Spring 的國際化機制

·首先對spring進行配置

<!-- SpringFramework configurations -->

<context-param>

<param-name>contextConfigLocation</param-name> <param-value>/WEB-INF/context/spring.\*.beans.xml</param-value>

</context-param>  
<!-- SpringFramework context loader -->

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

* 對spring的message進行配置

<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource" scope="singleton">

<property name="basename" value="messages"/>

</bean>

<bean id="localeResolver"

class="org.springframework.web.servlet.i18n.CookieLocaleResolver" scope="singleton">

<property name="cookieName" value="iPLM\_Locale"/>

<property name="cookieMaxAge" value="31536000"/>

</bean>

<bean id="localeChangeInterceptor"

class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor" scope="singleton"/>

**[Spring2.5的国际化配置](http://coolyongzi.javaeye.com/blog/218560)**

引用出处：http://hi.baidu.com/renyijiu/blog/item/601251f92054c95c252df2c6.html   
  
  
  
基于浏览器语言的**国际化**配置   
  
使用**Spring**的MVC，并且配置中有配置Resource文件   
  
<!-- 资源文件绑定器 -->           
<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">                   
  
<property name="basename" value="message-info" />                 
<property name="useCodeAsDefaultMessage" value="true" />         
  
</bean>   
  
  
其中，message-info是你的properties文件的通用名。如：我的配置文件叫message-info.properties,message-info\_zh\_CN.properties等等   
  
只要有了这个配置，然后配置JSP渲染器为JSTL支持的，那么在你的JSP文件中使用fmt标记就可以实现客户浏览器语言**国际化**了。   
如：<fmt:message key="info.login.title" />   
其中的info.login.title和你的资源文件对应   
  
二、基于动态加载的**国际化**配置   
  
1、基于请求的**国际化**配置   
  
基于请求的**国际化**配置是指，在当前请求内，**国际化**配置生效，否则自动以浏览器为主。   
  
配置方式如下：   
首先配置拦截器   
<!-- **国际化**操作 拦截器 必需配置，可以和其它**国际化**方式通用 -->         
  
<bean id="localeChangeInterceptor" class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor" />   
  
  
  
这个配置呢，是不论请求级别的**国际化**，还是Cookie级别的**国际化**，再或者Session级别的**国际化**，都必需有配置这个拦截器，否则会不能使用。   
  
配好上面的拦截器之后，就将拦截器注入到你的UrlHandlerMapping中，例如:   
<bean id="defaultUrlMapping" class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">                   
  
<property name="interceptors" ref="localeChangeInterceptor" />                   
  
<property name="order">                           
<value>1</value>                 
</property>         
  
</bean>   
  
  
  
这个时候，但凡有了符合UrlMapping的请求，就会被拦截，并且开始配置**国际化**参数   
         <bean id="localeResolver" class="org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver">         </bean>   
  
  
默认的参数名为locale主意大小写。里面放的就是你的提交参数。如:en\_US，zh\_CN之类的，这个时候，你在页面上加一句<a href="?locale=zh\_CN">简体中文</a>   
如果你的资源中，饱含建议中文的配置，那么就会变成你确定的简体中文拉。   
  
2、基于Session的**国际化**配置   
  
拦截器和基于请求的相同   
  
Session的配置如下：   
         <bean id="localeResolver" class="org.springframework.web.servlet.i18n.SessionLocaleResolver">         </bean>     
  
  
在你的处理的Controller中，将提交上来的locale字段信息生成真正的Locale对象，然后将对象保存在Session中，默认保存的ID是SessionLocaleResolver.LOCALE\_SESSION\_ATTRIBUTE\_NAME   
  
  
  
这样，当你的Session不过期，那么**语言种类**始终保持正确的说。我一直是这样子用的，我觉得还是Session的好，老外们用了很满意。   
  
3、基于Cookie的**国际化**配置   
  
这个我就不说了，反正用的不多，至少我做的项目能不用Cookie就不用Cookie，所以，基于Cookie的**国际化**配置我就不细说了，如果想知道怎么配置，那么下载一个**Spring**，其中的例子程序就是用Cookie配置的，你自己读代码就OK了。   
  
三、注意事项   
  
如果不用默认的浏览器语言**国际化**方式，那么拦截器一定要配置，如果你有多个UrlMapping，那么就每个都配上拦截器。   
至于配置的LocaleResolver的名字，一定要用上面的配置中的名字localeResolver当然了，这个是默认的名字来的，自己设置成别的也可以，但是就是麻烦，反正我用默认的就感觉不错   
  
