中文会有错误，使用get方式提交表单时，地址栏上的中文参数就已经是乱码了，此时获取这些字符的真实值就存在困难，使用URLDecoder类解析request可以获取正确的字符，或者对请求参数进行重新编码。当使用forward与include时，request与response在当前页面所做的修改也会直接转发到目标页面；属性不会丢失，新添加的属性也不会丢失。而且request本身有getRequestDispatcher(path)方法可以执行include与forward方法，这个方法返回RequestDispatcher对象，调用include或者forward方法完成forward与include操作，path必须以/开头。

setAttribute(String attName,String attValue);

Object getAttribute(String attName);操作Request对象里面的参数值。



### 2.6.7 response对象

Response对象：out输出响应，但是只能是字符型的，response用来进行输出非字符对象、重定向或者增加cookies；代表服务器对客户端的响应，是一个HttpservletResponse接口，提供getOutputStream()返回响应输出字节流；访问jsp时，直接返回字节对象，<img src=”\*.jsp”>嵌入到页面中，这种方式适合做那种生成验证码的功能。重定向发生了二次请求，原先的请求与request范围的对象会丢失，地址栏地址也会改变，使用sendRedirect(String path)方法重定向；增加cookies需要浏览器支持，session会随浏览器的关闭而失效，cookies一直存在，这方法为addcookie(Cookie cookie)，在response中创建cookie，并设置生命周期，在request中读取cookies，如果是中文可以使用Encoder与Deconder解码。

### 2.6.8 session对象

Session对象：代表一次回话，一直到浏览器关闭才会消失，用于跟踪用户的会话信息信息可以在多个页面间共享，可以实现购物车的功能等。

## 2.7 Servlet介绍

### 2.7.1 Servlet的开发

Servlet被称为服务器端小程序，必须继承于HttpServlet，必须实现doGet（响应GET请求）、doPost（响应POST请求）、doPut（响应客户端的PUT请求）、doDelete（响应客户端的DELETE请求）响应请求，现在可以只实现service方法，事实是上面的4个方法都调用了service方法，它完成所有的功能，还有init初始化Servlet、destroy（销毁时调用）方法，只有在涉及到资源分配即回收时才写init与destroy方法，所以不用编写构造器，Servlet没有内置对象（JSP有比如Application）。

### 2.7.2 Servlet的配置

Servlet可以使用注解@WebServlet与web.xml这2种配置方法，如果在命令行上直接编译，必须将servlet的相关类路径加入到CLASSPATH里面，[注解配置就是@WebServlet，使用web.xml](mailto:注解配置就是@WebServlet，使用web.xml)配置时使用<servlet>配置Servlet的名字与<servlet-mapping>配置Servlet的URL，如果没有URL则不能响应用户请求；这2个元素；

### 2.7.3 Servlet的声明周期

servlet具有2种启动方式：1.第一次访问时编译为Servlet与class，2.web应用启动时，它们都是先创建Servlet实例，用inti初始化，调用方法响应，销毁；

### 2.7.4 load-on-startup Servlet

load-on-startup指定就是应用启动时实例化，这种Servlet通常用于提供后台服务，或者拦截请求的，作为基础Servlet使用；有注解与xml配置2种配制方法，配置时可以指定一个整数值，代表优先级，越小越优先初始化。

### 2.7.5 访问Servlet的配置参数

也可以配置初始化参数initparam，也是注解与xml2种配置方式。注解是使用@WebServlet的initParams属性指定，XML是<init-param>元素指定。在内部可以使用servletConfig对象获取这些配置参数。

### 2.7.6 使用Servlet作为控制器

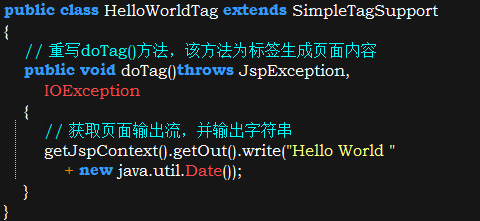
Servlet用于表现层，效果太差，不如JSP，在MVC模式中，一般都是作为控制器层存在的，JSP->Servlet(Controller)->model(逻辑处理)->返回。

## 2.8 JSP2的自定义标签，非常重要的技术

JSP标签就是封装的<%%>脚本，具有类似于HTML的语法结构，完成脚本的功能，便于开发；开发步骤为：开发标签处理类、建立.tld文件包含多个标签、使用标签。

### 2.8.1 开发自定义标签类

标签在JSP生成Servlet时，实际上替换成了java脚本，然后编译为Servlet；开发自定义标签类要继承SimpleTagSupport方法，包含的属性必须要有setter与getter方法，重写doTag()方法用于生成页面内容。比如：



### 2.8.2 建立TLD文件

TLD文件的全称是tag library definition，标签库定义文件，就是一个xml文件，根元素是taglib，包含很多tag元素，每个tag元素就是一个标签，将tld文件复制到WEB-INF任意子路经下就会自动加载，<taglib>包含4个子元素tlib-version（自定义的版本，没什么用），short-name（默认短名，没用），uri(很重要，唯一标识名)，tag（name-标签名字，tag-class-标签处理类，body-content指定标签体内容（tagdependent-指定标签处理类自己负责处理标签体、empty-指定标签只能作为空标签使用、scriptless-指定标签体可以是静态HTML、表达式不能是JSP脚本、JSP-指定标签体可以是JSP脚本），dynamic-attributes-指定标签是否支持动态属性）。

### 2.8.3 使用标签库

使用标签分为2点：1.导入标签库，将URI标签与指定前缀联系起来，2.使用标签；

<%@ taglib uri=”” prefix=”tagprefix” %>在绑定前缀；

<tagPrefix:tagName tagAttribute=”vakue”...>

<tagbody/>

</tagPrefix:tagName>,如果没有标签体，直接写标签，没有属性，可以去掉属性。

### 2.8.4 带属性的标签

带attributes的标签中的标签处理类的属性要具有setter、getter，并且在tag元素内要增加<attribute>元素，每个元素都代表标签处理类中的一个属性，attribute还包括name属性名，requered是否为必须属性，fragment是否支持JSP脚本表达式动态内容等子元素。

### 2.8.5 带标签体的标签

带标签体的标签也要有特殊的处理，getJspBody()会返回标签体对象JspFragment，这个对象有一个invoke()方法，执行它就输出标签提的内容。

### 2.8.6 以页面片段作为属性的标签

标签中的属性也可以是页面片段，也就是传入JspFragment 类型的成员变量，如果有这种成员，赋值的时候要使用<jsp:attribute>元素。比如：



### 2.8.7 动态属性的标签

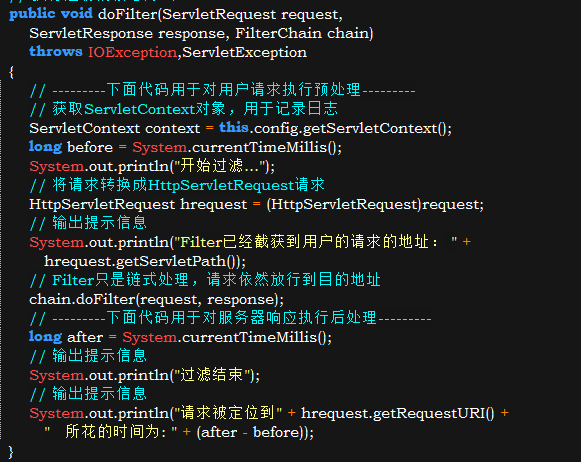
同时，标签的属性可以设置为动态的，个数不限。标签处理类必须实现DynamicAttributes接口，在TLD文件配置时要指定<dynamic-attributes>元素为true。接口实现setDynamicAttribute()方法，这个方法就是用来处理不限个数的属性的，因为标签处理类里面的成员都是List，所以理论上是可以存无限个属性的。

## 2.9 filter介绍

是Servlet的加强版，拦截请求request与响应response，可以做任何修改，在客户端JSP与服务器Servlet处理类之间的桥梁，拦路收费那种的，一个filter可以拦截多个请求，一个请求也可以被多个filter拦截。步骤：创建Filter、配置Filter。Filter有几种：用户授权的Filter、日志Filter、负责解码的Filter、能改变XML内容的XSLT Filter等。

### 2.9.1 创建Filter类

Filter必须实现javax.servlet.Filter接口，必须实现3个方法init(FilterConfig config)初始化、destroy销毁前释放资源、doFilter（ServletRequest、ServletResponse、FilterChain）实现桥梁关卡功能。



doFilter方法内请求与响应的分割线为是否调用了chain.doFilter(request,resp onse)；将请求重新发送到目的地址。只是记录了日志。

### 2.9.2 配置Filter

配置Filter基本跟Servlet一样，需要配置Filter名字与拦截的URL模式，不同的是Fileter的URL是范围URL，配置Filter可以通过注解与XML2种方式@WebFilter（里面有一些配置Filter的详细属性），XML配置与Servlet配置方式一致；Filter的主要用途是将很多Servlet的共同的操作抽象出来形成Filter操作，主要是为了解决代码重复的问题，Filter与Servlet基本是一致的，配置参数的方法也是一致的。

### 2.9.3 使用URL Rewrite实现网站伪静态

可以通过URL rewrite实现html到jsp的伪静态网站操纵，下载这个工具，在web.xml文件中配置拦截路径在WEB-INF路径下建立一个拦截规则的xml文件就可以了。

## 2.10 Listener介绍

Web应用启动与运行过程中，会发生各种事件，Servlet API提供了大量监听器接口，当这些事件发生时该做什么处理时就回调到实现了相应接口的Listener，步骤：首先要定义，然后配置Listener。

### 2.10.1 实现Listener类

常用的监听器接口为ServletContextAttributeListener、ServletContextListener、ServletRequestAttributeListener、ServletRequestListener、HttpSessionListener、HttpSessionAttributeListener；ServletContextListener用于监听web应用的启动与关闭；包含contextInitialized(ServletContextEvent sce)应用启动时调用与contextDestroyed(Serv letContextEvent sce)应用关闭时调用2个方法。

### 2.10.2 配置Listener

配置Listener要向Web应用注册Listener实现类，没有配置参数；可以使用注解或者XML配置，@WebListener注册监听器，xml使用<listener>与<listener-class>配置。

### 2.10.3 使用ServletContextAttributeListener

用于监听ServletContext(application)范围内属性的变化，需要实现3个方法：

AttributeAdded(ServletContextAttributeEvent event),增加属性时调用；

AttributeRemoved(ServletContextAttributeEvent event),移除属性时调用；

AttributeReplaced(ServletContextAttributeEvent event),替换属性时调用。

### 2.10.4 使用ServletRequestAttributeListener与ServletRequestList ener

request部分的监听器是监听一次请求与request域的改变的，request包含requestInitialized（ServletRequestEvent sre）用户请求到达触发与requestDestroyed(ServletRequestEvent sre)用户请求结束触发；ServletRequestAttributeListener与ServletContextAttributeListener差不多。

### 2.10.5 使用ServletSessionAttributeListener与ServletSessionList ener

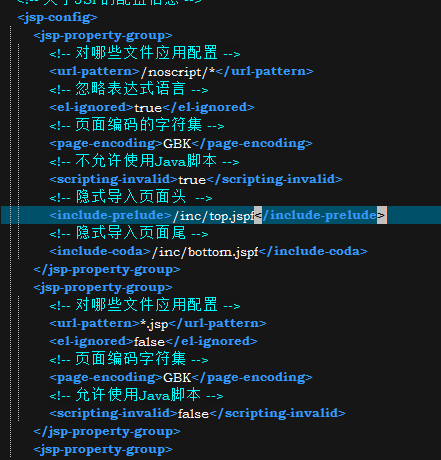
这些监听器的方法都差不多，一个类如果都实现了很多监听接口，则他就可以监听所有的事件。

## 2.11 JSP 2 特性

主要新特性：直接配置JSP属性、表达式语言、自定义标签API、Tag文件语法。

### 2.11.1 配置JSP属性

配置JSP属性：<jsp-property-group>(<url-pattern>,<el-ignored>,<scri pting-invalid>,<page-encoding>,<include-prelude>)。如下：



<url-pattern>指定的通配符JSP页面组都含有的共同的属性。实际是配置编译指令。

### 2.11.2 表达式语言

表达式语言是简化的数据访问方式，是一种数据访问语言，${expression}，expression支持算术与逻辑运算符，表达式语言有很多内置对象，代替了JSP中的内置对象，它们是相通的。内置对象有pageContext、pageScope、requestScope、sessionScope、applicationScope、param、paramValues、header、headerValues、initParam与cookie；还可以使用自定义函数，步骤为：1.定义类，类中的函数定义成类的静态方法；2.配置在标签库文件中，使用<function>配置，有3个子元素<name>函数调用名<function-class>函数实现类<function-signature>函数实际函数名，可以像标签那样使用函数，并且可以传入参数。

### 2.11.3 Tag File支持

还可以使用tag file，放在特定路径下，加载，tag就是文件名，tagfile里面就像JSP一样，有很多编译指令，其实web容器处理后也是生成了java代码的。标签替换了JSP上的内容片段。

## 2.13 Servlet 3.0的新特性

### 2.13.1 Servlet3.0的注解

注解代替xml配置，@WebServlet、@WebInitParam、@WebListener、@WebFilte r等。

### 2.13.2 Servlet3.0 的Web模块支持

模块化开发，Servlet3.0不要求所有的Web组件都部署在同一个web.xml中，可以采用Web模块来部署，每个Web模块是一个jar包，jar包含有2部分内容：1.META-INF文件夹，里面包含了一个web.fragment.xml文件，叫做Web模块的部署描述符，跟Web.xml类似的；2.web模块所用的类文件与资源文件等。web-fragment.xml与web.xml一样，多了<name>用于标识模块名，<ordering>标识模块之间加载的相对顺序。web-fragment.xml的根元素是<web-fragment>,web模块打包成jar包，复制到任意web应用的lib文件夹下就可以使用了，可以在Web.xml文件中设置模块的绝对加载顺序。这种方式为框架开发提供了方便。

### 2.13.3 Servlet3.0提供的异步处理

Servlet会等待业务方法的返回，这样耗时业务方法会令Servlet阻塞，异步处理可以建立一个新线程执行业务方法，Servlet启动线程后便返回，这种异步处理是使用AsyncContext类来实现的，ServletRequest开启异步调用创建AsyncContext的方式如下：

AsyncContext startAsync();

AsyncContext startAsync(ServletRequest,ServletResponse);

在同一个Servlet中重复调用返回的是相同的AsyncContext代表异步处理的上下文，可以设置异步调用的超时时常等，dispatch()方法可以用于转发请求。

必须为Servlet显示指定启动异步调用，使用AsyncContext需要对Servlet进行支持配置，有注解与xml2种方式，注解配置时为@WebServlet配置asyncSupported=true属性，XML配置时在<Servlet>元素下配置<async-supported>元素，Servlet启用异步线程后，会立即向客户端输出响应，异步线程执行结束后，生成的响应会被再次送到客户端。如果想要对执行细节了解，可以使用异步监听器，实现AsyncListener接口来监听异步线程的执行。包含的方法有：

onStartAsync(AsyncEvent event)异步调用开始时触发；

onComplete(AsyncEvent event)异步调用完成时触发；

onError(AsyncEvent event)异步调用出错时触发；

onTimeout(AsyncEvent event)异步调用超时时触发；定义了监听器后，使用AsyncContext.addListener()方法添加监听器。

### 2.13.4 改进的Servlet API

改进的Servlet API：ServletRequest加了文件上传的支持，ServletContext对象可以在程序中动态的注册Servlet、Filter、Listener等。

ServletRequest对象中处理文件上传的方法：Part getPart(String name)根据名称来获取请求参数中的文件上传域，Collection<Part> getParts()获取求参数中的所有的文件上传域；上传文件的JSP标签为<input type=”file”>,在表达域要指定enctype=”multipart/form-data”,指定JSP以二进制流处理表单域数据，会把文件的内容也封装到请求参数中，application/x-www-form-urlencoded：默认的编码方式，只处理表达域的value值，以URL编码方式编码，随意这种方式提交文件时只是把文件名作为值提交了，text/plain：一般用于发邮件。处理文件上传的Servlet要使用@MultipartConfig注解或者<multipart-config>元素启用，获得Part对象后，可以访问文件对象的一系列属性，或者调用write(path)写到服务端的目录下。上传时，采用了文件本名，这样才多用户上传时容易重名，可以考虑在服务端改名。

ServletContext提供addServlet()、addFilter()、addListener()方法动态的注册Servlet、Filter与Listener。

## 2.13 新增的非阻塞式IO

Servlet底层的IO是通过ServletInputStream读取数据的输入流，ServletOutputStream输出数据的输出流，完成的，当数据没有读取完成时，Servlet会被阻塞，从3.1开始 ServletInputStream新增了setReadListener(ReadListener readListener)方法，可以以非阻塞IO方式读取数据，实现ReadListener接口需要实现3个方法：onAllDataRead()所有数据读取完成,onDataAvailable()有数据可用时,onError(Throwable t)读取数据出错时激发。使用非阻塞IO，要在Servlet上开启异步调用，通过Request对象获得ServletInputStream流，为这个流设置监听器，在监听器中以非阻塞的方式读取数据。

## 2.14 Tomcat 8的WebSocket支持

WebSocket允许通过javascript与服务器端建立连接，从而允许服务器端将数据推送给浏览器，可以构建实时性要求比较高的应用；Java EE提供了WebSocket的规范，Tomcat则提供了优秀的实现。1.使用注解方式：@ServerEndpoint修饰的java类可作为WebSocket服务端；2.继承Endpoint基类实现WebSocket服务端；与Servlet一样，Web容器负责网络连接与并发。Websocket只负责提供业务逻辑即可。

使用注解开发时，可以在类中的任意方法上设置4个注解：@OnOpen-客户端与服务端连接时回调这个方法，@OnnClose-断开连接时回调这个方法，@Message-服务端接收到客户端发送的消息时回调，@OnError-客户端与服务端连接出现错误时回调。服务端有多少个，就对应多少个客户端，每次客户端连接，就生成一个服务端端的Java对象，客户端javascript建立一个WebSocket对象，该对象也定义了onopen、onclose、onmessage几个回调函数。

# Struts 2 的基本用法

Struts2 继承了Struts1与WebWork两个经典的MVC框架。

## 3.1 MVC思想概述

### 3.1.1 传统的Model1与model2思想

Model1模式：程序全部由JSP构成，所有逻辑都在JSP中；

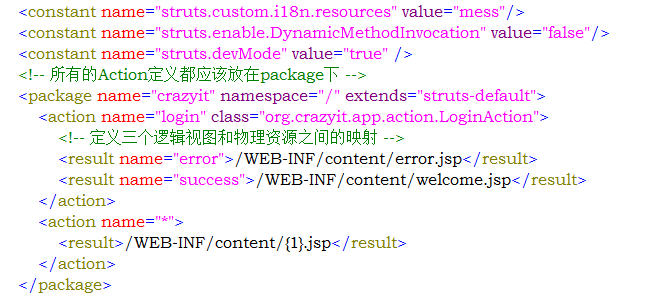
Model2模式：MVC:Control-Servlet，View-JSP，Model-Javabean，缺点就是代码更多，有点就是方便开发与维护。

### 3.1.2 MVC思想与优势

多个视图对应一个模型，模型返回的数据与显示逻辑分离，降低了耦合性，符合软件工程化管理。

## 3.2 Struts2的下载与安装

下载Struts2的完整版本，将最少属性的jar包放到lib路径下，如果需要自己手动编译java文件，在CLASSPATH里面要加入struts2核心jar包路径；在XML中配置Struts2的Filter核心过滤器，编写struts.xml文件，配置访问情况。使用Struts要严格遵守MVC模式，不能直接访问JSP，都只向控制器发请求，由控制器决定显示哪个视图；



在Eclipse中使用Struts2不过多赘述。

Struts.xml是非常重要的，需要在里面配置很多东西:1.国际化资源文件，2.Action配置，Struts.xml需要放到类加载路径下，因为在Eclipse里面开发，放到src目录或者子包下，Eclipse将自动将不是java的文件复制到相应的目录，src目录对用classes目录，所有的视图推荐放到WEB-INF里面，这样用户不能绕过控制器来直接访问JSP文件。

## 3.3 Strust2的流程

Struts2应用的开发步骤：

1. 在Web.xml文件中定义核心Filter拦截用户请求；MVC框架都要在Web.xml中定义框架自己的Servlet或者Filter才能介入到Web应用中，Struts2定义了自己的Filter拦截用户请求与返回的响应；
2. 定义JSP页面视图；
3. 定义处理用户请求的类Action，这是控制器，负责调用对用的业务逻辑的；其底层逻辑是：监听器接收到请求，对请求进行预处理，解析及参数封装等，通过反射创建action调用特定方法（struts1是execute(),struts2可以是任意方法）处理，有2种方法知道系统需要创建哪个Action实例：XML配置与约定，约定就是根据逻辑连接与Action之间的名字对应性进行反射，所以存在2个控制器，Filter是核心控制器，处理通用请求，Action是业务控制器。
4. 配置Action，XML文件；
5. 配置处理结果与物理试图资源之间的映射关系，返回的字符串就是逻辑视图名，需要配置与其关联的物理视图JSP路径；
6. 编辑视图资源；当Action需要把数据传给视图资源，可以使用OGNL表达式。

Struts的Filter与action共同构成了控制器，Filter称为核心控制器，action称为业务控制器，业务控制器只负责返回处理结果，处理结果与视图的关联是通过核心控制器处理的，核心控制器调用视图资源不是直接调用，是将请求forward到指定JSP页面。



## 3.4 Struts 2的常规配置

Struts2的常规配置就是xml文件，默认名字为struts.xml，放在类加载路径下/WEB-INF/classes路径下，作用是配置action与请求之间的对应关系，配置逻辑视图名与物理视图之间的对应关系，还有Bean配置、常量配置、导入其他配置文件等；还可以使用struts.properties来配置常量，key-value对，对Struts2应用整体起作用，也称为Struts属性，struts.properties放到类加载路径下，struts2框架将在启动后自动加载，可以在核心框架jar包中找到default.properties文件查看所支持的常量，常量中有很多非常重要要记住，仔细读。Struts2默认加载类加载路径下的struts.xml用户自定义，struts-default.xml自带的配置文件，struts-plugin.xml插件的默认配置文件，配置常量有3种，struts.properties，struts\*.xml，web.xml；在xml中配置常量就是使用<constant>元素，在web.xml中配置就是在配置StrutsPrepareAndExecuteFilter核心控制器的时候指定initParam，里面包含一个key-value对；Struts2框架加载Struts2常量的顺序struts-default.xml、struts-plugin.xml、struts.xml、struts.properties、web.xml；如果在多个文件都配置了，则后面的常量会覆盖前面的常量，随着struts.xml中action数量增多，可以进行拆分，形成很多与struts.xml形式一样的独立xml文件，在struts.xml文件中使用<include>包含进来，<include file=”\*\*\*\*\*.xml”>。

## 3.5 实现Action

Action就是一个简单的java类，但是action应该包含一个无参的execute()方法，并且Struts直接使用Action封装HTTP请求参数，所以action中要有request中的同名变量，需要设置setter与getter方法，通过set方法调用注入，真实变量名无所谓，只要set方法名跟request对象中变量对上即可，action里面的private变量还可以用于输出结果，也要提供set、get方法，可以直接在JSP中使用标签输出，Struts不区分Action中的变量是封装的请求参数，还是封装的处理结果，在JSP中使用标签输出时，也不会区分，都是一样处理；（我猜测应该request中数据封装到action中处理，处理后，action中的新变量值又加入了request中，然后forward时转交了request，或者action接到request后，根据所有的域生成了一个新的request，转交给了JSP）。

### 3.5.1 Action接口和ActionSupport基类

Struts2含有一个action接口与一个基本实现了Action接口的action类Action Support，Action接口包含了几个结果字符串与execute()方法，ActionSupport是默认的Action实现类，是Struts2的默认处理类，如果没指定class，就是用它处理，execute直接返回SUCCESS，实际开发中的Action继承它就简化开发。

### 3.5.2 Action访问Servlet API

Action需要获取连接的状态信息，则必须访问Servlet API对象，ActionConte xt类可以访问Servlet API，几个主要的方法如下：Object get(Obeject key)获取request中的属性，Map getApplication()获取ServletContext对象，static ActionContext getContext()返回系统的ActionContext实例，Map getParameters获取所有请求参数，Map getSession获取HttpSession，void setApplication(Map application)设置application，将该application中的key-value转换为application对象的属性，void setSession(Map)，与上面的方法相同。

### 3.5.3 Action直接访问Servlet API

实现接口获得API对象，实现ServletContextAware、ServletRequestAware、ServletResponseAware实现接口有一个set对象的方法，可以在这个方法中获得接口所代表的对象(这个方法中的对象参数应该是Struts2框架自己生成，方法是自己调用的)，即使可以在Action中获取Response对象，response也不能输出响应。因为他是业务控制器，不是核心控制器，核心控制器控制输出的。Forward操作时会替换响应。

### 3.5.4 使用ServletActionContext访问Servlet API

使用ServletActionContext工具类，这个类中包含了获取API对象的静态方法：static PageContext getPageContext()，static HttpServletRequest getRequest()，static HttpServletResponse getResponse()，static ServletContext getServletContext()。

## 3.6 配置Action

配置Action就是让Strust的拦截器知道哪个Action处理哪个请求，完成用户请求与Action之间的对应关系。

### 3.6.1 包和命名空间

Struts2使用包来组织Action，元素<package>代表包，每个包都是Action、拦截器、拦截器引用的集合，<package>元素具有以下几个属性：name包名字，extends，继承的包，是其他的包名字，将里面的Action等定义继承过来，namespace，命名空间，很重要；abstract：是否是抽象包，不包含Action定义；父包要在子包前面定义，Struts2在struts-default.xml中定义了一个抽象包struts-default，包含很多结果类型定义与拦截器定义等，这个是配置普通Action的基础，开发者的Pakage要继承这个包；如果用到了插件，也可以继承该插件的struts-plugin包来使用插件，namepsace用于指定该包对应的命名空间，是为了处理同名Action的问题，使用命名空间隔开，只能用于定义包，没指定命名空间，则为默认命名空间，设置了包的命名空间后，该包下访问Action的访问URL为命名空间/action名字的形式，记住跟路径/与默认命名空间是不同的，默认命令空间是某一个根命名空间下的命名空间，解析URL时，系统首先到命名空间中寻找该action，找到则处理，没有则去默认命名空间中寻找，找到处理，没有报错，命名空间只能处理该空间下的访问，默认命名空间可以处理所以空间的访问，包括根命名空间，命名空间不论写多少/都是一个级别，跟目录不一样，没有上级。

### 3.6.2 Action的基本配置

配置action指定name与class，class就是类，name是名字也是所处理请求的URL，如果不指定class，默认用ActionSupport处理，Action是一个逻辑控制器，不直接对浏览器生成响应，Action需要将指定视图资源呈现给用户，所以还要配置逻辑视图名与物理JSP，使用result元素，name中可以存在斜线，在struts的常量属性中指定，写点号或者横杠可能会报未知错误。

### 3.6.3 使用Action的动态方法调用

可以通过动态方法调用来将请求指向Action的其他方法，这样，一个方法就是一个逻辑控制器，Action类成为了逻辑控制器集合，这个叫做动态方法调用，可以在Struts2的常量设置中开启，数据提交URL的写法是acton！method；method与execute只是方法名字不同，其他相同。

### 3.6.4 指定method属性及使用通配符

还可以将一个Action类分成多个逻辑Action，就是加入method属性，每一个方法就可以是一个逻辑action，指定方法后就代替了execute来处理请求，方法签名必须是：参数列表为空，返回值为String，这种配置方式太麻烦，Struts2允许通配符的配置方式，通配符的意思就是在配置Action的name属性中可以出现\*号，代表任意个字符，接下来可以在class中、method、result中使用｛N｝来代表第几个\*号出现的字符串；使用通配符后，可以在一个action中配置多个逻辑action，class从处理类上实现多逻辑，method从方法上实现多逻辑，result实现结果的逻辑；可以同时配置这3个；（校验文件的命名规则是ActionName-validation.xml，或者ActionName-methodName-validation.xml），当请求的路径中找不到逻辑action来处理后，容器中可以定义一个默认的action，找不到就可以用它来处理，Struts2匹配Action的顺序就是先后顺序，哪个先匹配就找哪个物理action来处理。Name=”\*”可以匹配所有Action，所以一定要放到最后。

### 3.6.5 配置默认Action

配置默认的action使用元素<default-action-ref>。里面有一个属性name，值为容器中的一个逻辑action名字，这个逻辑action指定的class就是默认的处理类了；默认action配置在默认默认命名空间中就可以处理所有的用户请求。

### 3.6.6 配置Action的默认处理类

如果没有指定class，默认是ActionSupport处理，也可以自己制定默认处理类，使用元素<default-class-ref>，属性class指向一个自定义类，ActionSupport的默认处理类配置在struts-default包下面。

## 3.7 配置处理结果

Action负责处理请求，JSP负责展示，Action返回逻辑视图名字符串，控制器根据逻辑视图名找到对应的物理视图资源，所以要配置逻辑视图名与物理视图的对应关系。

### 3.7.1 理解处理结果

实际资源可是任意的前端界面或者是另一个Action也行。Action返回逻辑视图名，Struts根据逻辑视图名将请求转发到对应的物理前端（JSP）。

### 3.7.2 配置结果

使用<result>配置结果，<result>元素分局部与全局；局部配置在<action>作为子元素中，全局配置在<global-results>作为子元素配置；一个action中可以有多个result，result分name逻辑视图名与type指定的结果类型；典型的例子是：

<action …..>

<result name=”success” type=”dispatcher”>

<param name=”location”>/WEB-INF/content/a.jsp</param>

<param name=”parse”>true</param>

</result>

</action>

逻辑视图名是success，结果类型是dispatcher也是默认的结果类型，不指定默认是它，这个结果类型包含2个参数（参数配置都是参数名与参数值成对的）location与parse，不同的结果类型给定的参数是不同的，dispatcher结果类型就是转发请求到JSP页面的，location就是实际视图资源，parse代表是否允许在location中使用OGNL表达式，默认是true的，所以可以不写这个参数；简化法：

<action …..>

<result name=”success” type=”dispatcher”>/WEB-INF/content/a.jsp</param>

</result>

</action>

由于dispatcher是默认的结果类型，所以也可以不写：

<action …..>

<result name=”success” >/WEB-INF/content/a.jsp</param></result>

</action>可以通过修改配置文件修改默认的返回类型，省略type就用默认返回类型代替，也可以省略name，因为默认返回success字符串，没给出就是success了。

<action …..>

<result >/WEB-INF/content/a.jsp</param></result>

</action>

如果没给出location参数，result将把<result ></result>之间的资源作为视图资源。等等没给出的都是采用默认值代替。

### 3.7.3 Struts2支持的结果类型

结果类型决定了Action处理完后，下一步将调用哪种视图资源来呈现处理结果；Struts2的结果类型要求实现Result接口，自定义需要实现该接口并在struts.xml中配置，struts2的结果类型定义在struts-default.xml中，拿dispatcher来说<result-type name=”dispatcher” class=”…….ServletDispatcherResult” default=”true”/>，default表示默认的结果类型，name表示返回结果类型名字，class是实现类；struts-plugin.xml也会定义结果类型；struts-default.xml定义好的结果类型包括：chain（Action链式处理）、dsipacther(指定使用JSP作为视图的结果类型)，freemaker（指定FreeMaker作为视图的结果类型）、httpheader、redirect（直接跳转到其他URL的结果类型）、redirectAction（直接跳转到其他Action）、stream（返回流）、plainText（文本文件）。

### 3.7.4 plainText结果类型

plainText直接返回源码文本文件内容，如果文本包含中文字符，需要指定charSet参数，plainText的参数location与charSet。

### 3.7.5 redirect结果类型

Redirect：与dispatcher对应，dispacther是将请求forward到JSP，redirect是重定向，请求参数与属性都会丢失，实际是调用了HttpServletResponse的sendreDrect(String) 方法，产生新的请求。页面的路径也会变，包含的参数与dispatcher相同；注意，结果物理资源不能直接给出/WEB-INF/这种，因为这是受保护资源，重新请求需要给出访问的那种路径。

### 3.7.6 redirectAction结果类型

redirectAction：与redirect一样，重新生成请求，不过这个请求是重定向到action的，这个结果使用ActionMapperFactory提供的ActionMapper来重定向请求，参数name：action逻辑名，namespace，命名空间；请求属性等也会丢失。

### 3.7.7 动态结果

前面的配置action的通配符也可以用在result中，在物理资源视图中使用｛N｝动态生成字符，然后形成最终的资源视图。

### 3.7.8 Action属性值决定物理视图资源

还可以使用OGNL表达式，可以使用${0},${属性名}的方式动态生成物理视图资源，属性名必须是处理action里面定义的同名属性，属性名可以分层；否则返回null。

### 3.7.9 全局结果

全局的result配置在<global-results>中，对所有的Action都有效，如果全局结果与局部result都定义相同的返回name，则首先用局部name的result，局部result没有才去使用全局结果。记住是对所有action都有效的。

### 3.7.10 使用PreResultListener

还可以使用PreResultListener，一个监听器接口，在Action处理完，系统转入实际物理视图资源前被回调，向Action、拦截器添加PreResultListener监听器，通过ActionInvocation的addPreResultListener()完成，事件发生后回调接口的beforeResult方法，一旦为拦截器定义了PreResultListener。该监听器会对这个拦截器拦截的所有Action都起作用。可以用来生成日志。

## 3.8 配置Struts 2的异常

Struts2程序会出现异常，可以在execute方法中手动捕捉异常然后返回特定的逻辑视图名显示特定的异常信息，但这样繁琐，耦合性高；更好的方法是抛出异常，然后让strust2容器来处理异常。

### 3.8.1 Struts2的异常处理机制

strust容器处理异常需要在strust中配置异常映射；就是出现什么异常转到什么视图资源那里；首先要在strust.xml中打开异常异常映射功能，需要一个拦截器与拦截器栈；struts-default.xml中已经开启了异常的拦截器与拦截器栈；拦截器：

<interceptor name=”exception” class=””>拦截器

<intercepter-stack name=”defaultStack”>

<interceptor-ref name=”exception”>

</interceptor-stack>

### 3.8.2 声明式异常捕捉

异常映射是通过在strust.xml中配置<exception-mapping>完成的，需要指定exception指定java异常类、result2个属性（返回的逻辑视图名）；<exception-mapping>也分为全局与局部，局部定义在action内部，逻辑视图名可以是这个局部的或者是全局的；全局的定义在<global-exception-mappings>元素内，跟result类似；全局的对所有action都有效，如果出现相同的异常，局部的会覆盖全局的；转向的逻辑视图名对应的物理视图后，可以在这个视图中显示异常信息，使用标签表示，比如<s:property value=”exception”>或者<s:property value=”exception.message”>输出异常信息，<s:property value=”exceptionStack”>输出异常栈信息。

## 3.9 Convention插件与约定支持

从Struts2.1开始，Struts2引入Convention插件支持零配置，不需要strust.xml与注解进行配置而是根据约定来配置，也就是约定优于配置，加入Convention插件后，Convention会搜索action、actions、struts、struts2包下的所有的java类，这些类中实现了Action接口与名字结尾是Action的java类都作为Action控制器；convention插件的struts-plugin.xml中允许配置3个常量属性，exclude.packages，指定不搜索哪些包下的java类，locators，使用该常量指定的包作为搜寻Action的根包，这个包下面还有其他包一样搜索，action.packages指定哪些包作为根包来搜索Action，至搜索该包下的Action类，不搜索子包。找到action后会按照约定部署Action，部署时action、actions、struts、struts2、packages、locators包会影射成根命名空间/，这些包下的子包映射成相应的子空间，action的逻辑控制器名根据action类名生成，遵循去掉Action结尾，没有则不去掉，将去掉后结尾的驼峰命名法的名字变成小写，单词之间使用中划线-分割；比如：

Actions.books.Getbooks,映射的空间为/books,逻辑视图名为get-books。

Convention也为逻辑视图名与实际的物理视图资源提供了约定规则，默认情况下，Convention总会到/WEB-INF/content路径下定位物理视图资源，定位资源的约定路径是/WEB-INF/content/action所在命名空间/actionName-resultCode.suffix，如果找不到就找actionName.suffix，比如action.user.LoginAction返回success时，约定生成的路径是/WEB-INF/content/user/login-success.jsp或者login.jsp；

可以查看Action、result等资源的配置情况（不论xml、注解还是约定配置），使用插件struts2-config-browser-plugin；重启应用，访问Web\_context/config-browser/index.action就可以看到配置情况了；进入后默认列出默认命名空间下的配置；然后会给出其他的不同的空间。

如果想要重定向到第二个Action处理，约定的规则是第一个Action返回的逻辑视图名没有物理资源，第二个Action与第一个Action处在同一个包下且逻辑名称规则是firstActionName-resultcode，所以直接Action命名规则是FirstResultAction.java，比如action.FirstAction,处理完后返回逻辑视图名second，寻找/WEB-INF/content/first-second.jsp or first.jsp，不定义，就会找逻辑控制器first-second，约定的生成这个逻辑Action的Action名字是FirstSecondAction.java。假如修改了Action或者JSP名字等，Convention插件是不知道的，需要重新加载映射，可以设置自动加载，在.xml或者.properties中配置struts2的常量属性：

<constant name=”struts.devMode” value=”true”>

<contant name=”struts.convention.classes.reload” value=”true”>。

Convention只是按约定配置action，result映射，Filter需要在web.xml配，全局配置等也需要配置在.xml中，全局变量这里忽略不写。

## 3.10 使用struts2的国际化

Struts2的国际化与java相同，都是根据环境信息加载有特定资源文件，完成国家化；步骤：1.加载国际化资源文件（自动加载，Action、包；手动加载，JSP，全局范围），使用标签即配置常量的方式加载，2.输出国际化，视图使用标签或者Action使用ActionSupport的getText()方法。视图资源的国际化使用<s:i18n>标签完成，一些html组件的key属性会显示资源文件中指定的值。在Action中加载资源文件是在这个路径下定义一系列ActionName\_language\_country.proper ties，Action可以自动加载，在JSP页面中输出使用<s:text>标签，name属性指定了资源文件中的key，label中的key指定key，getText(name)name指定了资源文件中的key，在Action的规则校验文件中使用<message>标签访问资源文件中的key。包范围的国际化资源与Action基本相同，放在该包的目录下，命名规则是

Package\_<language>\_<country>.properties，该包下所有的Action都可以访问；还可以做成全局资源文件，定义在classes文件夹下，在struts.roperties或者struts.xml中通过常量指定资源文件名字，在所有范围内都可以使用，是有那个方法跟上述是一样的，国际化消息可以包含占位符，｛N｝这种形式，需要在使用资源文件的key时传入相关的参数来代替占位符，在JSP层使用<s:param>标签传参，在Action层，使用String数组或者String的list数组传参。也可以直接使用先关页面内的表达式变量值代替占位符，当在本页面或者Action使用时，自动将变量值替换，不用传参${变量}。加载资源文件的先后顺序是：Action：1.类文件相同位置且basename相同的；2.加载父类的；3.加载接口的4.找包下的；5.顶层包6.struts2常量指定的7。如果还找不到，直接输出。JSP：1.找<s:i18n>指定的；2.长struts常量配置的；3.直接输出。以上找到了即停止搜索。

## 3.11 使用struts2的标签库

Struts2标签不依赖表现层技术，是通用的；定义在struts-tag.xml文件中，分为3个类别：UI（生成HTML）、非UI（数据访问，逻辑控制）、Ajax用于Ajax支持，UI标签又分为：表单标签，非表单标签，非UI标签分为：流程控制标签，数据访问标签。使用Struts2的标签就是先导入，定义前缀然后使用。Struts2使用OGNL语言（Object Graph Navigation Language，对象图导航语言），访问数据；OGNL表达式语言就是为了直接访问Action里面对象的属性值什么的；封装好了，可以直接访问，OGNL中存在一个Stack Context对象，是一个map类型，map中含有Valuestack对象map，这是根对象；此外还包括parameters、request、session、application、attr等对象，访问跟对象不需要加#，访问其他对象需要加#；给出属性名，就在Stack Context中自上而下查找，找到了就获取这个数据。可以在页面中添加<s:debug>标签生成属性名字的链接。OGNL可以床架list与map集合，还可以使用in not in判断是否在集合中，使用一些操作符完成集合的操作。还可以访问静态成员，在常量配置中开启访问静态方法的功能，使用如下格式在JSP中访问：

@classname@staticfield或者@classname@staticMethod(val)；还可以使用lambda表达式来完成简单函数的功能。

控制标签：完成流程控制：

1. If、elseIf/elseif、else，类似于开发语言中的语法，但是使用test=”boolean”的表达式判断真假的；
2. Append：将多个集合拼合组成一个集合。
3. Generator:genertator标签用于将整个字符串按分隔符分成字符串数组，这个集合可以使用iterator迭代输出，临时生成的集合位于ValueStack的顶端，属性如下：count，指定生成的集合的总元素数，separator，指定分隔符，必填的属性，val，指定解析的字符串，converter，转换器，将元素转换成特定对象，var/id,指定集合放入到Stack Context中。
4. iterator，对集合进行迭代输出，属性value，指定集合，id，status等。
5. merge
6. sort
7. subset。