  
  
解决问题:   
在前几天引用“**Spring** 的MVC I18N－**国际化**相关配置 ”并做了测试，发现 有一问题。程序运行会抛出异常   
  
“Cannot change HTTP accept header - use a different locale resolution strategy”，根本原因是**spring** source 做了限制,源码如下   
  
public class AcceptHeaderLocaleResolver implements LocaleResolver {   
  
public Locale resolveLocale(HttpServletRequest request) {   
   return request.getLocale();   
}   
  
public void setLocale(HttpServletRequest request, HttpServletResponse response, Locale locale) {   
   throw new UnsupportedOperationException(   
     "Cannot change HTTP accept header - use a different locale resolution strategy");   
}   
  
}   
  
请注意上面的类，该类允许继承，所以需要改写setLocale方法，   
源码示范如下   
  
package org.springframework.web.servlet.i18n;   
  
import java.util.Locale;   
  
import javax.servlet.http.HttpServletRequest;   
import javax.servlet.http.HttpServletResponse;   
  
import org.springframework.web.servlet.DispatcherServlet;   
import org.springframework.web.servlet.LocaleResolver;   
  
public class MyAcceptHeaderLocaleResolver extends AcceptHeaderLocaleResolver {   
private Locale myLocal;   
public Locale resolveLocale(HttpServletRequest request) {   
   return myLocal;   
}   
public void setLocale(HttpServletRequest request, HttpServletResponse response, Locale locale) {   
   myLocal = locale;   
}   
     
}   
然后在action-servlet.xml里的设置为   
  
<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">   
   <property name="basename" value="message" />   
</bean>   
<bean id="myViewController" class="org.springframework.web.servlet.mvc.UrlFilenameViewController" />   
<bean id="filenameController" class="org.springframework.web.servlet.mvc.UrlFilenameViewController" />   
<bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">   
<property name="interceptors" ref="localeChangeInterceptor"/>   
   <property name="mappings">   
   <value>   
     chinese.do=filenameController   
     us.do=filenameController   
   </value>   
   </property>   
</bean>   
<bean id="defaultHandlerMapping" class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping" />   
<bean id="localeChangeInterceptor" class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor" />   
<bean id="localeResolver" class="org.springframework.web.servlet.i18n.MyAcceptHeaderLocaleResolver"/>   
  
<bean id="viewResolver";   
   class="org.springframework.web.servlet.view.InternalResourceViewResolver">   
   <property name="viewClass"   
   value="org.springframework.web.servlet.view.JstlView" />   
   <property name="prefix" value="/WEB-INF/jsp/" />   
   <property name="suffix" value=".jsp" />   
</bean>

**Spring Web MVC架構**  
  
  
**以前端控制器(Front Controller)做先發**  
Spring Web MVC架構讓user不直接與所提供服務的目的端連接，而是先透過**前端控制器(Front Controller)**把守第一道關卡，再由前端控制器將請求**分派(Dispatch)**給一個相對應的**控制器(Controller)**。而這個前端控制器是由Spring所提供的org.springframework.web.servlet.**DispatcherServlet**來負責，望文生義，我們可以知道這個前端控制器其實是一個包裝過的Servlet。因此，我們要將DispatcherServlet定義在**web.xml**裡面。  
  
web.xml:  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
...(略)  
 <servlet>  
         <servlet-name>dispatcherServlet</servlet-name>  
         <servlet-class>org.springframework.web.servlet.**DispatcherServlet**</servlet-class>  
          <init-param>  
                    <param-name>**contextConfigLocation**</param-name>  
                    <param-value>**/WEB-INF/mvc-config.xml**</param-value>  
           </init-param>  
 </servlet>  
 <servlet-mapping>  
          <servlet-name>dispatcherServlet</servlet-name>  
          <url-pattern>**\*.do**</url-pattern>  
 </servlet-mapping>   
...(略)  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
先設定一個**DispatcherServlet**的bean實例，並且指定其設定檔位置，本例是以mvc-config.xml為例，若不指定，則會以「此servlet名稱-servlet.xml」來命名。若有超過一個以上的**contextConfigLocation**定義檔要設定，則在其值使用**逗號分隔**。在servlet-mapping對應上，所有以**.do**結尾的請求檔案，都會交由此前端控制器來處理，再進行分派給控制器。  
  