# 第四章 深入使用Struts2

本章将要介绍的知识：类型转换、校验支持、拦截器、Ajax与文件上下传。

## 4.1 详解Struts2的类型转换

MVC框架负责解析HTPP请求参数，并传给控制器组件，MVC框架负责将请求参数的字符表达形式转换成java的对象类型。Struts2提供了非常强大的类型转换机制，这种类型转换机制可以基于OGNL，只要HTTP的请求参数满足OGNL表达式形式就可以自动转换，也可以自定义类型转换机制，并且不用担心异常，Struts2会拦截异常信息。

### 4.1.1 Struts2内建的类型转换器

Struts2本身可以完成大部分的类型转换，内建字符类型与java对象的转换规则如下：boolean->Boolean、char->Character、int->Integer、long->Long、float->Float、double->Double、Date->Date、数组：默认是字符串数组，集合：默认集合为String类型的ArrayList，如果HTTP请求参数是以上的形式则无需处理，能够自动转换。

### 4.1.2 基于OGNL的类型转换

借助OGNL可以完成简单的字符串组合成复合对象。比如Action中包含User对象，提交的名字可以写成user.name与user.password的形式就组合成了User对象，这种方式中User要有无参的构造函数用于反射生成，必须提供setUser、setName等set方法；也可以生成Map<String,User>或者List<User>这种更高级的组合类；比如：Name=”users[‘one’].name”与Name=”users[‘one’].pass”，生成了<one,User(name,pass)>的map对象；Name=”users[0].name”与Name=”users[0].pass”生成了List的User组合类。上述都在JSP中可以访问属性，访问的方式也是相同的。

### 4.1.3 指定集合元素的类型

上面在定义List或者Map时使用了泛型，可以让Struts2知道类型信息进而完成转换，如果不指定泛型信息，就要指定转换规则文件，总之要指定转换类型信息，在需要转换的Action同目录下创建properties文件，。名字是ActionName-conversion.properties,只要涉及到类型就要写一个属性，比如list中只包含一种数据的类型，那么写法是：

Element\_<ListPropName(变量名)>=<ElementType(类型的全名)>

Map则是：

Key\_ <MapPropName变量名>=<KeyType(类型的全名)>

Element\_ <MapPropName变量名>=<ValueType(类型的全名)>。这种也叫做局部类型转换文件。

### 4.1.4 自定义类型转换器

如果默认的类型转换机制与OGNL不满足要求，可以自定义类型转换器；类型转换器要实现TypeConverter接口，在接口中实现一个转换方法，也可以选择继承DefaultTypeConverter类，它已经实现了TypeConverter接口，重写了convertValue方法，public Object convertValue(Map context,Object value,Class toType),toType是要转换的类型Class类对象，value是对象值（当把字符串向对象转化时，value是字符串数组，当反方向时，value是对象），context是类型转换的上下文，convertValue类型转换时双向的，所以要函数中要定义双方向的类型转换规则，toType的类型决定了转换的方向，返回值是转化后的值，为了通用性，比如很多一个name具有很多值，所以统一提交的字符串的所有的请求参数都视为字符数组，如果是单一值的转换为大小为1的字符数组。因为转换器是通过HttpServletRequest的getParameter(name)方法获得属性值得。

### 4.1.5 注册类型转换器

定义了类型转换器还需要注册，以便知道什么时候进行类型转换，注册类型转换器有3种：局部类型转换器对某个Action的某个属性起作用，全局类型转换器（对所有Action的特定类型起作用），使用注解注册类型转换器。

局部类型转换器：在该类下的类类型转换文件中指定<propNam e>=<converterClass>，propName这个是需要待转的属性，converterClass这个是转换的类处理，这种处理方式只对该action有作用，局限性太大；

全局类型转换不对特定action起作用，而是对特定的类型起作用，它是对所有action都起作用的，在WEB-INF/classes路径下提供xwork-conversion.properties文件，里面写上<propName>=<converterClass>即可，有一点说名，局部转换只根据action的属性名进行一次转换，所以不论是数组还是什么都是执行一次转换函数convertValue()，全局是根据类型的，如果出现数据或者list，有多少转换多少次类型。

### 4.1.6 基于Struts2的自定义类型转换器

Struts改进了继承的DefaultTypeComverter类型，提供了StrutsTypeConverter类型，这个类型简化了类型转化，StrutsTypeConverter实现了convertValue()方法，厘米昂会根据转换的方向调用2个方法：convertFromString与convertToString，继承后时重写这2个方法就可以控制具体的转换逻辑了，同时也更加明了。

### 4.1.7 处理Set集合

Action中使用Set集合属性，由于处于无序的状态同时又没有标识，所以不能准确的读取转换特定元素，可以通过对Set元素建立标识的方法来存取元素，这类似于Map的处理方式，类型转换部分是传统的部分，但是在类型内部要重写equals与hashcode方法，确立标识部分的唯一性，在类型转换配置文件写上KeyProperty\_<SetPropName>=<keyPropName>,放在set属性的类型转换器后，keyPropName就是set中元素中的属性标识，因为是唯一性标识，所以在元素中要根据这个标识重写equal方法，在View层可以像Map那样使用集合元素，只是[key]变成了(key)。

### 4.1.8 类型转换中的错误处理

类型转化的要求是非常高的，用户的输入可能不满足类型转换的条件，此时就需要异常处理，struts2提供了一个conversionError的拦截器，被注册在拦截器栈中，如果执行类型转换失败，该拦截器将错误封装成表单域错误（FieldError），并将错误信息放入ActionContext中，它是异常处理逻辑，只负责处理异常。

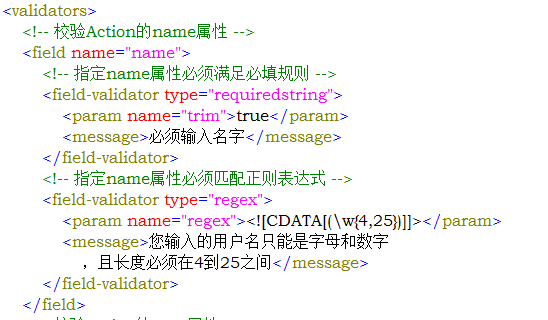
要使用这个异常处理逻辑，Action必须继承与ActionSupport基类，因为ActionSupport实现了收集类型转换错误，并将他们添加到ActionContext中的操作，处理完异常后，Action返回input逻辑视图名，可以在result中配置一个物理视图资源。在视图页面，使用struts的<s:fielderror>标签来输出错误信息。中文信息需要在国际化资源文件中指定。

## 4.2 使用struts 2的输入校验

输入校验主要防止误操作或者系统攻击，分为客户端与服务端校验，客户端进行简单校验，服务端进行深度校验，判断输入数据是否满足要求的格式。

### 4.2.1 编写校验规则文件

编写校验规则文件：struts2提供了基于验证框架的输入校验，所有的输入校验编写简单的校验配置文件就可以了。校验配置文件是一个xml文件，跟所需要校验属性的Action处于同一目录，并且名字形式为actionname-validation.xml的形式，校验文件的根元素是<validators>，里面包含了很多的<field>（字段校验器风格）或者是<validator>(非字段校验器配置风格)，<field>元素还包括<field- validator>元素，里面定义了具体属性的验证规则。图示如下：



验证失败通过action返回逻辑视图名input，这里跟类型转换的异常处理就想通了，可以通过标签输出错误信息，他们都被封装称为FieldError放入ActionContext中，如果使用struts2的表单标签定义表单，则会自动输出错误信息。

### 4.2.2 国际化提示信息

国际化提示信息：验证文件中的message是输出的验证失败信息，是硬编码的，不利于国际化，可以在message中指定一个key属性，内容为国际化资源文件中对应的key，这样输出的信息就显示为国际化资源文件中的信息。

### 4.2.3 使用客户端校验

使用客户端校验：客户端校验分2步：1.表单标签使用struts2的标签来定义；2.在<s:from>中写上validate=”true”；客户端校验具有很大局限性，实际上也是javascript实现的，支持的校验器包括：required validator（必填）、等等；4个注意的地方：<s:from>中的theme属性不能是simple、不能直接访问视图页面，可以通过filter转发，必须使用全局国际化资源文件来输出信息，提交的action中，namespace部分要拿出来单独写。

### 4.2.4 字段校验器风格

字段校验器配置风格：字段优先，使用field配置，每个field对应action中的一个属性，<field-validator>对应一个校验规则，type属性指定了校验器名，可以包含多个<param>与一个<message>。

### 4.2.5 非字段校验器配置风格

非字段校验器配置风格：是校验器优先的一种配置方式，包含<validator>元素，每个<validator>元素的type指定一个校验规则，包含多个<param>元素与一个<message>元素，param元素里要有一个name=“fieldname”的元素，指定需要校验的属性，其他任选。

### 4.2.6 短路校验器

短路校验器：没有短路校验，所有的校验规则都会起作用，往往会输出很多的错误信息，在<field-validator>或者<validator>元素中指定short-circuit=“true”让校验规则支持短路校验，只输出规定的错误信息。

### 4.2.7 校验文件的搜索规则

校验文件的搜索规则：一个物理Action可能对应多个逻辑Action，多个逻辑Action可能对Action里面的属性要求不一样。此时规则文件的作用如下，可以根据逻辑action的名字定义一个规则校验文件，里面可以定义方法要求的属性校验规则，名字定义如下：<ActionClassname>-<ActionaliasName>-validation.xml，ActionaliasName为逻辑Action名字，假设类RegistAction继承与BaseAction，那么搜索规则文件的顺序是：baseAction-validation.xml、BaseAction-逻辑Action-xml、RegistAction-validation.xml、RegistAction-逻辑action-validation.xml，校验的规则是上述规则的总和，如果规则冲突，后面的规则取胜，总之名字符合的都会进行处理。

### 4.2.8 校验顺序与短路

校验顺序与短路：校验器的执行顺序如下：非字段风格的校验器优于字段风格的校验器，非字段校验器中，排在前面的先执行，字段校验器中排在前面的先执行，短路规则如下：非字段校验器风格中的一个校验失败，不会影响其他非字段校验器的执行，但是字段校验器则没有机会得到执行，字段校验器一个失败，其后面的校验器不会得到执行。

### 4.2.9 内建校验器

内建校验器：struts2提供了大量的校验器，可以直接使用，如果我们自己的校验法则具有通用性，可以开发作为一个校验器，struts2提供的校验器注册在xwork-2.3.jar包中的default.xml文件中，里面定义了框架使用的校验器，<validators>作为根元素，<validator>作为基本元素，属性name指定校验器名称，class执行校验器的实现类，开发自己的校验器需要提供一个类似这个的validators.xml文件放在/WEB-INF/classes文件夹下，应用启动时找到了validators.xml，则不会再加载default.xml，需要先把default.xml的内容复制到validators.xml。内建的校验器有：

1. 必填校验器：名字为required，要求指定字段必须有值；
2. 必填字符串校验器：requiredstring，要求字段必须非空且长度>0；
3. 整数校验器：int/long/short，要求整数值必须在指定范围内；
4. 日期校验器：date，字段的日期必须在指定范围内；
5. 表达式校验器：expression，非字段校验器，要求OGNL表达式返回true，否则校验失败；
6. 字段表达式校验器：fieldexpression，要求指定的字段满足一个逻辑表达式；
7. 邮件地址校验器：email，非空且符合邮件地址；
8. 网址校验器：url，合法的url地址；
9. Visitor校验器：主要用来校验复合属性，在context属性中指定复合属性的规则校验文件；
10. 转换校验器：conversion，检查校验字段在类型转换过程中是否出现错误；
11. 字符串长度校验器：stringlength，要求字段长度在指定范围内；
12. 正则表达式校验器：regex，是否匹配正则表达式。

### 4.2.10 基于注解的输入校验

可以使用注解来定义每个字段应该满足的规则，struts2的注解包中开发了字段验证的注解，注解用来修饰setter方法。

### 4.2.11 手动完成输入校验

手动完成输入校验：struts2的校验器规则固定，无法验证一些特殊的逻辑，所以可以使用手动方式完成自定义校验，重写继承的ActionSupport类的validate()方法，里面调用addFieldError方法完成同样的处理，如果要使用国际化的信息资源使用getText()方法，也可以加一个方法validateXxx()方法，物理Action分为很多逻辑Action，每个方法对应一个，每个方法要求的属性规则各不相同，此时validate()对这些方法全部起作用，所以为了灵活性，建立一个方法validateXxx()，其中Xxx是逻辑Action名；其中validateXxx()首先调用，然后调用validate()。输入校验分为以下几个步骤：1.对请求参数进行类型转换，并设置Action属性值，2.转换出现异常将异常保存在ActionContext中，vonversionerror负责将其封装在FieldError中3.使用校验器进行校验，先执行validateXxx()，在执行validate()；4.调用处理逻辑。

## 4.3 使用struts2控制文件上传

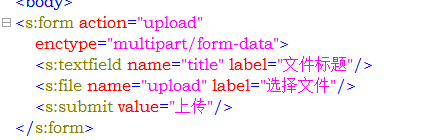
上传文件爱你需要将表单域的method设置为POST，将enctype设置为multipart/form-data(采用二进制流的方式处理表单数据)，只有这样才会将用户选择的文件以二进制数据发送到服务器。

### 4.3.1 Struts 2的文件上传

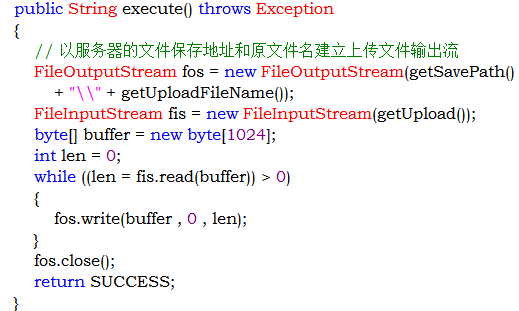
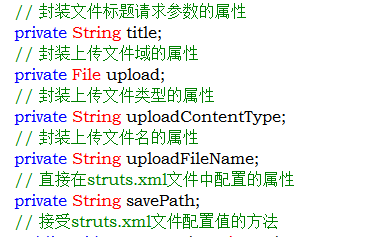
Struts2并未提供解析multipart/form-data类型数据的的模块，它集成与封装了其他框架，在struts.properties文件中可以配置解析器，key是struts.multipart.par ser变量，可以配置cos、pell或者jakarta解析包。由于封装了这些解析器，所以使用起来是透明的。

### 4.3.2 实现文件上传的Action

简单的文件上传代码如下：可以看到表达域定义的文件上传类型，



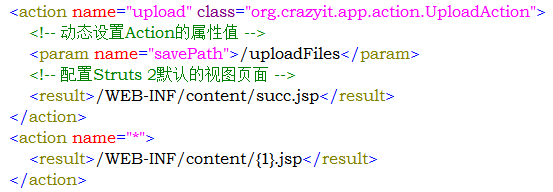
Struts2的action中的属性用来存储文件信息，这些信息由struts2框架解析请求参数得来，处理的Action部分的代码如下：



可以看到Action的4个属性，title与upload分别用来对应表单域的name属性，后面2个是框架解析的参数，分别代表文件名字与文件类型，因为upload是File类型的，只封装了文件内容，从File类型不能获得上传的文件名字与文件类型，所以框架解析出这2个参数，后面的3个名字是有关联的，

### 4.3.3 配置文件上传的Action

savePath变量是在配置逻辑action时使用<param>配置的值注入的服务器的文件保存路径



实际应用中这种方式上传的文件名没有发生变化，高并发时容易发生名字冲突，建议要重命名以生成唯一的名字。

### 4.3.4 手动实现文件过滤

文件需要过滤，比如对文件类型，大小的限制等，文件类型封装在action中，大小为File类型变量的length方法，可以在Action的validate()方法中，完成校验工作；假如这2个不符合要求，就使用addFieldError方法向域添加错误信息，并返回input逻辑视图名。

### 4.3.5 拦截器实现文件过滤

手动过滤太过于麻烦，可以配置拦截器实现文件过滤，在Action配置中配置fileUpload拦截器与相关的拦截器栈，配置文件的形式如下所示：



指定参数有allowedTypes，允许的上传文件类型，多个使用逗号分隔，maximumSize指定上传文件的最大大小。

### 4.3.6 输出错误提示

校验失败或者上传失败，框架返回input逻辑视图名，里面的失败信息可以使用国际化资源文件来产生提示信息。

### 4.3.7 文件上传的常量配置

Struts2在文件上传的过程中要指定一个临时目录，用于存放临时文件，默认是找key=struts.multipart.saveDir这个变量的value，如果没有就使用javax.servlet.context.tempdir这个目录。

## 4.4 使用struts2控制文件下载

Struts2提供的stream结果类型用于支持文件下载。

### 4.4.1 实现文件下载的Action

直接在界面的href处给出文件连接，不能下载中文文件，同时也不方便，Struts2的框架的文件下载添加了对于中文文件名与授下载等控制逻辑。在Action配置中写一个返回InputStream流的方法，这个流就是目标文件的入口。

### 4.4.2 配置Action

配置Action主要是配置一个一个类型为stream的result元素，其中的inputName用于指定Action里的流方法名，contentType指定文件类型，contentDisposition指定文件名，bufferSize指定缓冲区大小，如下图所示：



### 4.4.3 下载前的授权控制

可以在用户下载文件前，先判断Session里的user的权限。

## 4.5 详解Struts2的拦截器机制

拦截器是struts2的重要的组成部分，完成了大部分的操作，params拦截器负责解析请求参数并设置Action属性，servlet-config拦截器将Request与Responese实例传给Action，fileUpload负责解析文件域并将文件域设置成3个Action属性。拦截器是可插拔式的，使用某个拦截器，直接在struts2的配置文件中配置，不使用，直接取消。

### 4.5.1 拦截器在struts2中的作用

MVC框架都会完成一些通用的控制逻辑，在struts1时代，核心控制器中会进行全部所有的解析请求与处理请求的操作，这种形式太死板，不利于拓展，有些拦截处理可能还没有用，所以struts2时代按功能改成单个的拦截器，需要哪个在配置文件中加哪个，在strutsPrepareAndExecuteFilter拦截用户请求后，初始化一个ActionProxy实例，然后大量拦截器处理它，再然后转到Action处理，处理后根据返回的逻辑视图名返回物理试图，如下图：



Strust2已经在struts-default.xml中默认定义了很多拦截器，集成该包后，这些拦截器发挥了作用。

### 4.5.2 Struts2内建的拦截器

Struts内建了很多的拦截器，都在struts-default包中，继承它就会使用这些拦截器，没有继承就要自己定义。内建拦截器的定义是name-class形式，class是拦截器类，name是定义的逻辑名字；这里内建的拦截器看书，有很多很重要。

### 4.5.3 配置拦截器

定义拦截器很简单使用<intercepter>元素，里面有属性name及class。如果需要传入参数则在里面加上<param>参数，也可以组合成拦截器栈，使用元素<intercepter-stack>元素定义拦截器栈，里面使用元素<intercepter-ref>元素将定义好的拦截器引入到拦截器栈中，拦截器栈里也可以包含拦截器栈，与引入拦截器的方法相同，所以拦截器栈与拦截器是统一的，区别只是一个是每次一个发挥作用，一个是多个拦截器发挥作用；指定拦截器的参数时有2种时机：1.定义拦截器时，这时是作为默认的参数值；2.使用拦截器时。

### 4.5.4 使用拦截器的配置语法

在配置action里面加入拦截器的引用，引入方法与定义拦截器栈时是一样的，都是使用<intercepter-ref>元素，这样拦截器就能对特定的action发挥作用了。

### 4.5.5 配置默认拦截器

配置包时可以指定一个默认拦截器（拦截器栈），该包中的以及子包中的Action配置中如果没有指定拦截器，则默认的起作用，如果显示指定了一个，则默认的不起作用，除非手动加上拦截器的配置。一个包只能指定一个默认拦截器配置，配置默认拦截器使用<default-interceptor-ref>元素，指定的name属性是拦截器(栈)的名字，这个在<package>元素下面定义。

### 4.5.6 实现拦截器类

自己开发拦截器，实现接口Intercepter；包含3个方法：inti（实例化后，拦截处理前，回调执行，用于初始化），destroy（销毁前回调执行），intercept(ActionInv ocation invocation)（拦截处理操作），会返回一个字符串作为逻辑视图名，此时将直接转向对应的物理试图，ActionInvocation残苏包含了被拦截的Action引用，调用参数对象的invoke方法转向下一个拦截器或者Action处理；也可以继承AbstractInterceptor类，这个类给出了init与destroy的空实现；参数invocation可以获得拦截的Action实例；并可以用invoke方法将控制权转发到下一个拦截器，此时获得了Action实例，可以处理任何事情。

### 4.5.7 使用拦截器

首先定义配置拦截器实现类，然后将拦截器引入到action中，这种编程思想是AOP。

### 4.5.8 拦截方法的拦截器

如果为某个Action定义了拦截器，则拦截器会拦截该Action的所有访问，如果想要只拦截特定的方法，那么需要拦截器继承MethodFilterInterceptor，该类提供了ExcludeMethods与IncludeMethods属性与响应的set/get方法，分别是设置不拦截的方法与拦截的方法名的，还重写了interceptor(ActionInvocation)方法，这个方法实际是拦截所有的Action访问的，但是根据上面的2个属性里的值，在Include里面则调用doIntercept方法处理，如果在exclude里则转向下一步。所以，我们只需要自己实现doInterceptor方法就好了，可以通过在配置Action时配置拦截器时指定名为excludeMethod与includeMethod的属性来设置过滤的方法与拦截的方法，多个方法用逗号分隔，冲突时，以include为准。

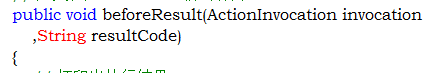
提供方法过滤的拦截器：TokenInterceptor、TokenSessionStoreInterceptor、DefaultWorkflowInterceptor、ValidationInterceptor。

### 4.5.9 拦截器的执行顺序

配置的拦截器的顺序遵循栈的原理，先定义先拦截，但最后退出；后定义的先退出；拦截器栈与Action中配置拦截器都可以一个拦截器配置多次。

### 4.5.10 拦截结果的监听器

Struts2还具有拦截结果的监听器，在转向物理资源之前执行拦截Action方法返回的逻辑视图名；执行拦截结果的监听器的接口是PreResultListener，在Action中注册该监听器，那么只对这个Action有效；如果把PreResultListener注册在拦截器中，那么可以用于很多的Action，ActionInvocation参数对象有一个addPreResultListener()方法，可以为拦截器加入结果监听器，接口中有一个beforeResult()方法。参数如下：



在方法内部可以对处理结果做针对性的处理；虽然有参数ActionInvocation，但是不要调invoke方法，因为它会再次执行Action的execute方法并在次回到其本身来，这样就会陷入死循环。

### 4.5.11 覆盖拦截器栈里特定拦截器的参数

有时Action需要一个拦截器栈，但是又需要改变拦截器栈中某个拦截器的属性的值，此时需要在配置Action的<Intercepter-ref>元素的<param>的name属性中写上<拦截器名>.<属性名>的形式，此时就可以覆盖拦截器的属性的初始值。

### 4.5.12 使用拦截器完成权限控制

权限检查。

## 4.6 使用struts2的Ajax支持

Ajax，Asynchronous JavaScript And XML；异步JavaScript与XML技术，浏览着与服务器采用异步通信，加载页面了还可以通信，此时将通信的结果通过DOM(Document Object Model)技术加载到页面所在的容器内，传统应用中，每个页面由于一次请求或者一次响应，所以页面的使用时间很短，页面也做的很简陋，因为传输的东西太多利用率低，Ajax就是异步加载响应的。

### 4.6.1 使用stream类型的Result实现Ajax

Struts2支持一种stream类型的result，可以直接向客户端返回二进制的数据，通过配置Ajax，返回stream数据，直接显示在页面上。

### 4.6.2 JSON基本知识

JSON，JavaScript Object Notation，JavaScript对象符号，语言无关的数据交换格式定义，JSON的主要的数据结构式key-value键值对对象使用｛｝包围，以及数组使用[]包围；JavaScript使用专门的JSON语法创建对象是十分方便的，json2.js提供了一个全局的JSON对象，包含stringify（将对象转换为JSON字符串）和parse（将字符串转换为对象）方法。

### 4.6.3 实现Action逻辑

JSON插件使用@JSON注解改变返回的JSON对象的名字。

### 4.6.4 JSON插件与JSON类型的Result

JSON插件提供了一种名为json的result，一旦为某个Action指定了返回结果是json，则无需指定物理试图资源，json插件直接将action内的属性信息序列化成json字符串发送给请求的页面。Json类型的Result配置允许指定非常多的参数，这些参数都很重要，要看书。

### 4.6.5 实现JSP页面

必须指定回调函数用来处理返回的数据。

# 第五章 Hibernate的基本用法

Hibernate是持久层解决方案，完成对象模型到基于SQL的关系模型的映射。低侵入式设计，将OOA、OOD、OOP整合。

## 5.1 ORM和Hibernate

### 5.1.1对象关系数据库映射

ORM/object Relation mapping 对象关系数据库映射，是桥梁；是一种用于描述面向对象与关系型数据之间映射的规范，这个规范目前是java ee的JPA规范（定义好的接口，所以只要是符合JPA规范的ORM框架都可以随便替换）；ORM框架将面向对象的操作转化为底层的SQL操作。

### 5.1.2 基本映射方式

基本的映射方式有表映射、行映射以及列映射。应用程序只需关心持久化对象，不需要关注底层的关系数据库操作，Hibernate已经做了所有的事情。

### 5.1.3 流行的ORM框架

JPA、Hibernate、Mybatis、TopLink。

### 5.1.4 Hibernate概述

## 5.2 Hibernate入门

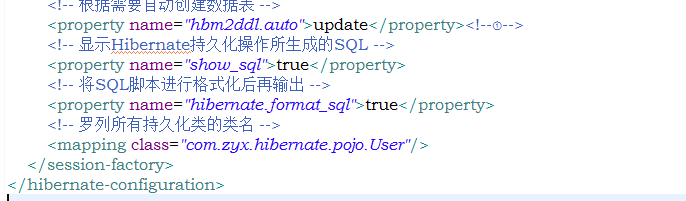
### 5.2.1 Hibernate的下载与安装

Hibernate的jar包下载。

### 5.2.2 Hibernate的数据库操作

Hibernate的PO=POJO+持久化注解；POJO类用于代表数据库表里字段的定义等信息，低侵入式代码，为该类田间注解@Entity与@Table(name=”表名”)2个注解，指定跟哪个表相关联，在具体字段上可以指定@Id等注解，标识哪个是ID栏位，在XML文件中指定连接的数据库的原信息，这些信息可以配置在properties文件里面也可以是XML文件里面，XML内容如下：





配置文件的默认文件名是hibernate.cfg.xml，调用Configuration对象的configure()方法时将会加载它，可以在上面的图片里看到元素信息，Hibernate还推荐使用数据源的方式管理数据库，也就是c3p0。hbm2ddl.auto指定是否根据持久化类自动创建数据表,<session-factory>元素里面还包括<mapping>元素，每个可以指定一个映射的class类，进行持久化操作的步骤：开发持久化类、获取Configuration、获取SessionFactory、获取Session打开事务、用面向对象的方式操作数据库，关闭事物、关闭session。示例如下：



映射类有3种状态：瞬态，PO从未与Session关联过；持久化，与Sessio关联；脱管，曾经关联过，现在脱离了关联，脱管状态。PO操作要在session中同步数据库，SeesionFactory是数据库编译后的内存镜像；一个应用对应一个Factory，可以生成Session，Factory由Congiguration对象生成，还负责加载配置文件。

### 5.2.3 在Eclipse中使用Hibernate

开发Hibernate应用可以借助工具，只是提高了开发效率；可以先安装Hibernate tool插件工具包；然后在build path里添加类路径；然后在src目录下添加hibernate.cfg.xml文件，里面可以可视化的编辑，也可以文本编辑；如果希望显示sql语句那么加入hibernate.show\_sql属性，自动建表就加入hibernate.hbm2ddl.auto属性；开发POJO文件，添加注解，将pojo添加到hibernate.cfg.xml配置文件中去，然后在新建的主程序中调用，最后看结果。

## 5.3 Hibernate的体系结构

Hibernate的架构如图：



SessionFactory：这是Hibernate的关键对象，是单个数据库映射关系经过编译后的内存镜像；

Session：交互操作的单线程对象，封装了JDBC底层的连接，是Transaction的工厂，所有的持久化对象必须在session管理下才可以进行持久化操作，在显式进行flush之前，所有的持久化操作的数据都缓存在session对象处。

持久化对象，javabean与特定session关联，对对象的任何修改都将转化为对持久层对象的修改；

瞬态对象/脱管对象，单单指java对象而言，没有与session相关联，也就是没关联到数据库表的数据。

事务，原子操作，数据库概念；

连接提供者，生成JDBC连接的工厂。

## 5.4 深入Hibernate的配置文件

SessionFactory是数据库映射关系经过编译后的内存镜像，该对象由Configuration对象产生，一个Hibernate配置文件对应一个Configuration对象，openSession()方法可打开Session对象。

### 5.4.1 创建Configuration对象

Configuration对象的buildSessionFactory创建SessionFactory对象，也可以使用addAnnotatedClas()或者addPackage()动态的添加持久化类；分为3种方式：

1.使用hibernate.properties作为配置文件，创建Configuration对象直接使用new Configuration()构造方法，这种配置方式需要构造完Configuration对象后手动添加持久化类，麻烦，耦合度高；

2.使用hibernate.cfg.xml作为配置文件，可以在xml文件中配置持久化类，但是new Configuration()构造出Configuration对象后要调用configure()方法加载该XML文件的内容；

3.不使用配置文件，编程方式设置各种属性；主要用到的方法有：

Configuration addAnnotatedClas(),Configuration addPackage(),Configuration setProperties(Properties proprties),Configuration setProperty(String propertyName,String value)。

### 5.4.2 hibernate.properties与hibernate.cfg.xml

实质是完全一样的。

### 5.4.3 JDBC连接属性

连接数据库的相关属性，基本属性要看书，其中一条是连接池的最大并发数采用的是C3P0，这个具有实用价值。

### 5.4.4 数据库方言

数据库之间存在明显的差异以及自己拥有的特性，为了区分连接的是哪个数据库产品，采用什么样的底层操作，需要设定数据库方言，支持的方言可以看书。

### 5.4.5 JNDI数据源的连接属性

JNDI java naming directory interface java命名目录接口，Hibernate不自己管理数据源，直接访问容器的数据源，可以使用容器的JNDI数据源，配置的一些属性可以看书。

### 5.4.6 Hibernate事务属性

需要设定一个属性。

### 5.4.7 二级缓存相关的属性

### 5.4.8 外连接抓取属性

### 5.4.9 其他常用的配置属性

一些很重要的配置属性。

## 5.5 深入理解持久化对象

Hibernate框架使用户只需要关注对象的状态而无需注意底层的SQL实现；优化数据访问的时候需要了解底层的SQL操作。

### 5.5.1 持久化类的要求

持久化类只是简单的java类，这种持久化类也要满足一些要求：

1. 必须有一个无参的构造器，以便反射生成实例，访问修饰符必须>包可见性；
2. 2.必须提供标识属性，对应数据表的主键，联合主键可以采用自定义组合类与注解的方式标识联合标识属性，属性的类型通常是可以是null的对象类型，不建议是基本类型；
3. 每个成员要有gettter，setter方法；
4. 使用非final类，否则无法生成代理子类；
5. 重写hashcode与equals方法，这是根据数据库表数据的标识属性来重写的，因为要判断2行数据是否是同一行，可以使用行的标识属性，但是指定自动生成标识值的持久化类对象不能这么用，因为在save后，标识值发生了变化，同样的2个对象就会影响相等性判断。

### 5.5.2 持久化对象的状态

持久化对象的状态有瞬态、持久化与脱管；瞬态：new的普通对象，持久化状态，与session相关联，任何改动都会由Hibernate同步到数据库，脱管对象，就是session关闭了，持久化对象变成脱管，与数据库之间失去了关联，此时可以对其做任何改动，重新让脱管的对象关联session后，改动不会丢失；状态演化图：



### 5.5.3 改变持久化对象状态的方法

将一个瞬态对象变为持久化对象的方式就是调用save或者persist，都具有同时指定主键值的重载方法，如果标识属性是具有生成策略时，那么save时自动生成并分配给对象，如果标识属性是组合类型的，要自己设置在save前设置。

Save操作返回标识的属性值，同时立即插入数据，persist不立即插入，写入会话流程中，而且没有返回值。

Load/get函数是select操作，根据标识属性来加载一条数据，load方法可以延迟加载，没有的话抛出异常，get不能延迟加载，没有的话返回null。

持有持久化对象后，任何对对象的操作都将后台转化为update操作；提交脱管对象的操作是update、merge（将数据写到数据库返回持久化对象，原对象状态不变）、updateOrsave（判断之前是否持久化过，没有则是save，有则是update）；delete删除持久化实例，也将数据据库的删除。

## 5.6 深入Hibernate映射

Hibernate提供3种方式将POJO变成PO类：

1. 使用持久化注解；
2. 使用JPA2提供的XML配置描述文件；
3. 使用Hibernate传统的形如\*.hbm.xml的XML映射文件。

使用注解修饰POJO建立到数据库对象的映射；需要很多注解，这些注解包括@Entity实体、@Table映射的表、@UniqueConstriant、@Index、@Access、@Proxy、@DynamicInsert、@DynamicUpdate、@SelectBeforeUpdate、@PolymorphismType、@where、@BatchSize、@OptimisticLocking、@Check、@Subselect。

### 5.6.1 映射属性

@Entity修饰的持久化类的所有属性被映射到底层数据表列，可以使用@Column注解修饰该属性指定列信息，里面有非常多的属性，@Access指定属性的访问策略，@Formula指定SQL语句，没有数据列，值为SQL语句计算值，该SQL语句就是普通的SQL语句，如果要使用持久化类里面的变量，可以直接使用，@Generated，指定列值是否由数据库自己生成，value可以接受一些值表明。@Transient修饰不想持久保存的属性，该属性不会映射到数据列也不会保存到数据库中；@Enumerated修饰持久类的枚举属性，可以用value指定保存的方式保存为枚举属性名还是其序号；@Lob修饰大数据量属性，比如byte数组等，@basic修饰属性表明是否延迟加载，因为大数据栏位的读取需要消耗很多时间；@Temporal修饰时间属性，指定将java的日期时间类型转换为数据库的何种时间日期类型，支持的属性值有TemoralType.DATE、TemoralType.TIME、TemoralType.TIMESTAMP。

### 5.6.2 映射主键

主键就是标识属性，不推荐物理意义主键，主要是为了避免数据冗余及减少维护的复杂度，推荐毫无意义的逻辑主键，使用@Id表示主键，Hibernate为主键提供了很多的生成策略，使用@GeneratedValue注解标识主键生成策略，属性有strategy,标识主键生成策略，generator，标识生成器，比如一个触发器什么的。

### 5.6.3 使用Hibernate的主键生成策略

JPA标准注解只支持AUTO、IDENTITY、SEQUNCE、TABLE几种主键生成策略，Hibernate支持更多的主键生成策略，使用@GenericGenerator注解可以定义主键生成器。这里详细支持的生成策略看书。

### 5.6.4 映射集合属性

同一个ID往往对应外部表几条数据，这些数据可能值是相同的，也可能是不同的，所以当需要定义一个实体类，这个实体类包含了多个表的内容时，为了避免冗余的存储，需要定义集合属性或者Map类结构的属性，实体类的集合属性的类型都必须是接口，不能是实现类，因为实现类与Hibernate的内部操作不一定相符合，所以使用接口确保操作的一致性，外表的集合属性的状态跟随主表的状态，集合属性使用@ElementCollection注解，属性有fetch，targetClass等。多个表的话使用外键管理，使用@CollectionTable与@JoinColumn注解，他们都有自己特定的属性，有索引值或者key的列还需要定义索引列，使用注解@OrderColumn与@MapKeyColumn2个列，集合元素的数据类型可以是任意的类型。

List与数组的使用方式如下：



Set与上面基本相同，只是Set里面没有@OrderColumn注解，Map也是类似的。

### 5.6.5 集合属性的性能分析

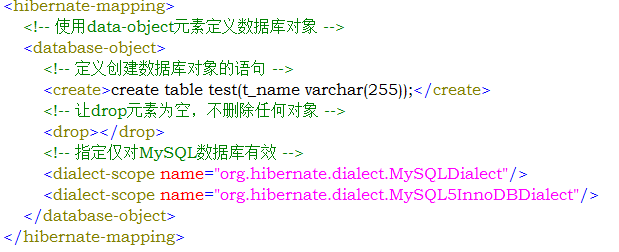
由于集合属性一般都会加载很多数据，所以一般采取延迟加载策略，有序集合在添加、删除、修改有好的表现。

### 5.6.6有序集合映射

Hibernate支持SortedSet与SortedMap2个有序集合实现，要使用@SortNatural与@SortComparator注解指定排序的规则，@SortNatural注解就是指定按照自然排序，@SortComparator要指定一个Comparator的实现类，如果要查询自己实现排序功能，可以在集合属性上加入@Orderby注解，在数据库查询时就是排序了的。

### 5.6.7 映射数据库对象

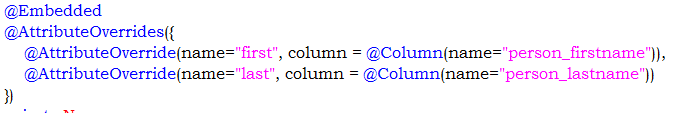
如果要创建数据库对象例如存储过程、触发器、表、序列等，可以使用<database-object.../>元素，在传统的\*.hbm.xml文件中声明，还能指定其应用范围，就是指定其方言，属于该方言的数据库才会创建。具体形式如下：



创建表也可以使用SchemaExport，可以直接执行SQL文件。

## 5.7 映射组件属性

持久化类的属性都是基本类型就使用@Column指定相关列，如果是自定义组合列的类型呢，这种就是组件，自定义组件类需要使用@Embeddable标示，如同@Entity一样，里面使用@Column指定列明，会定一个使用该组件的类，使用@Parent标识，它会指向使用该组件的持久化对象，也可以不这样写，在持久化对象内部的组件上使用@Embedded组件标识，为组件内部的属性指定列名使用@Attributeoverrides与@Attributeoverride2个注解。如下所示：



### 5.7.1 组件属性为集合

如果组件里面有属性是集合类型，那么使用方式与在持久化对象里面是一样的，其实组件与持久化对象的属性示统一的。

### 5.7.2 集合属性的元素为组件

大多数情况下，集合属性的元素都是自定义的组件，配制方法与单一的时候差不多，不能使用@Column来映射单一列了，使用@Embeddable修饰，其他相同，要在组件内部指定列名。

### 5.7.3 组件作为Map的索引

Map中使用自定义组件作为key，需要在注解中@MapKeyClass中指定组件类，同时需要覆写hashcode与equal函数。

### 5.7.4 组件作为复合主键

复合主键是经常情况，需要自定义一个主键组件，这个组件要有无参数的构造器，要实现Serializeable接口，要覆写hashcode与equal函数，使用@EmbedddedI D注解来在持久化类中标识，其他注解与普通的组件使用一样，但是这个组件注解无法指定生成策略，需要在程序中指定自定义的主键值。

### 5.7.5 多列作为联合主键

也可以不使用自定义组件做复合主键，之间在所有的主键列上用@Id标识，就定义了复合主键，但是需要有无参的构造函数，以及实现Serializeable接口，还要根据主键属性覆写hashcode与equal函数。

## 5.8 使用传统的映射文件

传同方式使用XML文件表达数据库数据之间的关系，来完成映射。

### 5.8.1 增加XML映射文件

需要在同一个包下有2个文件，POJO+XML文件，名字要符合name.hbm.xml的形式，配置形式如下：



需要在Hibernate的配置文件中加载该XML文件，  
<mapping resource="org/crazyit/app/domain/Person.hbm.xml"/>。

# 第六章 深入使用Hibernate

这一章主要讲述的内容是对象之间关系对底层数据库的映射关系的影响，Hibernate的查询体系，HQL，性能，事务等。

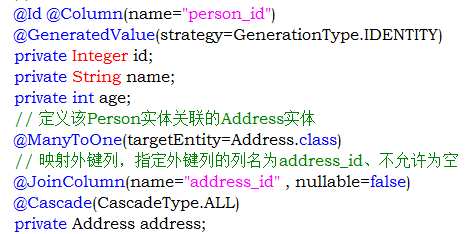
## 6.1 Hibernate的关联映射

对象关联的方式有单向，单向访问关联端，双向关联的关系2端可以互相访问。单向关联1->1，1->N，N->1，N->N；双向关联1<->1，1<->N，N<->N。