**各司其職的控制器(Controller)做中繼**  
  
由DispatcherServlet (Front Controller)將請求分派給控制器(Controler)，而這些Controller必須要實作org.springframework.web.servlet.mvc.**Controller**介面，而此介面有一個**handleRequest()**方法必預要實作其內容。handleRequest()方法的原型涵數會傳入一些與Servlet相關的API，如HttpServletRequest , HttpServletResponse 等。其定義如下：  
  
Controller介面定義:  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
public interface Controller {  
        **ModelAndView** handleRequest(HttpServletRequest request, HttpServletResponse response) throws Exception;  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
  
當一個Controller收到DispatcherServlet的請求時，會去執行其handleRequest()方法，處理完成後會回傳一個**ModelAndView**物件，這個物件是要給MVC中的View層做為處理後結果呈現用的相關訊息，當然他本身也決定了要給哪一個View來接手。可以在ModelAndView物件中，包裝一個Map集合物件做為各種參數或結果的傳遞，我們現在只先關心產生這個ModelAndView物件需要傳入什　樣的參數，然後把這個物件回傳給Spring容器，後續如何交由View來處理，我們之後再繼續談。  
  
  
**ModelAndView物件**  
產生ModelAndView物件，有幾種不同的constructor可以使用，基本上最少要給定一個「目的View」的物件或者其所代表的View名稱字串。  
  
有三種較常用的Constructor，Spring API定義如下：  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
**ModelAndView(String viewName)**          Convenient constructor when there is no model data to expose.   
          處理完後不帶結果給View，只單純做頁面轉發的動作。  
  
**ModelAndView(String viewName, Map model)**   
          Create a new ModelAndView given a view name and a model.   
          處理完後，將結果存入一個Map物件中，一起轉發到View中呈現。  
  
**ModelAndView(String viewName, String modelName, Object modelObject)**          Convenient constructor to take a single model object.   
          若是要傳送的結果只是單一的物件，則可使用這個方法來轉發到View。  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
  
我們建立一個自定義的Controller名為HelloController，其實作了Controller介面，並且實作裡面的handleRequest()方法，一個簡單的範例如下：  
HelloController.java:  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
public class HelloController implements Controller {  
        private String viewPage;  
  
        public ModelAndView **handleRequest**(HttpServletRequest request, HttpServletResponse response) throws Exception{  
                String user = request.getParameter("user");  
                **return new ModelAndView(viewPage, "user", user);**  
        }  
  
        public void setViewPage(String viewPage){  
                this.viewPage = viewPage  
        }  
}  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
ps.上面**viewPage**的目的是決定處理完後要轉發的View名稱，使用了Spring的IoC方式，由外部Bean定義檔注入。  
  
**重新導向(redirect)頁面**  
  
在ModelAndView預設是用**轉發(dispatch)**的方式進行頁面的轉移，若想使用**重新導向(redirect)**的方式，可以在產生ModelAndView時，傳入一個org.springframework.web.**servlet.view.RedirectView**的實例。  
ex:  
return new ModelAndView(new **RedirectView**(this.getViewPage()));  
  
在此注意一下兩者的差別，使用**重新導向**功能，viewPage要從「伺服器的根目錄開始」寫，例如：/SpringDemo/pages/index.html，但是不能訪問WEB-INF下的檔案，而且重新導向後網址列會改變。  
  
而**轉發**的方式，可以訪問WEB-INF以下的目錄，而且網址列上的位址也不會改變。  
  
  
  
**在定義檔中定義Controller的mapping**  
  
以上的HelloController需要在Bean定義檔(mvc-config.xml)中定義一個bean的實例，在定義時，不使用"id"屬性，而使用"**name**"屬性來定義這個bean實例，這個name屬性是一個\*.do的pattern對映，例如是**name="/hello.do"**，如此一來若使用者要求一個"/hello.do"的網頁位置(在此不稱作「檔案名稱」，因為這樣的pattern對映只是一個虛擬的對映，其可能是一個完全不存在的名稱而已)，經由前端控制器轉接手使用者的需求後，則可以在mvc-config.xml裡找到他要轉發的目的控制器是哪一個，也就是由"/hello.do"所屬的HelloController控制器。  
要其在Bean定義檔定義如下：  
  