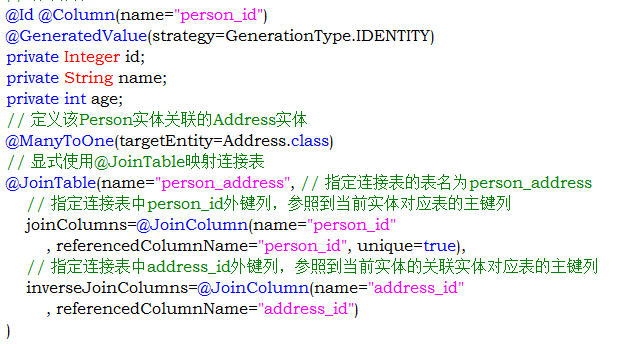
### 6.1.1 单向N->1关联

N->1是常见的关系，含义是多个对象中含有同一个对象的引用，为了表达这种映射关系，需要在N端增加一个属性引用端的关联实体，Hibernate用@ManyToOne修饰N端这个属性，这个注解支持的属性包括：cascade，指定级联策略；fetch，指定抓取策略；optional，关联关系是否可选；targetEntity，指定关联实体的类名，一般不给出，Hibernate通过反射推算出。在用于泛型的属性时要给出。

1. 无连接表的N->1关联中，只需要设置相关属性为外键，并设置关联，并设置级联策略；当对表进行操作时，会先更新关联的外键的表，否则会报出异常。



1. 连接表关联，本质跟上面是一样的，只不过这通过本表的主键与连接表的外键相连接，这样本表只能最多对应一条连接的数据，使用的是@JoinTable注解



### 6.1.2 单向1->1 关联

在持久化类里增加代表关联实体的成员变量，并用注解@OneToOne修饰，注解里可以增加很多的属性，可以通过外键关联并设置unique=true来形成单向的11关联；有连接表或者无连接表的用法与上面相同。

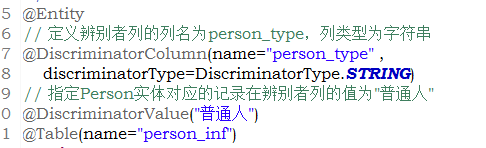
此处省略不做继续补充，需要可以看书。开发时参考资料即可。

## 6.2 继承映射

类对象存在继承与多态的关系，为了表达Java的这种关系，Hibernate也提供了映射方式。有3种映射策略：整个类层次对应一个表、连接子类的映射策略、每个具体类对应一个表。

### 6.2.1 整个类层次对应一个表的映射策略

默认的映射策略，父类、所有子类的信息都存储在一个数据表中，为了区分每行记录属于哪个子类或者父类实体，需要增加辨别者列，使用@Discrimination Column标识；@DiscriminationColumn修饰根父类，@DiscriminatorValue修饰每个子类，指定不同子类在辨别者列上的值。



这种方式将所有数据集合在一个表中，性能是最好的。

### 6.2.2 **连接子类的映射策略**

这种策略是父类的属性存储在父类表中，子类增加的属性存储在子类自己的表中，由于不是默认的注解，需要指定@Inheritance注解。定义的格式如下：

@Inheritance(strategy=InheritanceType.JOINED)，在父类上加入继承策略，同时不需要指定辨别者列。

### 6.2.3 每一个具体类对应一个表的映射策略

这种映射策略与上一个的不同指出在于，子类的数据完全存在与子表中，需要用注解指定@Inheritance(strategy=InheritanceType.TABLE\_PER\_CLASS)，不能指定生成策略。

## 6.3 Hibernate的批量处理

Hibernate针对大批量数据处理的解决方案。

### 6.3.1 批量插入

Hibernate的session持有一个一级缓存，当大批量数据操作时，可能产生内存溢出问题，所以要定期刷新缓存，输出到数据库，可以采用计数的方式。

### 6.3.2 批量更新

更新也可以采用计数的方式，可以采用scroll()方法用游标的方式读取记录。

### 6.3.3 DML风格的批量更新/删除

使用HQL语句批量更新或者删除，在下一章介绍。

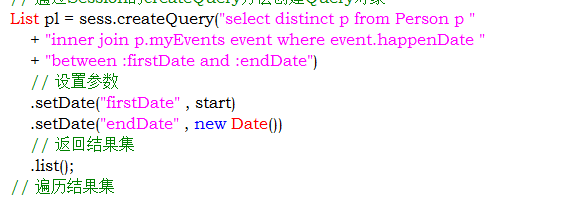
## 6.4 使用HQL查询

Hibernate有多种查询方式可以选择，功能丰富。

### 6.4.1 HQL查询

HQL，Hibernate Query Language；面向对象查询语言，语法类似于SQL，操作对象是类、实例、属性等。

查询步骤：1.获取Hibernate Session对象，2.编写HQL语句，3.使用HQL作为参数调用Session的createQuery()，4.为参数赋值，5.调用查询对象的list()，uniqueResult()返回查询列表（持久化实体集）。基本代码的形式如下：



语句中可以使用占位符，占位符的形式可以是？N序号，这种也可以是：name这种，setXXX可以级联添加参数，createQuery()创建的Query对象有setFirstResult(int)与setMaxResults(int)方法用于分页查询。

### 6.4.2 HQL查询的from子句

From Person as p表明选择所有的Person持久化类对象，推荐使用别名用法as，起的别名为p，后面可以添加多个持久化类，添加时将产生一个笛卡尔积连接。

### 6.4.3 关联和连接

从多个表中取得数据，使用多表连接查询，HQL支持2种关联方式，隐式连接与显式连接。隐式连接使用点号连接关联实体，显式使用join，就是说在隐式连接中，from后只有一个表，读取的数据也是该表内容，关联实体可能出现在where判断中，显式连接就是真正的组合表记录了。Join连接支持标准的SQL关联语法。由于隐式连接不能出现在from中，所以其不能是集合，否则会报异常，对于持久化对象的集合属性，默认采用延迟加载策略。如果托管了，可能是没有值的，可以采用立即加载，可以在类结合属性上加注解fetch策略，也可以在HQL查询时使用join fetch。Join连接中，可以使用with 添加连接条件相当于SQL的on。

### 6.4.4 HQL的select子句

Select p.name.firstName from Person as p；select可以选择出任何对象，任何列，只要是跟后面的Person有关的，select后的元素默认会生成一个Object数组作为List的一个集合元素，默认元素形式可以改变

Select new list(p.name,p.address),为使用list为元素，List处也可以是自己定义的类，元素就是自定义类对象。里面的表达式可以使用as 起别名，在元素形式是Map时，别名会成为Key。

### 6.4.5 HQL查询的聚集函数

聚集函数的使用与SQL相同。

### 6.4.6 多态查询

HQL查询也会返回持久化类的子类对象。

### 6.4.7 HQL查询的where子句

Where子句中，如果没有为持久化类起别名，直接使用属性。里面可以是各种的复合表达式。

### 6.4.8 表达式

HQL支持大部分的标准SQL表达式用法与运算符。

### 6.4.9 order by子句

### 6.4.10 group by子句

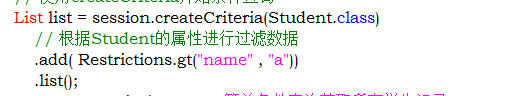
### 6.4.11 子查询

### 6.4.12 命名查询

命名查询就是将查询语句放到注解中，使用@NamedQuery或者@NamedQueries注解。

## 6.5 查询条件

查询条件操作的相关类：Criteria代表一次查询，Criterion代表一个查询条件，Restrictions产生查询条件的工具类。步骤如下：1.获得Hibernate的Session对象；2.以Session对象创建Criteria对象；3.使用Restrictions的静态方法创建Criterion查询条件；4.向Criteria查询中添加Criterion查询条件；5.执行Criteria的list()或uniqueResult()方法返回结果集。程序示例如下：



关于详细细节看书。

### 6.5.1 关联和动态关联

如果需要关联其他实体，可以对Criteria对象再次调用很多重载的createCriteria()函数，参数表明了一些关联的相关信息，如果是连接自身，使用重载的createAlias()函数连接。

### 6.5.2 投影、聚合和分组

投影是一种基于列的运算，计算指定列的值，并可以完成分组功能。Projection相关的类代表投影功能。

### 6.5.3 离线查询和子查询

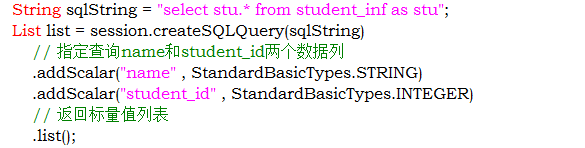
DetachedCriteria代表离线查询，允许在一个session之外创建一个查询，当把DetachedCriteria对象作为一个查询条件添加到Criteria中时，DetachedCriteria就变成了一个子查询。

## 6.6 SQL查询

原生SQL查询：可以利用数据库特有的特性，将JDBC应用迁移到Hibernate时。主要使用SQLQuery接口，这个接口是Query接口的子接口。方法主要有：list()获取结果集，setFirstResult()与setMaxResult()；addEntity()将查询到的特定记录与特定实体关联，addScalar()将查询到的结果与特定类型关联。执行的步骤如下：

1.获得session对象；2.编写SQL语句；3.以SQL语句作为参数，调用createSQL Query()创建查询对象；4.调用addScalar()或者addEntity()与变量关联；5.调用setXXX()添加参数；6.调用list()获得结果集。

### 6.6.1 标量查询



AddScalar()指定查询结果包含哪些数据列，没有被addScalar()选出的列将不会被包含在查询结果集中，指定查询结果中数据列的数据类型。指定addScalar()后使用的是ResultSet数据集，没有的话默认用ResultSetMetadata来判断返回数据列的实际顺序和类型。

### 6.6.2 实体查询

标量查询的一条记录对应的书一个Object数组，要把一条记录对应成Entity就要使用addEntity()方法，SQL查询必须是返回全部的列的查询才可以。有关联多个表时，可以得到多个表的实体，组成一个数组，对应一条记录；setTransforme r()可以将结果集转换为普通的javaBean对象的list。

### 6.6.3 处理关联和继承

关联实体的查询结果由addJoin()方法返回。

### 6.6.4 命名SQL查询

使用@NamedNativedQuery注解命名原生SQL查询，可以指定一些属性。看书。

### 6.6.5 调用存储存储过程

仍使用@NamedNativedQuery注解，query部分写调用存储过程或者函数的语句，调用的语法遵循SQL92，｛call prodedurename(parameters)｝等。

### 6.6.6 使用定制SQL

Hibernate在后台操作持久化对象时遵循一套预定义的SQL操作，这个SQL操作可以更改为自己定义的，在持久化类上加上@SQLInsert...等语句，里面的value为对应的SQL语句，在对这个持久化类进行对应的操作时，执行自己定义的SQL操作。

## 6.7 数据过滤

定义过滤器@FilterDef，使用过滤器@Filter，session启用过滤器。

## 6.8 事务控制

每个业务逻辑方法都是由一系列数据库操作完成的，这一系列操作是一个整体，要么成功，要么失败全失败，这就是事务。一个最小的逻辑执行单元。

### 6.8.1 事务的概念

具备4个特性：原子性Atomicity、一致性Consistency、隔离性Isolation、持续性Durability。

原子性：最小执行单位；

一致性：执行前后的数据库状态都是一致的，符合规则的；

隔离性：不影响其他事务的执行；

持续性：事务产生的结果被有效记录。

### 6.8.2 Session与事务

Hibernate的事务是通过sesssion开启的，是Transaction对象，以调用commit终止，Hibernate只是对底层事务进行了抽象，Hibernate的事务产生是有TransactionFactory产生的，根据事务环境的不同，Hibernate自动为你返回特殊的事务实现，session打开并不是打开数据库连接，只有操作数据时才进行连接，所以长session对性能影响不大，事务要尽可能短。通常用户在客户端的一次HTTP请求对应一个session，跨HTTP请求的session是不推荐的，影响用户的并发数量。

### 6.8.3 上下文相关的Session

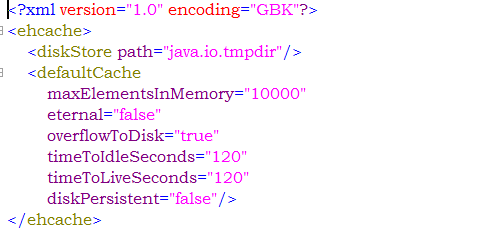
主要是HibernateUtil类，这个类可以产生上下文相关的Session，根据Session状态完成事务控制。

## 6.9 二级缓存和查询缓存

Hibernate分为2级缓存，一级缓存为Session级缓存，二级缓存为SessionFactory级缓存，一级缓存一致开启，对于持久化对象的操作窦腺存到Session缓存，flush或者关闭时在执行数据库操作，提高数据库访问性能，二级缓存默认是不开启的，需要手动开启。

### 6.9.1 开启二级缓存

开启二级缓存需要在hibernate.cfg.xml文件内增加<property name="hibern ate.cache.use\_second\_level\_cache">true</property>标签，需要指定缓存的实现类，有EhCache与Infinispan等可选，如下：<property name="hibernate.cache.region. factory\_class">org.hibernate.cache.ehcache.EhCacheRegionFactory</property>；缓存加入进来后需要配置缓存的一些参数，通常是classpath路径下的XML文件，如下文件ehcache.xml如下：



详细每个参数的含义需要参考资料，设置对哪些实体类、持久化对象启用二级缓存@Cache表示启用或者XML配置方式的<class-cache />等。

@Cache注解支持usage属性，表示缓存的策略有CacheConcurrencyStra tegy.READ\_ONLY（不修改，只读）、CacheConcurrencyStrategy.READ\_WRITE等。

### 6.9.2 管理缓存与统计缓存

Session是局部缓存，将对象从缓存中除去调用.evict(object)方法，判断是否在缓存中使用contains(Object)方法，清空缓存使用clear()方法，SessionFactory的二级缓存是全局缓存，提供了getCache()方法返回Cache对象，可以管理与统计缓存的信息。

### 6.9.3 使用查询缓存

慎用。

## 6.10 事件机制

Hibernate执行持久化过程中，会发生特定的一些事件，可以监听这些事件的发生，决定此时进行怎么样的处理。这样应用程序就能参入到持久化的操作中来。由2个部分：拦截器动作回调定义与事件监听

### 6.10.1 拦截器

实现Interceptor接口或者继承EmptyInterceptor，通过session或者Configu ration启用全局拦截器。

### 6.10.2 事件系统

Session接口的每个操作方法都有对应的事件对象，这些对象是单例的，所有session共享，比如LoadEvent、FlushEvent等。Session操作时，会生成事件并调用对应的事件监听器方法。使用事件系统的步骤：1.实现自己的监听器；2.注册监听器，代替默认的监听器。实现自己的监听器可以实现接口，但是Hibernate的持久化操作都是在监听器方法内实现的，如果自己实现接口，就要自己写持久化操作，一般选择继承，覆写方法加入自己的一些逻辑，在调用父类方法完成持久化，注册监听器使用EventListenerReqistry接口，可调用的方法：appendListener()在监听器队列后追加，prependListener()添加到队列的前面，3.setListener()替换队列里面的监听器。

# 第七章 Spring的基本用法

## 7.1 Spring简介和Spring4.0的变化

Spring是从实际开发中抽取出来的框架；提供了轻量级的解决方案，包括：基于依赖注入的核心机制，基于AOP的声明式事务管理，与多种持久层技术的整合，优秀的web mvc框架，是一站式解决方案。



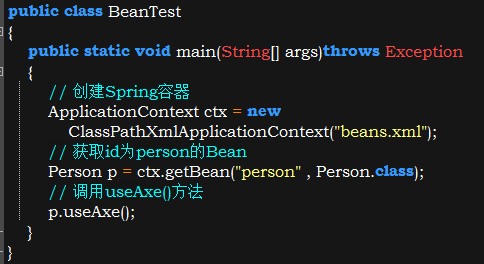
Spring4.0全面支持java8，Spring还是升级了很多有用的东西。

## 7.2 Spring入门

添加Spring的jar包；Spring核心容器的理论：超级大工厂，所有的对象，包括数据源、Hibernate SessionFactory等基础性资源；都会被当成Spring核心容器管理的对象，这些对象统称为Bean；一切java对象都是bean，Spring通过XML文件知道应该管理哪些Bean。



根元素是<beans>每个<bean>元素定义一个bean，Spring读取文件，通过反射的方式生成class对象。大致的过程是；idStr为id属性值，classStrs为class属性值，通过反射调用该类的无参的构造函数生成对象；然后container.put(idStr,object)放入到Spring容器中；不能是接口，不能是抽象类。上面还有<property>元素，它驱动Spring通过反射执行setter方法，方法名字的确定根据name属性，如果是基本类型的值可以通过value指定值，如果是其他bean，通过ref指定，作为传入参数；bean创建后就会马上执行setter方法；ApplicationContext是Spring容器最常用的接口，这个接口有2个实现类：ClassPathXmlApplicationContext,根据类加载路径来搜索配置文件，并创建Spring容器；FileSystemXmlApplicationContext，从文件系统的绝对路径或相对路径搜索配置文件并创建Spring容器。



Spring容器获取bean有2种方法，Object getBean(id)，T getBean(id,Class<T> type)，一个需要进行类型转换，一个不需要类型转换。

## 7.3 Spring的核心机制：依赖注入

依赖在对象中来说就是A对象里面有B对象，Spring框架的功能，Spring容器作为超级大工厂，负责管理创建java对象，并管理对象之间的依赖关系。

### 7.3.1 理解依赖注入

依赖现象，Spring通过<property>元素的setter方法完成依赖注入，不需要硬编码，完成了解耦合；Spring通过Bean创建管理对象，通过<property>完成bean之间关系的管理。依赖注入就是setter方法的使用，也叫做控制反转；IoC；依赖注入有2种方式：设值注入，IoC容器使用成员变量的setter方法来注入被依赖对象；构造注入，IoC容器使用构造器来注入被依赖对象。

Spring大容器管理bean，业管理bean之间的依赖关系。

面向接口编程，使程序设计与实现分离，有助于解耦合；构造注入是在进行构造实例时，完成依赖关系的初始化，实际就是反射时调用带参数的构造函数，此时类中要有带参数的构造函数，在bean元素里配置<constructor-arg>元素，代表一个输入的构造函数中的一个参数值，可以通过index属性指定序号，type属性指定类型等；多少个参数就设置多个个<constructor-arg>元素。

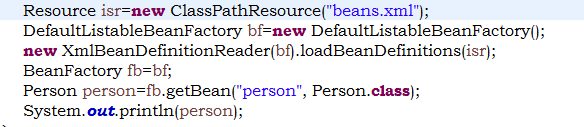
设值注入更容易理解，构造注入会使代码臃肿，构造注入可以决定构造顺序。

## 7.4 使用Spring容器

Spring有2个核心接口，BeanFactory与ApplicationContext，Spring是bean工厂，所有的组件都被当成Bean处理，一些java对象都是Bean，Spring需要知道Bean的实现类以便生成对象。

### 7.4.1 Spring容器

Spring容器的接口是BeanFactory，负责创建、配置管理Bean，子接口是ApplicationContext，BeanFactory含有几个获取Bean实例、类型，或者判断是否存在的泛型方法，基本的实现类是DefaultListableBeanFactory，ApplicationContext接口的实现类是FileSystemXmlApplicationContext、ClassPathXmlApplicationContext、AnnotationConfigApplicationContext。在web中有XmlWebApplicationContext与AnnotationConfigWebApplicationContext2个实现类，Bean详细配置信息通过XML文件通过Resource的方式传入，Web下ApplicationContext自动加载为实例（在Web.xml文件中配置），应用程序则手动创建，单个或者多个XML文件都可以导入进去。这2个容器可以代表生成Bean的工厂。创建BeanFactory（知只能传单个配置文件）。



ApplicationContext，应用更广，创建方便，可以加入多个xml文件。

### 7.4.2 使用ApplicationContext

ApplicationContext（Spring上下文）比Beanfactory更好用，web中无需手动创建，ApplicationContext还提供的功能如下：默认预初始化所有的singleton bean，提供国际化支持、资源访问、事件机制、同时加载多个配置文件、以配置文件的方式生成（我觉得BeanFactory的生成方法太别扭了）等；如果不想初始化singleton bean，加上lazy-inti=true。所以创建容器时系统开销较大，不加修饰，默认的Bean都是singleton的，

### 7.4.3 ApplicationContext的国际化支持

ApplicationContext接口继承了MessageSource接口，MessageSource接口定义了2个用于国际化的方法：

String getMessage(String code,Objects[] args Locale loc);

String getMessage(String code,Objects[] args,String default,Locale loc));