mvc-config.xml:  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
...(略)  
<bean **name="/hello.do"** class="com.aidc.controller.**HelloController**">  
        <property name="**viewPage**">  
                <value>**hello**</value>  
        </property>  
</bean>  
...(略)  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
以上這種直接以bean的name屬性直接對映後可以找到其要接手的Controller，這種方式是Spring預設的Mapping方式，其類別為org.springframework.web.servlet.handler.**BeanNameUrlHandlerMapping**。  
使用這種方式，較沒有辦法統整的了解Bean定義檔中所有需要對映的控制器有哪些。因此可以選用另一種方式來達成對映的工作，其為org.springframework.web.servlet.handler.**SimpleUrlHandlerMapping**，使用方法如下：  
  
mvc-config.xml:  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
...(略)  
<bean id="urlHandlerMapping" class="org.springframework.web.servlet.handler.**SimpleUrlHandlerMapping**">  
        <property name="mappings">  
                <props>  
                        <prop key="/hello.do">**helloController**</prop>  
                        <prop key="/welcome.do">**welcomeController**</prop>  
                </props>  
        </property>  
</bean>  
  
<bean id="**helloController**" class="com.aidc.controller.HelloController">  
        <property name="viewPage">  
                <value>hello</value>  
        </property>  
</bean>  
<bean id="**welcomeController**" class="com.aidc.controller.WelcomeController">  
        <property name="viewPage">  
                <value>welcome</value>  
        </property>  
</bean>  
...(略)  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
ps. 上述mappings的key還可以使用**萬用字元(wildcard)**，例如key="hello\*.do"  
  
  
**目的地View技術元件做終結者**  
  
還記得上述的Controller處理一些邏輯程式之後，回傳了一個ModelAndView給Spring容器吧？這個ModelAndView裡面的第一個參數ViewName由viewPage的值所指定，而這個值又從mvc-config.xml中注入，在上面幾行可以看到，名為**hello**的view名稱。這個看似簡單的hello字串，關係著等一下要把這個處理完的結果轉送給哪一個view來呈現。而把ModelAndView物件回傳回去給Spring容器後，會經由一個所謂的**解析器(Resolver)**來解析，決定要將剛剛的處理結果轉發給哪一個View來處理。  
  
resolver都需要實作org.springframework.web.servlet.ViewResolver介面，而Spring本身也提供了一些已經實作此介面的一些類別：  
  
1. **InternalResourceViewResolver**  
最簡單的resolver是org.springframework.web.**servlet.view.InternalResourceViewResolver**  
簡單的作法如下：  
mvc-config.xml:  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
...(略)  
<bean id="viewResolver" class="org.springframework.web.**servlet.view.InternalResourceViewResolver**">  
        <property name="**prefix**">  
                <value>**/WEB-INF/jsp/**</value>  
        </property>  
        <property name="**suffix**">  
                <value>**.jsp**</value>  
        </property>  
</bean>  
  
<bean name="/hello.do" class="com.aidc.controller.HelloController">  
        <property name="viewPage">  
                <value>**hello**</value>  
        </property>  
</bean>  
...(略)  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
上述會將Controller中的ModelAndView的viewName，對應到resolver所指定的前置字串(prefix)與後置字串(suffix)組合後的檔案程式。例如，此例中的viewName經由DI注入後，resolver會取得**hello**這個name，再和前置字串**/WEB-INF/jsp/**以及後置字串**.jsp**組合，變成"**/WEB-INF/jsp/hello.jsp**"這個完整的檔名，因此，經由resolver的處理後，可以知道回傳給容器的ModelAndView要交給哪一個View去接手。  
  
在resolver解析完之後，實際view的render以及model物件的處理是要交給有實作org.springframework.web.servlet.**View**介面的類別的實例來處理，我們稱之為viewClass。  
以**InternalResourceViewResolver**來說，他預設的viewClass是org.springframework.web.servlet.**InternalResourceView**。有可以設定成其他技術相關的viewClass，例如org.springframework.web.servlet.**JstlView**。  
  
  
  
2. **BeanNameViewResolver**  
再來介紹的是org.springframework.web.**servlet.view.BeanNameViewResolver**。這個方法很簡單，只要設定一個viewClass，並且指定其id，剛好就是ModelAndView的view名稱，並且設定其處理的view檔案為何就行。  
mvc-config.xml:  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
...(略)  
<bean id="viewResolver" class="org.springframework.web.**servlet.view.BeanNameViewResolver**">  
</bean>  
  
<bean id="**hello**" class="org.springframework.web.**servlet.view.JstlView**">  
        <property name="url">  
                <value>**/WEB-INF/jsp/hello.jsp**</value>  
        </property>  
</bean>  
...(略)  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
  
3. **XmlViewResolver**  
再來介紹我最常用的一個resolver，org.springframework.web.**servlet.view.XmlViewResolver**，他跟**BeanNameViewResolver**幾乎一樣，只不過他可以將所對應的view處理程式對應寫到另一個xml做管理。但是說穿了，這也沒什　特別的，因為在一開始web.xml定義contextConfigLocation時，本來就可以定義多個bean定義檔，只是用此作法加強了兩者之間的關聯性而已。  
  
mvc-config.xml:  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
...(略)  
<bean id="viewResolver" class="org.springframework.web.**servlet.view.XmlViewResolver**">  
        <property name="location">  
                <value>**/WEB-INF/views-mapping.xml**</value>  
        </property>  
</bean>  
...(略)  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
**views-mapping.xml:**  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
...(略)  
<bean id="**hello**" class="org.springframework.web.**servlet.view.JstlView**">  
        <property name="url">  
                <value>**/WEB-INF/jsp/hello.jsp**</value>  
        </property>  
</bean>  
...(略)  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
  
  
4. **SimpleMappingExceptionResolver**  
另外，介紹一個特別的resolver，**Exception Resolver**，可以針對不同的例外，導向到不同的錯誤訊息頁面。可以直接使用Spring提供的一個類別org.springframework.web.servlet.handler.**SimpleMappingExceptionResolver**來達成。  
  
mvc-config.xml:  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
...(略)  
<bean id="viewResolver" class="org.springframework.web.**servlet.view.InternalResourceViewResolver**">  
        <property name="**prefix**">  
                <value>**/WEB-INF/jsp/**</value>  
        </property>  
        <property name="**suffix**">  
                <value>**.jsp**</value>  
        </property>  
</bean>  
  
<bean id="exceptionResolver" class="org.springframework.web.servlet.handler.**SimpleMappingExceptionResolver**">  
        <property name="exceptionMappings">  
                <props>  
                        <prop key="**java.sql.SQLException**">**sqlexception**</prop>  
                        <prop key="**java.sql.IOException**">**ioexception**</prop>  
                </props>  
        </property>  
</bean>  
...(略)  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
在上面prop中的value，InternalResourceViewResol會將之處理成為**/WEB-INF/jsp/sqlexception.jsp**。因此，只要產生的例外是在設定之中的，就可以將之導向我們事先設定好的頁面去，而不需要讓使用者看到程式錯誤的error message。