创建Spring容器时，使用配置文件中名字为messageSource的Bean，上述2个方法的调用为该Bean处理，没有的话，找父容器有没有这个bean,还没有的话创建一个空的staticmessagesourcebean；配置messagesSource Bean常用ResourceBundleMessageSource类。



这个类会到类加载路径下加载message开头的资源文件，在前端代码调用时，根据环境选择响应的资源文件内的内容。code为资源文件中的key，Objects为传入的参数｛0｝｛1｝等值。



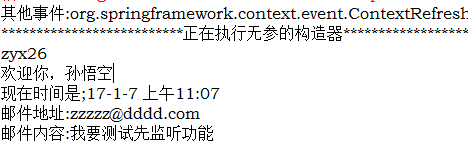
### 7.4.4 ApplicationContext的事件机制

ApplicationContext是观察者模式的实现，通过ApplicationEvent与Applicatio- nListener接口，实现ApplicationContext的事件处理，容器中有一个Applicatio- nListener的Bean，那么容器中发布ApplicationEvent的时候，这个Bean就会被自动触发执行。ApplicationEvent容器事件，ApplicationListener监听器，事件由事件源（Spring容器）、事件ApplicationEvent与事件监听器ApplicationListener组成，且事件必须有java程序显示触发。定义一个事件，继承ApplicationEvent，任意对象均可以；定义一个ApplicationListener，实现这个接口，必须实现方法onApplicationEvent(ApplicationEvent event);配置在容器中，容器创建，加载时会自动发出一些内置事件，也可以通过applcaitonContext手动发出事件，publishEven t()。



监听器不需要ID。





结果看到监听了内置事件，可以在监听器里实现回调自定义方法。

Spring内置的事件：

1. ContextRefreshedEvent：ApplicationContext初始化或者刷新触发，所有的Bean被加载、SingleBean被预实例化，后处理的bean被检测激活，容器就绪可用；
2. ContextStartedEvent：ConfigurableApplicationContext，子接口，调用start启动容器时触发；
3. ContextClosedEvent：ConfigurableApplicationContext，子接口，调用close关闭容器时触发。
4. ContextStopedEvent：ConfigurableApplicationContext，子接口，调用stop停止容器时触发。
5. ContextHandledEvent：只能用与Web，在Spring做MVC中，处理完用户请求，自动触发。

### 7.4.5 让Bean获取Spring容器

Bean都是在容器中的，在Web中，容器是通过声明式产生的，Bean都没有持有Spring容器，假如Bean需要输出国际化消息或者发布容器事件，就需要持有Spring容器了，此时必须实现容器接口BeanFactoryAware或者Applicatio nContextAware接口，里面有一个set方法，可以在初始化bean后由容器自动执行，自动传参赋值。

## 7.5 Spring容器中的Bean

Spring是超级大工厂，产品是Bean，开发Bean，配置Bean，Spring创建Bean并调用里面的属性方法完成依赖注入，Spring通过XML驱动java代码，由JAVA代码管理的耦合关系提取到XML配置文件中，实现了解耦，利于维护升级。

### 7.5.1 Bean的基本定义和Bean别名

<beans>是配置文件的根元素：可配置一些属性。

* Default-lazy-init：所有Bean的默认的延迟初始化行为；
* Default-merge：所有bean默认的merge行为；
* Default-autowire：所有Bean默认的自动装配行为；
* Default-autowire-candidates：候选的自动装配行为；
* Default-init-method：所有Bean的默认初始化方法；
* Default-destroy-method：默认的回收方法。

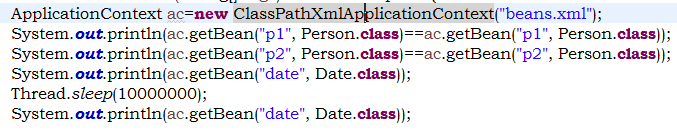
上述属性去掉default后也是每个Bean可以配置的属性，bean配置的属性可以覆盖beans的。

<bean>代表一个java实例，id唯一标识、class属性，实现类；id是XML规范的ID，只能是字母数字，name属性作为别名可以用特殊字符，且别名之间用空格冒号逗号分开，也可以通过<alias>起别名，name属性指定id或者alais的别名，alais属性指定别名。

### 7.5.2 容器中的bean的作用域

* Singleton：单例模式，bean只生成一个实例；
* Prototype：每次通过容器的getBean()方法获取Bean时，产生一个新的Bean；
* Request：web中有用，一次请求，只生成一个实例，会话中得到的都是一个实例；
* Session：web中有用，同一次HTTP回话内，只生成一个实例，会话中得到的都是一个实例；
* Global session：web有效，全局HTTP session对应一个Bean实例。

Singleton Bean，容器负责创建跟踪回收，prorotype的容器只负责new，不会跟踪；默认singleton，prototype频繁创建回收开销较大，属性scope负责指定作用域。



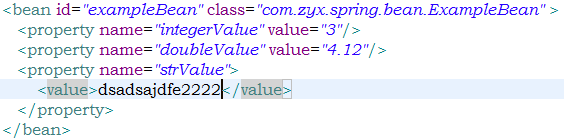
Spring3.0还支持自定义作用域，看官方文档。

### 7.5.3 配置依赖

Bean有属性就是依赖，属性的注入，依赖注入有2中方式：设值<property>执行setter方法与构造<constructor-arg>执行带参数的构造函数；基本类型值不建议使用配置文件，配置文件主要管理Bean与Bean之间的关系，applciationContext预初始化singleton bean与它的依赖Bean，beanfactory不会初始化任何Bean，启动时不抱错，可能实例化Bean时报错。可以在在<property>与<constructor-arg>中配置的java类型如下：value(基本值)、ref(Bean)、bean、list、set、map、props。

### 7.5.4 设置普通属性值

普通值使用元素<value>或者属性value配置，基本类型（包括包装类）与String类型都可以，XML解析器来解析，PropertyEditor完成类型转换（从String转换成相应类型）；在<property>中使用元素<value>来配置是的代码多，所以一般是使用属性value来配置。

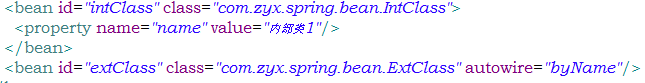


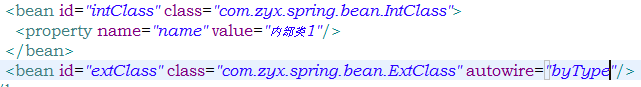
### 7.5.5 配置合作者Bean

配置依赖Bean，也可以使用元素<ref bean=””>或者属性ref。<constructor-arg>同理。

### 7.5.6 使用自动装配注入依赖Bean

Spring能够自动装配Bean与Bean之间的依赖关系，无序用ref指定，<beans>元素的<default-autowire>指定这个集合bean所有的默认的装配规则，<Bean>元素的autowire也可以指定，指定的属性值如下：no：不使用自动装配，bean依赖必须通过ref元素定义，这是默认的配置；byName：setter方法名(首字母小写)与Bean的id匹配，没有不注入；byType：根据setter方法的形参类型与Bean类型匹配，找到多个抛异常，没有不注入；constructor：与byType类似，Bean匹配构造函数的参数类型，匹配注入；autodetect：Spring根据Bean结构决定使用constructor或者byType，如果有无参的构造函数，使用byType；



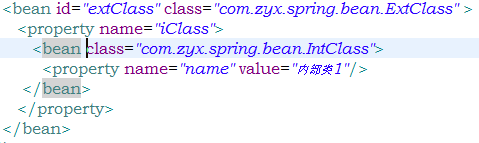


使用ref显示指定依赖时，自动装配不起作用；指定Bean的autowire-candidate

=”false” 这个bean不参与自动装配，<beans>的default-autowire-candidate可以指定很多bean或者很多正则表达式将符合这个id的bean排除在外。

### 7.5.7 注入嵌套Bean

某个Bean不想被Spring容器持有引用，理解为，这个bean只用于配置setter或者<constructor-arg>的参数；可以不要id。这个就是嵌套Bean。

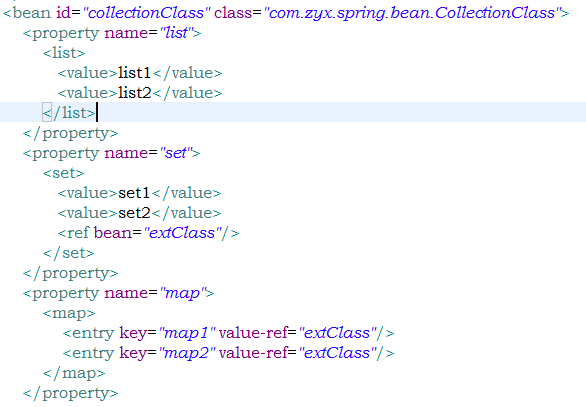


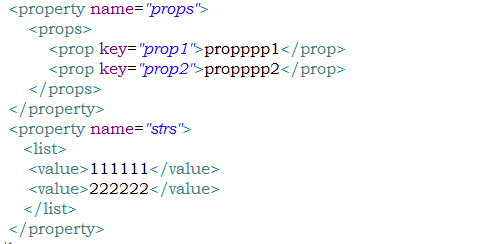
### 7.5.8注入集合值

Setter方法或者<constructor-arg>参数是集合，也可以注入，<list>、<set>、<map>、<props>分别设置类型List、Set、Map、Properties的集合参数值。

要点：<list>可用于为List与数组类型注入，<list><set><map>可接受的元素类型有：value-基本类型，ref-容器中的另一个Bean实例，bean-嵌套Bean，list、set、map、props集合元素；props只用与key=value都是字符串，所以容易配的；<map>有<entry>组成，entry配置key-value，key可以是value或者key-ref，map value可以是value或者value-ref，如果是臃肿的写法，那么还可以使ref、bean、list、set、map、props等等；props可直接在<value>元素里写key=value，会解析。

集合支持合并，加入父Bean中注入了一个集合元素，子bean中也注入来了集合元素，都是同一个属性，那么指定属性merge=“true”会合并集合，同名的覆盖父类的。集合是泛型的，注入的时候XML解析器会根据定义的类型解析bean中定义的集合类型值。





### 7.5.9 组合属性

类A含有类对象B，类B含有类对象C；C中含有属性name，在配置类A的bean时，property元素的name=”B.C.name”将为最后一个对象属性赋值，有一个前提就是此时B、C此时不为null，此时相当于通过反射调用了方法A.getB().getC().setName()；所以嵌套的对象不能为空。

### 7.5.10 Spring的Bean和Javabean

Spring的bean只要是java类就可以，有几点要注意：尽量为每个Bean提供无参构造函数，提供setter方法。

## 7.6 Spring3.0提供的JAVA配置管理

就是注解，通过注解的方式配置Bean，定义一个类，用@Configuration修饰表明这是个配置Bean的配置类，相当于<beans>的级别；

@Bean，方法修饰，将返回值定义成一个Bean；

@Value，定义一个变量；

@Import 导入其他配置类；

@Scope：修饰方法，指定Bean的声明周期；

@Lazy：修饰方法，指定Bean是否要延迟初始化；

@DependsOn：在初始化这个Bean之前需要初始化的Bean；

使用注解要用AnnotationConfigApplicationContext的实现类创建容器；这个类有个方法register(Class)用于添加配置文件的。

XML与注解混合使用：XML为主：在XML文件中加入<context:annotation-config>与<bean class=”配置类”>引入注解的配置bean;注解为主：在配置类上写@ImportResource(“XML文件路径”)引入XML配置。

## 7.7 创建Bean的3种方式

Spring支持调用构造器创建Bean、静态工厂方法创建Bean、实例工厂方法创建Bean。

### 7.7.1 使用构造器创建Bean实例

无参的，调用默认构造器，属性直接初始化为0、false或null，再根据依赖关系初始化相应的bean，再注入；带参数的直接用带参构造器创建对象，创建完成后返回对象。必须指定class属性。

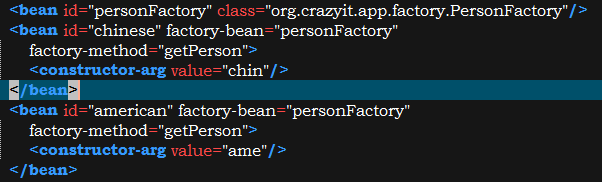
### 7.7.2 静态工厂方法创建Bean

Class属性指定生成Bean实例的静态工厂类，factory-method指定创建Bean实例的静态工厂类方法，工厂方法需要的参数用<constructor-arg>传入；其余与构造器差不多。



### 7.7.3 调用实例工厂方法创建Bean

与静态工厂类似，无需class属性，需要factory-bean属性，指定容器中工厂Bean的id，一样有factory-method指定方法，同样的传参方式。



## 7.8 深入理解容器中的Bean

管理Bean

### 7.8.1 抽象Bean与子Bean

抽象bean不能实例化，（不能注入，得到），用于被继承，存储共享的一些信息；abstract=”true”指定为抽象Bean，像抽象类；可以不指定class，指定了class，继承的类可以不指定class，class默认为父类的class实现类；子bean使用parent属性指定父Bean id，可继承实现类、构造器参数，property，子bean可以新增或者覆盖父bean的；无法继承depends-on\autowire\singleton\scope\lazy-init；

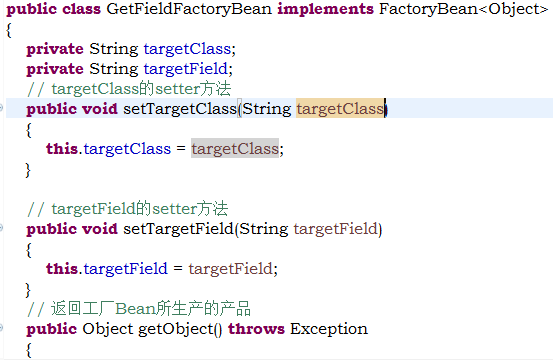


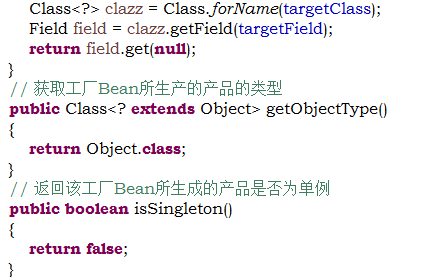
### 7.8.2 Bean继承与Java继承的区别

Bean继承（参数值延续，对象之间的关系，不具备多态性，实际就是没什么联系硬联系起来）与java继承完全不是一回事。

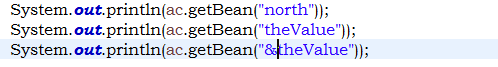
### 7.8.3 容器中的工厂Bean

工厂Bean是指实现了了FactoryBean接口的Bean，配置方法与普通Bean相同；用途也是产生Bean的，Spring的getBean()取得这个bean时得到的是它生产的Bean（getObject()）的值，而不是工厂Bean实例；FactoryBean接口：T getObject()，返回该工厂Bean生成的实例，Class<?> getObjectType()返回生成Bean的类型，boolean isSingleton()，返回的生成Bean是否是Java实例。









前面2个取得bean是工厂Bean生产的产品，前面加&会取得该工厂Bean本身，工厂Bean是很重要的工具类。

### 7.8.4 获得Bean本身的ID

实现接口BeanNameAware，里面有个方法setBeanName(String arg0)，这个方法在容器初始化该实例时，检查到接口BeanNameAware，容器调用这个方法，传入id参数。本对象中就可以获得这个值了。

### 7.8.5 强制初始化Bean

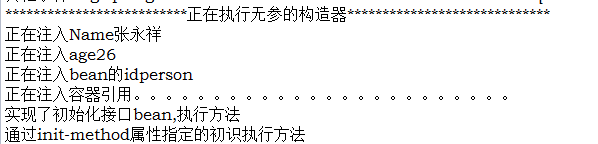
Spring在返回Bean之前，先初始化依赖bean，再将具备完整依赖的bean返回，如果依赖不够直接，被依赖的bean还没初始化完成，会抛出异常；depends-on属性指定这个依赖的bean要强制在本Bean之前初始化。

## 7.9 容器中Bean的生命周期

Spring可以管理singleton bean的生命周期，精确知道Bean何时被创建，何时初始化完成，何时销毁；Spring可以管理实例化结束后与销毁之前的行为，比如申请资源与销毁资源等；prototype的bean，Spring仅仅负责创建，创建完不在进行后续跟踪。

### 7.9.1 依赖关系注入后的行为

bean全部属性注入后，2种方式指定执行特定行为：使用init\_method属性，实现InitializingBean接口（void afterPropertiesSet()方法），可以同时指定，不矛盾，执行顺序如下图：



### 7.9.2 Bean销毁之前的行为

与上面类似，bean实例销毁前也有2种方式指定执行特定行为；使用destroy-method属性，实现DisposableBean接口（void destroy()）；执行的销毁顺序：



ApplicationContext接口不能关闭Spring容器，AbstractApplicaitonContext里面有个方法可以关闭。实现了生命周期接口，Spring会自动检测并执行相应的方法，可以使用<beans>元素的default-inti-method属性与default-deatroy-method属性指定一组bean的响应的周期方法。只要其中的Bean有这个方法就执行。

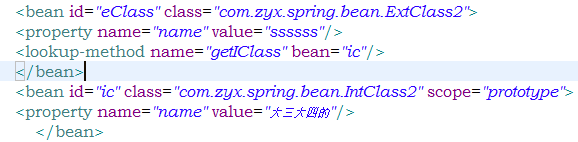


### 7.9.3 协调作用域不同步的Bean

Singleton Bean互相依赖，protoTypeBean依赖singletonBean注入没问题，singleton bean依赖prototype bean会出现问题，singleton bean只初始化一次，注入的prototype bean将跟随singletonbean的声明周期，因而失去它的prototype属性，表现为singleton因为持有引用，所以不会销毁；解决方法1，在singletonbean中创建prototypebean对象，造成了代码的耦合性。2.利用方法注入，lookup方法，在bean中设置一个抽象方法，返回值是prototype的Bean类型，在XML定义bean为抽象类，lookup-method属性指定那个抽象方法，name指定spring实现的方法，bean指定方法的返回bean id，Spring实现这个抽象方法逻辑就是在容器找到响应的Bean id返回。

代码如下：





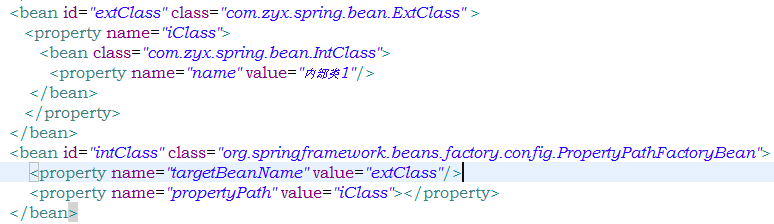
实际上Spring实现的抽象方法就是ctx.getBean(“ic”)；所以每次调用方法时注入的都是最新的Bean，为抽象方法定义方法后就是具体类可以实例化。

## 7.10 高级依赖关系配置

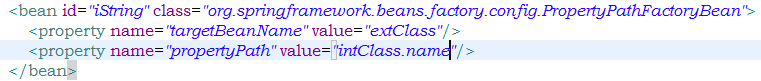
Spring还有很多非常规的注入方式，Spring的本质是XML元素驱动Spring执行对应代码，<bean>调用构造函数或工厂方法创建对象；<property>执行一次setter方法，<PropertyPathFactorybean>调用getter方法，FieldRetrievingFactoryBean访问类或者对象的Field值，MethodInvokingFactoryBean调用普通方法。

### 7.10.1 获取其他Bean的属性值

PropertyPathFactoryBean用来获取目标Bean的属性值，他是一个实现了FactoryBean的工厂Bean，调用getter方法返回属性，PropertyPathFactoryBean通过方法setTargetObject指定返回哪个对象Bean的属性，setPropertyPath，返回哪个gettter方法，支持多级嵌套，注意是Path。

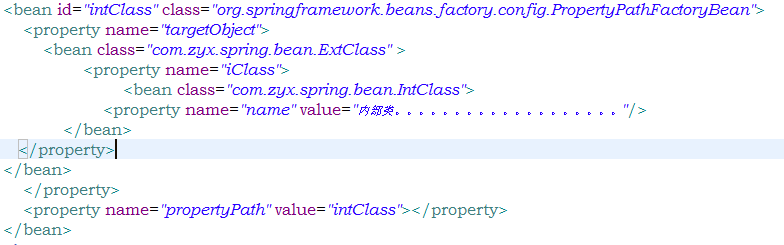


通过反射的方式获取Bean中的属性，value是getter方法名，不是属性名；getter返回的对象定义成了id=intClass的Bean，path里可以是嵌套的，因为是Path