關於Tiles組件

精通Struts：Tiles框架

2009-02-11 10:27

|  |
| --- |
| Tiles框架特性和内容  Tiles框架为创建Web页面提供了 一种模板机制，它能将网页的布局和内容分离。它允许先创建模板，然后在运行时动态地将内容插入到模板中。Tiles 框架建立在JSP的include指令的基础上，但它提供了比JSP的 include指令更强大的功能。Tiles框架具有如下特性：     ·创建可重用的模板     ·动态构建和装载页面     ·定义可重用的Tiles组件     ·支持国际化      Tiles框架包含以下内容：     ·Tiles标签库     ·Tiles组件的配置文件     ·TilesPlugIn插件      在开发Web站点时，常常要求同一站点的所有Web页面保持一致的外观，比如有相同的布局、页头、页尾和菜单。  采用基本的JSP语句创建复合式网页      创建动态Web页面的最基本的办法是为每个页面创建独立的JSP文件。如果网页的相同部分发生需求变更，必须手工修改所有的JSP文件。可见，采用基本的JSP语句来编写上述网页，会导致JSP代码的大量冗余，增加开发与维护成本。      采用JSP的include指令创建复合式网页      为了减少代码的冗余，可以把index.jsp和product.jsp中相同部分放在单独的JSP文件中，然后在index.jsp和 product.jsp文件中通过JSP include指令把其他JSP文件包含进来。这样提高了代码的可重用性。但是JSP include指令不能完全避免代码冗余，尽管这种方案减少了重复代码，但JSP文件的数量增加了，由原来的2个文件增加到7个文件，所以软件的复杂度也 增加了。      采用Tiles:Insert标签创建复合式网页      Tiles标签库的tiles:insert标签和JSP include指令具有相同的功能，也能把其他的JSP页面插入到当前页面中。用tiles:insert标签取代JSP include指令来创建复合式页面，代码仅有稍微的差别，两者的利弊也很相似。单纯使用tiles:insert标签来创建复合式页面，还没有充分发挥 Tiles框架的优势。 以下两条语句的作用是相同的： <jsp:include page="indexContent.jsp"/> <tiles:insert page="indexContent.jsp"/>      采用Tiles模板创建复合式网页      尽管使用了tiles:insert标签，index.jsp和product.jsp文件还是存在很多的重复代码。为了提高Web页面的可重用性和可维 护性，可以引入Tiles的模板机制。通俗的讲，Tiles模板是一种描述页面布局的JSP页面。Tiles模板仅仅定义Web页面的样式，而不指定内 容。在Web应用运行时，才把特定内容插入到模板页面中。同一模板可以被多个Web页面共用。使用模板，可以轻松的实现Web应用的所有页面保持相同的外 观和布局，无需为每个页面硬编码。在一个应用中，大多数页面使用同一模板，某些页面可能需要不同的外观，使用其他的模板，因此一个应用可能有一个以上模 板。 <%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles"%> <tiles:insert attribute="sidebar"/> <tiles:insert attribute="header"/> <tiles:insert attribute="content"/> <tiles:insert attribute="footer"/>  <%@ page contentType="text/html; charset=UTF-8" %> <%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles" %> <tiles:insert page="layout.jsp" flush="true">        <tiles:put name="sidebar" value="sidebar.jsp"/>       <tiles:put name="header" value="header.jsp"/>        <tiles:put name="content" value="indexContent.jsp"/>        <tiles:put name="footer" value="footer.jsp"/>  </tiles:insert>  采 用Tiles模板机制，大大提高了代码的可重用性和可维护性，模板中包含了网页共同的布局。如果布局发生变化，只需要修改模板文件，无需修改具体的网页文 件。不过，从例程16-13和16-14可以看出，尽管 index.jsp和product.jsp文件的长度都缩短了，但是两者还是存在重复代码。      Tiles组件的基本使用方法      为了最大程度的提高代码的可重用性和灵活性，Tiles框架引入了Tiles组件的概念。Tiles组件可以代表一个完整的网页，也可以代表网页的一部分。简单的Tiles组件可以组合成复杂的Tiles组件，或被扩展为复杂的Tiles组件。     Tiles框架允许在专门的XML文件中配置Tiles组件。例如，以下代码定义了一个名为"index-definition"的Tiles组件，它描述整个index.jsp网页： <tiles-definitions>    <definition name="index-definition" path="/layout.jsp">       <put name="sidebar" value="sidebar.jsp"/>       <put name="header" value="header.jsp"/>        <put name="content" value="indexContent.jsp"/>        <put name="footer" value="footer.jsp"/>     </definition> </tiles-definitions>      definition元素的name属性指定Tiles组件的名字，path属性指定Tiles组件使用的模板，definition元素的put子元素用于向模板中插入具体的网页内容。  例程16-15 tiles-defs.xml  <?xml version="1.0" encoding="ISO-8859-1" ?> <!DOCTYPE tiles-definitions PUBLIC "-//Apache Software Foundation//DTD Tiles Configuration 1.1//EN" "<http://jakarta.apache.org/struts/dtds/tiles-config_1_1.dtd>">  <tiles-definitions>    <definition name="index-definition" path="/layout.jsp">       <put name="sidebar" value="sidebar.jsp"/>       <put name="header" value="header.jsp"/>        <put name="content" value="indexContent.jsp"/>        <put name="footer" value="footer.jsp"/>     </definition>     <definition name="product-definition" path="/layout.jsp">       <put name="sidebar" value="sidebar.jsp"/>       <put name="header" value="header.jsp"/>        <put name="content" value="productContent.jsp"/>        <put name="footer" value="footer.jsp"/>     </definition>  </tiles-definitions>      以上代码定义了两个Tiles组件，它们分别代表完整的index.jsp和product.jsp页面。      （4）在Strut配置文件中配置TilesPlugin插件，代码如下：  <plug-in className="org.apache.struts.tiles.TilesPlugin" > <set-property property="definitions-config" value="/WEB-INF/tiles-defs.xml" /> <set-property property="definitions-parser-validate" value="true" /> </plug-in>     TilesPlugin插件用于加载Tiles组件的配置文件。在plug-in元素中包含几个set-property子元素，用于向TilesPlugin插件传入附加的参数：     ·definitions-config参数：指定Tiles组件的配置文件，如果有多个配置文件，则它们之间用逗号分隔。     ·definitions-parser-validate参数：指定XML解析器是否验证Tiles配置文件，可选值包括true和false，默认值为true。      （5）在web.xml文件中配置ActionServlet     为了保证在Web应用启动时加载TilesPlugin插件，应该加入ActionServlet控制器，ActionServlet控制器在初始化时能加载所有的插件。以下是在web.xml文件中配置ActionServlet的代码：  <servlet>     <servlet-name>action</servlet-name>     <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>     <init-param>       <param-name>config</param-name>       <param-value>/WEB-INF/struts-config.xml</param-value>     </init-param>     <load-on-startup>3</load-on-startup> </servlet>  <servlet-mapping>     <servlet-name>action</servlet-name>     <url-pattern>\*.do</url-pattern> </servlet-mapping>      （6）在index.jsp和product.jsp中插入Tiles组件，参见例程16-16和例程16-17：     例程16-16 index.jsp  <%@ page contentType="text/html; charset=UTF-8" %> <%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles" %> <tiles:insert definition="index-definition"/> &nbsp;&nbsp;&nbsp;&nbsp;例程16-17 product.jsp <ccid\_nobr> <table width="400" border="1" cellspacing="0" cellpadding="2" bordercolorlight = "black" bordercolordark = "#FFFFFF" align="center"> <tr>     <td bgcolor="e6e6e6" class="code" style="font-size:9pt">     <pre><ccid\_code> <%@ page contentType="text/html; charset=UTF-8" %> <%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles" %> <tiles:insert definition="product-definition"/>      通过Struts Action来调用Tiles组件      如果Tiles组件代表完整的网页，可以直接通过Struts Action来调用Tiles组件。例如，如果希望通过Struts Action来调用16.5.1节定义的名为"index-definition"的Tiles组件，可以在Struts配置文件中配置如下Action 映射:  <action-mappings> <action   path="/index"           type="org.apache.struts.actions.ForwardAction"           parameter="index-definition"> </action> </action-mappings>      接下来通过浏览器访问 <http://localhost:8080/tilestaglibs/index.do>，该请求先被转发到ForwardAction， ForwardAction再把请求转发给名为"index-definition"的Tiles组件，最后在浏览器端，用户将看到和index.jsp 相同的页面。      通过Struts Action来调用Tiles组件，可以充分发挥Struts框架负责流程控制的功能。此外，可以减少JSP文件的数目。例如，如果直接通过Struts Action来调用名为"index-definition"的Tiles组件，就不必再创建index.jsp文件。      解析Tiles组件的组合与扩展      Tiles 组件是一种可重用的组件。可以象搭积木一样，把简单的Tiles组件组装成复杂的Tiles组件，例如，可以把名为"index-definition" 的Tiles组件的左边部分拆分为独立的Tiles组件，名为"sidebar-definition"。  <definition name="index-definition" path="/layout.jsp"> <put name="sidebar" value="sidebar-definition" type="definition"/>         …… </definition>      以上put子元素的value属性指定被包含的Tiles组件的名字，type属性设为"definition"，表示value属性指定的是Tiles组件，而不是JSP文件。  <definition name="index-definition" extends="base-definition"> |