如果没有给定targetBeanName与propertyPath，那么会根据id里面提供的内容自动解析targetBeanName与propertyPath，第一个点号之前的为Bean id，后面的为propertyPath，这是一种简写的方法如下：



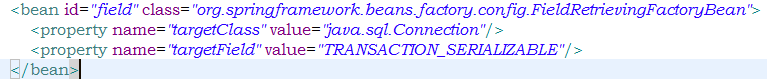


<util:property-path>（导入util命名空间）可以作为PropertyPathFactoryBean的简化配置，需要指定id与path。

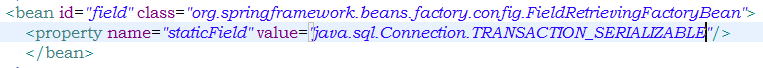
### 7.10.2 获取Field值

通过FieldRetrievingFactoryBean获得类的静态或这实例Field，可以注入其他bean，或者独立成Bean；

Field是静态：setTargetClass指定调用哪个类，setTargetField指定访问的Field；Field是实例：setTargetObject指定调用哪个对象，setTargetField指定访问的Field（要求Field是public，所以用处不大）；



还有setStaticField方法，直接指定哪个类哪个Field，简化写法；



跟上面一样，如果没有配置property，根据id来解析Field路径。



<util:constant>元素可以作为它的简化配置，指定id与static-field。

### 7.10.3 获取方法返回值

调用类或者对象的任意方法，返回的对象可以定义为Bean，或者注入其他Bean中；

类方法：setTargetClass调用哪个类；setTargetMethod调用哪个方法，setArguments(Object[] arguments)指定参数，无参可省略；

实例方法：setTargetObject调用哪个对象；setTargetMethod调用哪个方法，setArguments(Object[] arguments)指定参数，无参可省略。

XML文件中应该配置的信息：

项目升级、维护经常需要改动的信息，控制项目内各组件耦合关系的代码。

## 7.11 基于XML Schema的简化配置方式

简洁。

### 7.11.1 使用p:命名空间简化配置

p命名空间可以替换<property>，简化设值注入，例子：





如果要注入Bean，使用比如p:name-ref。

### 7.11.2 使用c:命名空间简化配置

简化构造注入；与p基本相同，此外还支持参数顺序法c:\_N代表构造函数的第几个参数。

### 7.11.3 使用util:命名空间简化配置

导入最新的spring-util-4.0.xsd，增加响应的空间；

元素：constant：获取静态域的值，FieldRetrievingFactoryBean的简化配置，property-path，获取指定对象的getter方法的返回值是propertyPathfactorybean的简化配置，list用于配置listbean，id，list-class，scope；set 配置set Bean，id，set-class，scope；map同样，properties：id，location，scope。

## 7.12 Spring 3.0提供的表达式语言SpEL

与JSP2的EL类似，功能强大，可以单独使用也可以在注解及XML配置中使用SpEL，简化Spring的Bean配置。

### 7.12.1 使用Expression 接口进行表达式求值

SpEL对表达式计算，求值有3个接口：ExpresstionParser，负责解析SpEL表达式，返回一个Expresstion对象，Expresstion，代表一个表达式；EvaluationContext：计算表达式值的上下文，比如含有变量时，需要它来计算。Expresstion含有的方法：

Object getValue()计算表达式值；

<T> T getValue(Class<T> desiredResultType)，计算结果，将结果当desiredResultType类型处理；

Object getValue(EvaluationContext context)，指定EvaluationContext来计算值；

<T> T getValue(EvaluationContext context,Class<T> desiredResultType)指定EvaluationContext计算，将结果当成desiredResultType类型；

Object getValue(Object rootObject)rootObject作为root对象来计算表达式的值；

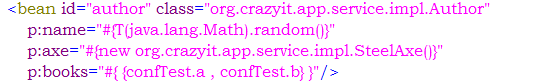
<T> T getValue(Object rootObject,Class<T> desiredResultType) rootObject作为root对象来计算表达式的值，结果当成desiredResultType类型处理。



EvaluationContext 代表上下文，可以包含多个对象，但是只能有一个root对象。SetVariable(String name Object value)放入对象，名字是name；访问时用#name，setRootObject(Object rootObject)放入跟对象，访问root对象的属性是可不用指定该root对象。

### 7.12.2 Bean定义中的表达式语言支持

SpEL可用于XML与注解的Bean配置，形式如下：



### 7.12.3 SpEL语法详述

1. 直接量表达式：就是java中的常量，基本类型，比如字符串、日期、数值、boolean值和null；
2. 支持数组；
3. List集合；
4. 在表达式中访问List、Map集合list[index]，map[key]的访问方式；
5. 调用方法与java的方式一样；
6. 支持算术、比较、逻辑、赋值、三目运算符；
7. T()限定将表达式中的字符传当成所表示的类处理，而不是字符串；
8. 允许new；
9. 设置变量，this变量，当前正在计算的变量，root，根变量；
10. 自定义函数，将java方法重新起名；
11. Elvis运算符，简写的三目运算符；
12. 安全导航操作。

# 深入使用Spring

本章介绍Spring后处理器、AOP以及Spring与其他框架的整合。

## 8.1 两种后处理器

Spring允许开发者对IoC容器拓展，无须实现BeanFactory与ApplicationContext接口，允许从2个方面拓展IoC容器：

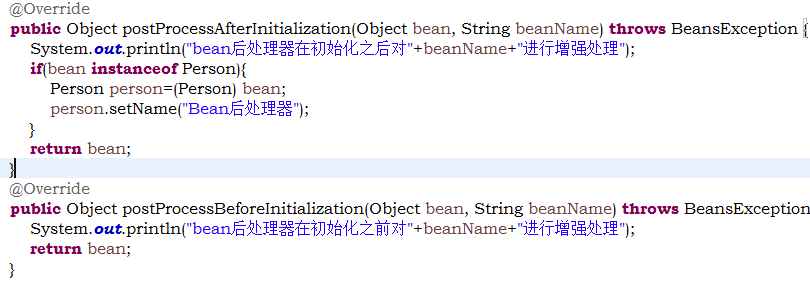
1. Bean后处理器，对Bean进行后处理，加强Bean功能；
2. 容器后处理器：对IoC容器进行后处理，增强容器功能；

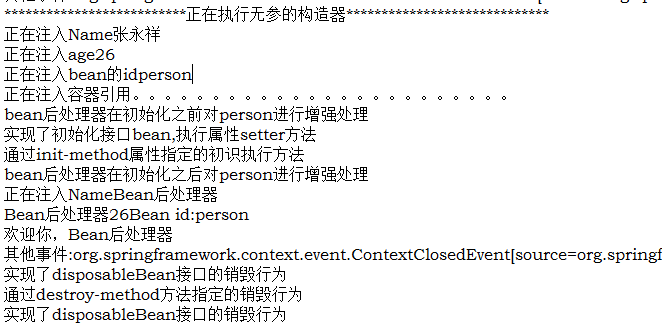
### 8.1.1 Bean后处理器

Bean后处理器是一种特殊的Bean，在Bean创建成功后，对Bean实例进行处理，实现接口BeanPostProcessor；包含2个方法：

1. Object postProcessBeforeInitialization(Object bean,String name);
2. Object postProcessAfterInitialization(Object bean,String name);

参数bean是即将进行后处理的bean对象，name是其id。这2个方法分别会在Bean初始化之前与初始化之后调用，配置Bean后处理器与普通Bean无区别，在ApplicationContext接口中，可以不指定id，在Beanfactory接口中要指定id属性，ApplicationContext容器会自动检测bean,如果实现了上述的接口，就自动注册为Bean后处理器，如果是BeanFactory就要手动注册，所以需要id。





### 8.1.2 Bean后处理器的用处

Bean后处理的用途是代理-Proxy；Spring框架提供了大量的Bean后处理器；BeanNameAutoProxyCreator：根据Bean实例的name属性，创建Bean实例的代理；

DefaultAdvisorAutoProxyCreator：根据提供的Advisor，对容器中的所有Bean实例创建代理。这里我也不知道有啥用处。

### 8.1.3 容器后处理器

Bean后处理器负责处理Bean，容器后处理器处理容器本身；实现接口：

PostProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)，方法体内部对容器进行自定义拓展；使用方式与Bean后处理器一致。在容器实例化之后与Bean实例化之前执行（此时已知道所有的bean id），Spring提供的常用的容器后处理如下：

PropertyPlaceholderConfigurer：属性占位符配置器；

PropertyOverrideConfigurer：重写占位符配置器；

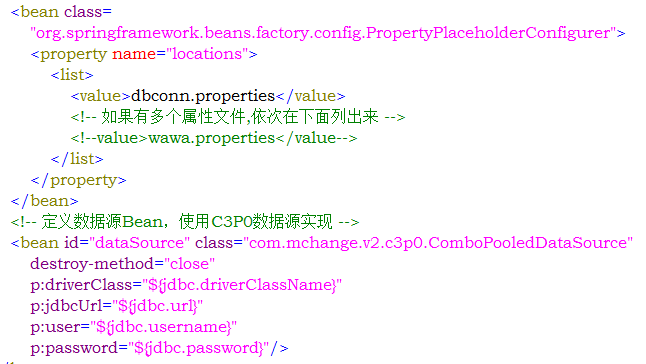
CustomAutowireConfigurer：自定义自动装配的配置器；

CustomScopeConfigurer：自定义作用域的配置器；

多个容器后处理器要指定order，需要实现接口ordered。

### 8.1.4 属性占位符配置器

PropertyPlaceholderConfigurer容器后处理器负责读取properties文件中的key-value，可以将这些属性值设置为XML文件中的数据；将SpringXML文件中的经常需要修改的配置放到properties中。



Spring将从指定的属性文件中搜索${}中的内容，为其中的key设置为value内容；如果XML文件导入了context:命名空间，那么：

<context:property-placeholder location=”classpath:db.proeprties”>是PropertyPlaceholderConfigurer的简化配置。

### 8.1.5 重写占位符配置

PropertyOverrideConfigurer后处理，属性文件指定的信息可以直接覆盖Spring的XML配置文件中的元数据。属性key-value的格式是beand.property=value,beanid必须是配置文件的中的id，property是其属性；



<context:property-override location=”classpath:db.properties”>是它的简化配置。

## 8.2 Spring的“零配置”支持

Annotation代替XML。

### 8.2.1 搜索Bean类

Spring使用注解代替XML配置时，需要在主XML文件中指定Bean的搜索路径，就是包；Bean要使用注解标识，表示是bean：@Component（一个普通的bean），@Controller(控制器组件Bean)，@Service(业务逻辑组件Bean)，@Repository（Dao组件Bean）；@Component比较通用，@Controller，@Service，@Repository，未来还会更深入，尽量使用它们代替@Component，需要用到context命名空间：<context:component-scan base-package=*"com.zyx.spring. bean.annotation"*/>，使用@Component标注后，容器默认生成的bean的id为类名首字母小写，@Component(“name”)可以自行指定名字；还可以为组件扫描器添加<include-filter.../>包含bean的规则，<exclude-filter>不包含的规则，type指定过滤器类型（annotation，assignable，regex，aspectj），expression指定表达式。



### 8.2.2 指定Bean的作用域

@Scope注解提供作用域，默认值是singleton，@Scope(“prototype”)，也可以自己指定域解析策略，实现ScopeMetadataResolver接口，在配置扫描器时配置。

### 8.2.3 使用@Resource配置依赖

@Resource解决注入依赖的问题，name属性指定注入的Bean id，放在setter方法上，就是将这个Bean作为参数传到方法里；注入到属性上，就是通过反射的Field操作注入给这个属性；name属性可以省略，省略时，id与方法去掉set在把第一个字母小写后的名字一致，会注入，修饰属性时，就是名字完全一致才会注入。

### 8.2.4 使用@PostConstruct和@PreDestroy定制生命周期

@PostConstruct与XML的init-method属性一样，修饰方法在Bean依赖注入后执行；

@PreDestroy与destroy-method一样，修饰方法在bean销毁前执行。

### 8.2.5 Spring3.0 新增的注解

@DependsOn强制初始化其他Bean,可以修饰Bean类或者方法比如@DependsOn({“aaa”,”bbb”})；@Lazy(true|false)指定是否预初始化。

### 8.2.6 Spring 4.0 增强的自动装配和精确装配

Spring提供了@Autowired注解来指定自动装配，@Autowired可以指定方法、实例变量或者构造函数；默认采用byType的策略。如果找到多个这个类型的Bean，报错处理，没找到就不会执行；多个参数也照找不误。

修饰数组或者set或者list时，容器会把所有的这个类型的Bean组合成数组、set、list；不会报错；组合类型的参数也是这个策略；但是list，set必须指定泛型，也就是类型。@Autowired可以根据泛型完成自动装配，它的功能很强大。

Class BaseDaoImpl<T> implements BaseDao<T>{}

Class UserDaoImpl extends BaseDaoImpl<User> implements UserDao{}

Class ItemDaoImpl extends BaseDaoImpl<Item> implements ItemDao{}

Class BaseServiceImpl<T> implements BaseService<T>{

@Autowired

Private BaseDao<T> dao;

}

Class UserServiceImpl extends BaseSerrviceImpl<User> implements UserService{}

Class ItemServiceImpl extends BaseServiceImpl<Item> implements ItemService{}

这种方式就是综合使用了JAVA的多态与泛型的规则。完成自动注入。

@Qualifier注解是byName的自动装配，根据id。用处不大。

## 8.3 资源访问

资源就是文件等，URL访问网络，IO访问本地文件；功能不行；Spring集成了Resource接口，里面很多针对各种资源的访问实现，功能强大；

GetInputStream()，返回资源的输入流；

Exits() 返回Resource指向的资源是否存在；

IsOpen() 返回文件是否打开；

GetDescrition() 返回资源的描述信息；

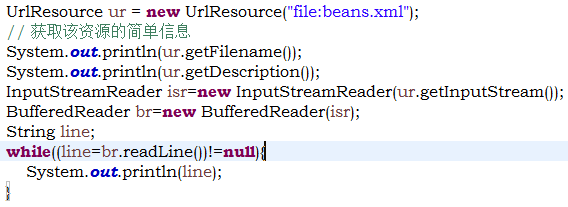
GetFile() 返回资源的File对象；

GetURL()返回资源对应的URL对象。

### 8.3.1 Resource实现类

UrlResource：访问网络资源的实现类；ClassPathResource：访问类加载路径资源的实现类；FileSystemReosurce：访问文件系统资源的实现类；ServletContextResource：访问相对与ServletContext路径下的实现类，ImputStreamResource：输入流资源的实现类；ByteArrayResource：字节数组资源的实现类；

1. 访问网络资源：UrlResource是java.net.URL的封装，网络资源需要提供协议，file:本地文件系统，http:网络，ftp:文件服务器。

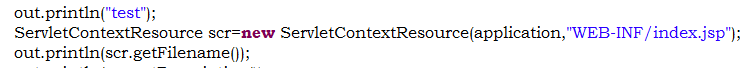


1. 访问类加载路径下的资源：ClassPathResource访问类加载路径下的资源，如果Web中字符串路径含有classpath:字样，将自动创建ClassPathResource对象。
2. 访问文件系统资源：



Spring执行某个方法时，接收代表资源路径的字符串参数，当识别到参数中包含file:前缀，系统自动创建FileSystemResource对象。

1. 访问应用相关的资源：ServletContextResource访问Web Context下路径下的资源，构造器接受一个代表资源位置的字符串参数，该资源是相对于Web Context应用根路径的位置；File也可以访问这种资源，但是要先使用getRealPath()获得资源的绝对路径；



1. 访问字节数组资源：InputStreamResource，访问二进制输入流，是对InputStream的Resource实现，InputStreamResource总是被打开，ByteArrayResource是字节流可以代替InputStreamResource，ByteArrayResource使用字节数组创建，封装了对于字节数组的访问。

### 8.3.2 ResourceLoader接口和ResourceLoaderAware接口

Spring有关与Resource的接口：ResourceLoader、ResourceLoaderAware；ResourceLoader实现类的实例可以获得一个Resource实例，ResourceLoaderAware接口实现类将获得一个ResourceLoader的引用；ResourceLoader接口里有如下方法：Resource getResource(String location)，用于返回Resource实例，ApplicationContext都实现了ResourceLoader接口，可以使用它获得Resource实例，获得的Resource资源访问策略与ApplicationConext创建时的资源访问策略相同；如果想不用创建ApplicationContext是的资源访问策略，可以在location字符串中指定前缀，”classpath:”使用ClassPathResource资源访问策略，“file”使用FileSystemResource资源访问策略，http:使用UrlResource资源访问策略。

ResourceLoaderAware与BeanFactoryAware，BeanNameAware等一样，ResourceLoader作为setResourceLoader方法的参数传入，是由Spring容器自动完成的，传入Spring容器自身，因为ApplicationContext的实现类都实现了ResourceLoader接口。

### 8.3.3 使用Resource作为属性

容器中的Bean需要访问资源有2种方法：1.在代码中获取Resource实例，代码耦合性高，不利于后期修改；2.使用依赖注入，在XML中配置资源；定义一个Resource属性，property属性的value中指定位置，不加入资源访问策略使用ApplicationContext创建时的资源访问策略，加前缀指定资源访问策略。

### 8.3.4 在ApplicationContext中使用资源

创建ApplicationContext需要指定配置文件，就需要访问资源；ApplicationContext允许加载多个配置文件；ApplicationContext的资源访问策略：使用ApplicationContext的实现类指定访问策略；使用前缀指定资源访问策略。

1. 使用ApplicationContext的实现类指定资源访问策略：ClassPathXmlApplicationContext：对应使用ClassPathResource进行资源访问，FileSystemXmlApplicationContext：使用FileSystemResource进行资源访问，

XMLWebApplicationContext：使用ServletContextResource进行资源访问。

1. 使用前缀指定访问策略：比如ApplicationContext ctx=FileSystemXMLApplicationContext(“classpath:beans.xml”)虽然使用的是FileSystem，由于指定了classpath，也将在类搜索路径下搜索，这个只对当前这一次资源访问有效，如果其他访问没有指定前缀，则仍是ApplicationContext的创建时的访问方式。
2. Classpath\*:前缀的用法：这个前缀可以提供加载多个XML文件的能力，使用这个前缀时，ApplicationContext将从类路径下搜索所有这个名字的文件，名字也可以是通配符，加载配置文件的定义，最后合并为一个ApplicationContext。XML文件定义重复也没关系，classpath只会搜索类路径下的第一个文件，搜到就返回；classpath\*:仅对ApplicationContext有效，不可用于Resource；另外一种加载多个配置文件的方法就是通配符。
3. File前缀：FileSystemXmlApplicationContext作为ResourceLoader使用时，会让绑定的FileSystemResource实例把绝对路径都当成相对路径来处理，不管是否以/开头；要使用file前缀来指定绝对路径。

## 8.4 Spring的AOP

AOP Aspect Orient Programming，面向切面编程。从动态角度考虑程序进程，面向对象编程是横向的，面向切面编程是分层思想的一种补充，在层次间完成一定的操作。

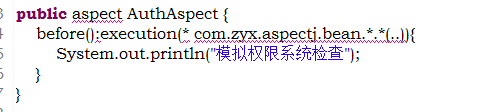
### 8.4.1 为什么需要AOP

AOP专门用于处理系统中分布于各个模块中的交叉关注点的问题，常常把AOP来处理一些具有横切性质的系统级服务，如事务管理、安全检查、缓存、对象池管理等。

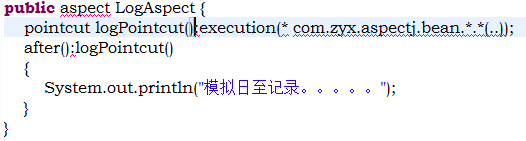
### 8.4.2 使用AspectJ实现AOP

AspectJ是Java语言的一个AOP实现，包含2个部分：1.定义了表达定义AOP的语法规范；2.工具，包括编译器调试工具等；使用AspectJ的方法如下：

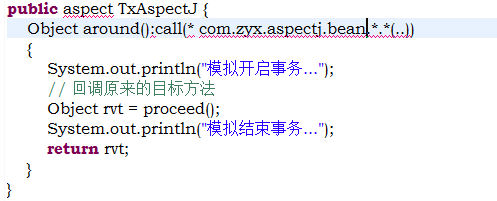




定义的这个AspectJ指定作用域在com.zyx.aspectj.bean包中任何类(\*)任何方法(\*)，不论什么任意参数(..)，任意返回类型(\*)；使用的是AspectJ能够识别的关键字aspect，这个是指定在这些方法执行前需要做的操作。



模拟日志记录，Pointcut logPointcut相当于给后面的execution起了个别名，可以在后面复用。



这是模拟事务的代码，其中的proceed()方法将代表原来的目标方法。

开发者无序修改代码，Aspect在编译成class文件的时候添加上了这些功能，所以AspectJ也叫做编译时增强的AOP框架，AOP的目的：程序员不修改源代码的情况下，为系统中业务组件添加通用的方法，本质是AOP框架本身去修改了代码文件；静态AOP：编译阶段对程序修改，以AspectJ代表；动态AOP实现：AOP框架在运行阶段动态生成AOP代理。

### 8.4.3 AOP的基本概念

AOP框架不与代码耦合：各步骤之间具有良好的隔离性、源代码无关性；

切面（aspect）：切面用于组织多个Advise，Advise放在切面中定义；

连接点（Joinpoint）：程序执行过程中明确的点，如方法的调用，异常的抛出等；

增强处理（Advise）：AOP框架在特定的切入点执行的增强处理；

切入点（Pointcut）：可以插入增强处理的连接点；当某个连接点满足指定要求时，连接点也就四切入点。Spring使用AspectJ切入点语法。

引入：将方法或字段添加到被处理的类中；

目标对象：被AOP处理增强的对象；

AOP代理：AOP框架创建的对象，JDK动态代理是实现相同接口，chlib动态代理是用子类继承；

织入：将增强处理添加到对象中，创建出来一个被增强的对象。

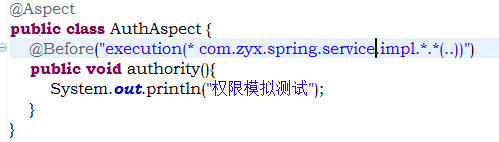
### 8.4.4 Spring的AOP支持

SpringAOP由容器负责生成，以其他Bean为目标，默认的策略是是JDK动态代理，即实现相同的接口；也可以使用cglib代理代理类而不是接口；Spring只支持将方法作为连接点，成员变量的访问与更新也作为连接点则使用AspectJ，Spring可以无缝的整合Spring AOP、IoC、AspectJ；AOP编程3个部分：定义普通业务逻辑组件、定义切入点、定义增强处理；AOP代理的方法=增强处理+目标对象方法，定义切入点即增强处理：注解方式，XML管理方式。Spring有自己的定义切面即增强处理的方式，但也能识别AspectJ的方式。

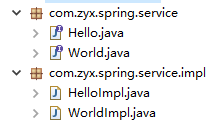
### 8.4.5 基于注解的零配置方式

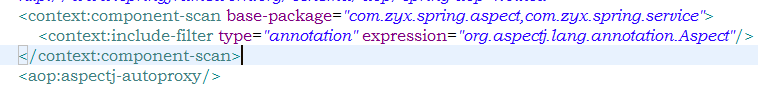
使用注解定义切面、切入点、增强处理等，注解可以是AspectJ的形式，但是底层的处理仍是Spring AOP，并且仍是运行时动态代理而不是编译时增强，Spring的主配置文件中要将加入:<aop:aspectj-autoproxy>，这个代表根据切面自动生成代理，如果不打算使用XML，加入AnnotationAwareAspectJAutoProxy Creator来启动对于注解的解析支持，这是一个Bean后处理器，它与<aop:aspectj-autoproxy>的作用是相同的；将为Bean生成AOP代理；同时添加aspectjweaver.jar与aspectjrt.jar与aopalliance.jar支持。

1. 定义切面Bean：Bean使用@Aspect标志后，Spring会自动将该Bean做切面处理类，里面会定义切面及对应的增强处理，当用@Aspect修饰后，Spring不会将他作为组件Bean处理，Bean后处理器也不会对它进行处理（没什么用啊）。
2. 定义Before增强处理：



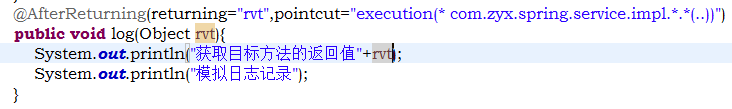
定义的2个业务方法都实现了接口；



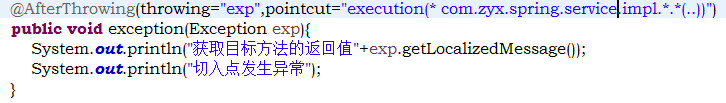


配置搜索切面类及组建类。Before总是在目标方法前执行，如果要阻止目标方法执行，可以抛出异常。Before增强处理无法访问目标方法的返回值。

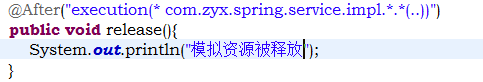
1. 定义AfterReturn增强处理：目标方法完成后织入，属性：Pointcut/value指定切入点的切入正则表达式，可以是一个已有的切入点也可以直接定义；returning：指定一个形参名，表示方法中可以定义一个这个名字的形参，形参的类型限制了切入点的返回类型必须是这个类型，不满足的匹配不上切入点，返回的对象赋给这个形参，不能改变值。



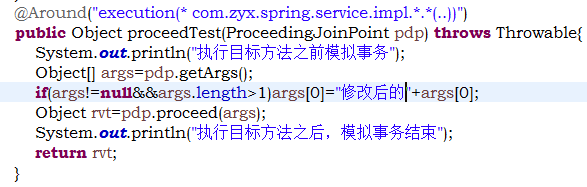
1. 定义AfterThrowing增强处理：@AfterThrowing注解用于处理程序中的异常；指定的属性Pointcut/value与上面相同，另外一个属性为throwing：指定一个形参名，表示增强处理方法中可以定义一个这个名字的形参，形参的类型限制了切入点发生的异常类型必须是这个类型，不满足的匹配不上切入点，异常对象赋给这个形参，不能改变值。@AfterThrowing虽然处理了异常，但是异常仍会被抛到上一层，并不是完全的处理。



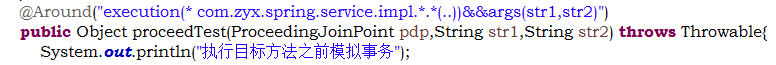
1. After增强处理：与@AfterReturn类似，@AfterReturn只在目标方法成功返回后才执行，@After不论如何都执行，与fianlly类似，所以常用与资源回收；@After只用指定Pointcut/value即可。



1. Around增强处理：功能强大，是@Before与@Afterreturning的总和，只有一个value属性指定切入点，增强处理方法的第一个形参必须是ProceedingJoinPoint参数，至少一个参数；ProceedingJoinPoint.Proceed()方法调用就是执行目标函数，可以传入参数，增强处理方法内部也可以获得参数，进行简单处理后传入proceed()中；参数是对象数组的形式，当参数个数或者类型对应不上时会抛出异常。



1. 访问目标方法的参数：定义增强处理方法时，第一个参数设置为JoinPoint类型，Object[] getArgs()获取执行目标方法时的参数，Signature getSignature()返回被增强的方法的相关信息，Object getTarget()返回被织入增强处理的目标对象，Object getThis()返回AOP框架为目标对象生成的代理对象。所有的增强处理方法都可以通过JoinPoint来访问被增强的目标对象、目标方法与方法的参数，织入增强方法的顺序是根据优先级，没有优先级则是随机的，指定优先级可以通过让切面类实现Ordered接口，或者直接用@order注解，都可以指定int值，越小优先级越高。访问参数还有另外一种办法，就是&&args表达式。



指定增强方法的形参名，匹配时根据形参名的形参类型匹配对应形参类型的方法，按照参数的顺序将参数赋给增强方法的参数；注意参数类型的匹配而不是参数名字。

1. 定义切入点：为切入点表达式起名字，方便复用；切入点包含2部分：切入点表达式，一个包含方法名字和任意参数的方法签名；



定义一个切入点表达式，返回值必须为void，方法体随意；必须用@Pointcut注解，这个名字的作用范围根据访问修饰符确定，当为public时，外部切面类使用时，前面要加上这个切入点所在的类名。

1. 切入点指示符：切入点指示符只是在什么地方织入增强处理，execution就是一个；Spring AOP 只支持方法作为切入点，所有的指示符都是作用与方法的；Spring只支持部分AspectJ切入点；execution：匹配方法连接点，通常是通过通配符指定方法名，格式如下：

Execution(modifiers-pattern(方法的修饰符可省略)Ret-type-pattern(返回类型) declaring-type-pattern(类路径，可省略) name-pattern(param-pattern)(方法签名)throws-pattern(指定抛出的异常))，都要匹配；within：匹配特定类的连接点，在Spring AOP中与execution使用重叠，比如within(com.zyx.serivce.\*)，匹配这个包下任意类的任意方法；this：限定AOP代理必须是指定类型的实例；target：与this用法重叠；args：根据方法的参数匹配连接点，进行织入，参数必须是指定类型，；bean：用于限定匹配指定Bean实例内的连接点，需要传入bean的id或者name，可以明确指定为哪个Bean织入增强处理，这个很重要。

1. 组合切入点表达式：切入点条件表达式支持逻辑运算符，&& 与，|| 或，！ 非。

### 8.4.6 基于XML配置文件的管理方式

Aop命名空间来定义切面、切入点与增强处理；注解的优点就是能被Spring AOP与AspectJ同时支持，而XML只被Spring AOP支持；XML中所有的切面、切入点、增强处理定义在元素<aop:config>中；首先配置切面使用元素<aop:aspect> ，将一个Spring Bean（java类）转换成切面Bean，指定的属性id切面标识名，ref引用Bean的id，order织入的优先级；配置增强处理：与注解的增强处理方式一致，支持Before(<aop:before>)、After()、AfterReturning()、AfterThrowing()、Around()5种增强处理，这些元素可以指定属性pointcut（指定切入点表达式）、pointcut-ref、method（将切面Bean的这个方法转换为增强处理）、throwing（指定抛出的异常的形参名）、returning（指定返回的对象名字）。切入点指示符与注解的相同；组合表达式使用and、or、not进行逻辑处理；



配置切入点：定义切入点表达式是为了重用切入点表达式<aop:pointcut>定义，放在<aop:config>下是全局切入点，放在<aop:aspect>下是单一切面的切入点；属性id，切入点标识名；expression切入点关联的表达式；



## 8.5 Spring 3.1新增的缓存机制

Spring 3.1新增了缓存，可以对任意Bean及方法缓存，还能缓存查询的结果；Spring缓存底层是其他缓存工具，Spring提供了缓存的API，是要实现了API，可以接入任意缓存工具。

### 8.5.1 启用Spring缓存

Spring配置文件的缓存配置是<cache:>元素；启用Spring缓存需要2步：1.在Spring配置文件中添加<cache:annotation-driven cache-manager=”缓存管理器ID”/>，指定根据主句来启用什么级别的缓存，2.针对不同的缓存实现配置对应的缓存管理器；缓存管理器的默认值是cacheManager。

1. Spring内置的缓存配置实现：简单测试，只是内存缓存，使用的缓存管理器ID=cacheManager，缓存实现是SimpleCacheManager；
2. EhCache缓存实现的配置：首先添加Jar包；配置ehcache.xml缓存配置文件，缓存管理器的实现类是EhCacheCacheManager。

### 8.5.2 使用@Cacheable执行缓存

修饰类就是类级别缓存，修饰方法就是方法级别缓存。

1. 类级别缓存：适用于全部方法，只要访问的函数签名一致，将不执行方法直接返回结果；
2. 方法级别缓存：；

### 8.5.3 使用@CacheEvict清除缓存

## 8.6 Spring的事务

Spring事务管理不与任何特定的API耦合，基于AOP实现。

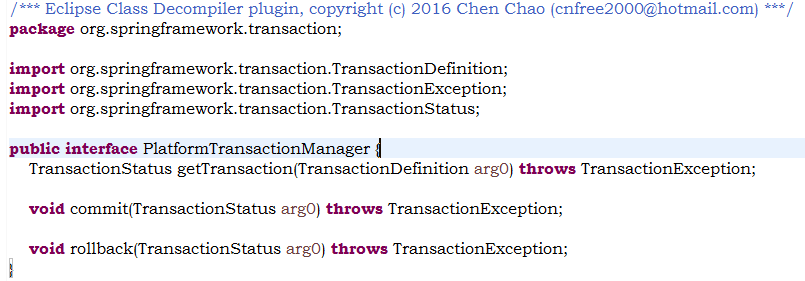
### 8.6.1 Spring支持的事务策略

Java EE的事务策略有2种：全局事务与局部事务；全局事务由应用服务器管理，是大范围横块多个模块的事务；局部事务与具体的底层实现技术有关，是单一模块的事务；比如JDBC下，Connecttion可以操作一个事务，Hibernate下，Session可以操作一个事务。

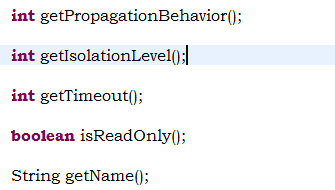
JTA全局事务的模式：Transaction tx=ctx.lookup、业务实现、if 正常 tx.cmmit

If 失败 tx.rollback()；JDBC局部事务,Connection conn=getConnection(),if 正常 conn.commit if 失败 conn.rollback()；Hibernate事务：Session s=getSession() Transaction tx=s.beginTransaction() if 正常 tx.commit if 失败 tx.rollback()。

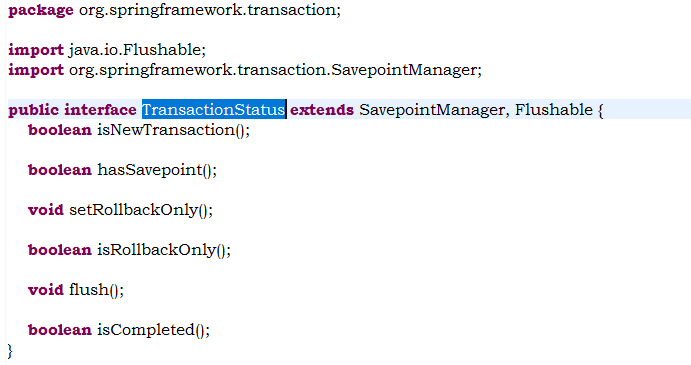
这种事务控制，事务代码与业务代码耦合在了一起，不利于事务策略的变化，当底层技术改变时，这些代码也需要改变。Spring对事务管理进行了抽象，抽象出了所有事务都具有的操作，具体的底层技术只需要实现这个Spring的抽象接口，Spring维护的代码就可以不做改变，Spring的抽象事务接口完成了事务管理与业务代码的解耦和。这个接口是PlatformTransactionManager。



在这个接口内，getTransaction方法相当于开始事务，返回一个TransactionStatus对象表示一个事务关联到当前的线程上，如果当前线程已经关联事务返回的是这个事务，没有就是新建的；参数TransactionDefinition是一个接口，必须指定事务的一些规则。在接口内部定义了get方法。



TransactionStatus代表事务本身，接口提供了一些操作事务的方法。



实现了PlatformTransactionManager接口的事务管理类，就像普通bean一样配置就可以，只是底层的实现类不同，具体的配置方式也有一些不同。

Spring提供2种方式完成事务管理：1.硬代码编写，获取容器中的事务实现类，用调方法的方式完成事务控制，这种方式与传统方式几乎没有区别；2.声明式事务管理，就是AOP，为需要进行事务控制的地方织入相关的事务控制方法。这就是面向接口编程的优势了。

### 8.6.2 使用XML Schema配置事务策略

Spring的声明式事务控制，完成了事务与业务逻辑解耦和。Spring提供了tx:命名空间来配置事务管理，元素<tx:advice />进行事务控制处理；<aop:advisor>启用事务代理。<tx:advice />元素需要指定具体的PlatformTransactionManager实现类，还需要配置<attributes>元素，<attributes>包含多个<method>元素，每个<method>元素都为一类特定的方法指定了事务规则定义。就是TransactionD efinition定义的一些规则，配置方法如下：



此时，切点出的bean请求将会被加入事务代理的bean所代替。

<aop:advisor>将Advice与切点绑定时，实际是由Spring的bean后处理器完成的，BeanNameAutoProxyCreator与DefaultAdvisorAutoProxyCreator。通过配置这种增强处理与切点之间的管理，可以为不同的逻辑方法指定不同的事务控制策略，可以指定遇到特定的异常时回滚或者不回滚事务。

### 8.6.3 使用@Transactional

@Transactional就是事务配置，与XML配置方式相同，可以放在bean上，对整个bean起作用，方法上，对该方法起作用，位置相当于定位了切点，与切点进行了管理；一些具体的属性就是一些事务规则定义了。

## 8.7 Spring 整合Struts2

### 8.7.1 启动Spring容器

Spring有2种创建方式：1.在Web.xml中定义监听器随web启动而启动；2.利用第三方MVC框架的扩展点，创建Spring容器。Spring容器随Web应用启动而启动需要实现ServletContextListener监听器接口的监听器就可以，这个监听器的方法将在Web启动时，由应用服务器回调，此时可以用来初始化Spring的容器，Spring实现ServletContextListener接口的类是ContextLoaderListener，这个类继承了ContextLoader这个类，他在创建时会自动查找WEB-INF/下的applicationContext.xml文件，如果只有这一个则无需任何配置。如果有多个XML配置，需要使用<context-param>配置XML所在路径，固定会查找带有contextConfigLocation的初始化参数，然后监听器将会创建WebApplicationContext对象，保存在ServletContext对象中，代码如下：

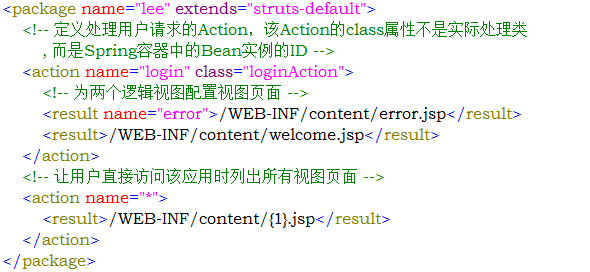
servletContext.setAttribute(WebApplicationContext.***ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE***, **this**.context);需要获取这个对象可以通过ServletContext对象的getAttribute方法，推荐使用WebApplicationContextUtils.getWebApplicationContext()方法来获取。

### 8.7.2 MVC框架与Spring整合的思考

MVC接收请求，调用业务逻辑组件完成处理，控制器该如何获取业务逻辑组件呢？New创建造成代码耦合度高；有服务器定位模式，适用于远程场景；还有工厂模式，业务逻辑组件由工厂负责产生，对于Spring来说，Spring就是这个工厂，向控制器返回所有的需要的业务逻辑组件。控制器如何访问Spring工厂中的业务逻辑组件有2种策略：1.Spring容器负责管理控制控制器，并利用依赖注入为控制器注入业务逻辑组建；2.利用Spring的自动装配，Action可以自动的从Spring容器中获取所需的组件。

### 8.7.3 让Spring管理控制器

Struts2的核心控制器拦截到用户请求，将请求转发给Action处理，Struts2将负责创建Action实例，并调用其execute()方法，这个过程固定，现在Action由Spring处理，就不能有Struts2产生，Struts2提供了一个与Spring整合的jar包，加入了这个jar包后，在strusts.xml配置文件中，class部位不能写class类名了，要写为Action在Spring中的bean id。也可以配置method，都可以使用通配符。



Spring管理Struts2的Action后，scope一定不能配成singleton，因为Action里包含了请求的状态信息，必须一个请求生成一个Action，否则数据可能会混乱掉。

### 8.7.4 使用自动装配

此时Action由Struts2本身创建，Struts2的配置文件与不使用Spring的时候没有区别，仍需指定Action的class，请求过来后Struts的拦截器负责创建Action，Struts2提供的Spring插件负责将Spring工厂中的业务逻辑组件注入到创建的Action中，默认的装配策略是byName，可以指定为byType、auto、constructor等。

## 8.8 Spring整合Hibernate

### 8.8.1 Spring提供的DAO支持

DAO组件是java EE中的重要组成部分，Spring针对底层操作提供了很多的抽象类与接口，提供了规定的操作，只要底层的ORM提供了实现类就可以整合到Spring中。

### 8.8.2 管理Hibernate的SessionFactory

Hibernate的sessionFactory配置可以配置在XML文件中，Spring可以生成这个Bean，并可以将这个bean注入到其他的bean组件中，还可以访问容器数据源。

### 8.8.3 实现DAO组件的基类

调用sessionFactory的getCurrentSession()方法获取Session对象。

### 8.8.4 传统的HibernateTemplate和HibernateDaoSupport

Spring提供了这2个类，HibernateTemplate为持久层访问模板，包含基本的所有的访问方法，传入一个sessionFactory即可工作那种；HibernateDaoSupport是抽离出来的DAO的基本功能。

### 8.8.5 实现DAO组件

### 8.8.7 使用声明式事务

## 8.9 Spring整合JPA

# 企业应用开发的思考和策略