# Spring结合Quartz实现多任务定时调用

2009年03月30日 星期一 下午 07:49

|  |
| --- |
| Quartz框架提供了丰富的任务调度支持，比如，在何时执行何种任务，它是一个开源的由OpenSymphony维护的项目，开发者能够在Java EE，或单独的Java SE应用中使用它。无论是简单的任务调度，还是复杂的企业级应用，Quartz都能够很好地胜任。其中，这些任务可以是普通POJO，甚至还可以是EJB 3.0组件。  如果开发者需要开发如下方面的应用，则Quartz是理想的选择。  ◆驱动工作流：比如，如果新创建的流程任务需要在2小时内处理完，则在2小时后Quartz会检查订单是否成功处理。如果没有处理，则Quartz会依据工作流定义的规则来对订单进行处理，销毁它，或者进行其他处理。  ◆系统维护工作：比如，在每个工作日的固定时间将RDBMS中的内容导出为XML文件。         Spring 2.0提供了org.springframework.scheduling.quartz包，以支持Quartz任务调度集成。为提供用户的任务，开发者实现的类必须继承于quartz.QuartzJobBean抽象类。QuartzJobBean是Quartz中org.quartz.Job接口的简单实现（子类），Spring为简化Job接口的实现而提供了QuartzJobBean类。QuartzJobBean同Java 2 SDK中的TimerTask类似，用于定义任务本身。其中，executeInternal()方法定义待执行的任务，这同TimerTask中的run()类似。         为了指定更复杂的任务调度规则，开发者要使用Spring提供的CronTriggerBean，它是Quartz中org.quartz.CronTrigger的子类，Spring 2.0为简化CronTrigger子类的开发而提供了CronTriggerBean。CronTriggerBean比SimpleTriggerBean功能更强大，它能够控制任务执行的精确时间，比如，早上九点半需要执行某QuartzJobBean中给定的任务。借助于CronTriggerBean中的cronExpression属性能够设定任务的执行时机。         Quartz是一个强大的企业级任务调度框架，Spring中继承并简化了Quartz，下面就看看在Spring中怎样配置Quartz：首先我们来写二个被调度的类QuartzJob，及QuartzJobTwo，二个以上类似：  package com.writchie.quartz;  public class QuartzJob  {       public void work()       {               System.out.println("吴理琪提示您：Spring Quartz的任务调度1被调用！");               //定时调用的业务逻辑       }  }   package com.writchie.quartz;  public class QuartzJobTwo  {       public void work()       {             System.out.println("吴理琪提示您：Spring Quartz的任务调度2被调用 ！");             //定时调用的业务逻辑       }  }   Spring的配置文件applicationContext.xml：  <?xml version="1.0" encoding="UTF-8"?>  <beans  xmlns="<http://www.springframework.org/schema/beans>"  xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"  xsi:schemaLocation="<http://www.springframework.org/schema/beans> <http://www.springframework.org/schema/beans/spring-beans-2.5.xsd>">           <!-- 要调用的工作类 -->           <bean id="quartzJob" class="com.writchie.quartz.QuartzJob"></bean>           <bean id="quartzJobTwo" class="com.writchie.quartz.QuartzJobTwo"></bean>           <!-- 可继续加新的任务   -->           <!-- 要调用的工作类结束 -->            <!-- 定义调用对象和调用对象的方法 -->           <bean id="jobtask" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">               <!-- 调用的类 -->               <property name="targetObject">                   <ref bean="quartzJob"/>               </property>               <!-- 调用类中的方法 -->               <property name="targetMethod">                   <value>work</value>               </property>           </bean>            <bean id="jobtask2" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">               <!-- 调用的类 -->               <property name="targetObject">                   <ref bean="quartzJobTwo"/>               </property>               <!-- 调用类中的方法 -->               <property name="targetMethod">                   <value>work</value>               </property>           </bean>           <!-- 可继续加新的   -->         <!-- 定义调用对象和调用对象的方法结束 -->            <!-- 定义触发时间 -->           <bean id="doTime" class="org.springframework.scheduling.quartz.CronTriggerBean">               <property name="jobDetail">                   <ref bean="jobtask"/>               </property>               <!-- cron表达式 此处定义为一直触发执行任务 -->               <property name="cronExpression">                   <value>10,15,20,25,30,35,40,45,50,55 \* \* \* \* ?</value>               </property>           </bean>                     <bean id="doTime2" class="org.springframework.scheduling.quartz.CronTriggerBean">               <property name="jobDetail">                   <ref bean="jobtask2"/>               </property>               <!-- cron表达式   此处定义周一至周日的下午13:15触发-->               <property name="cronExpression">                   <value>0 15 13 ? \* SUN-SAT</value>               </property>           </bean>           <!-- 可继续加新的   -->           <!-- 定义触发时间结束 -->            <!-- 总管理类 如果将lazy-init='false'那么容器启动就会执行调度程序   -->           <bean id="startQuertz" lazy-init="false" autowire="no" class="org.springframework.scheduling.quartz.SchedulerFactoryBean">               <property name="triggers">                   <list>                       <ref bean="doTime"/>                       <ref bean="doTime2"/>                     <!-- 可继续加新的   -->                    </list>               </property>           </bean>         <!-- 总管理类结束   -->  </beans>   测试程序：  package com.writchie.quartz;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class QuartzService {  public static void main(String[] args) {     System.out.println("测试任务调度开始...");     ApplicationContext context = new ClassPathXmlApplicationContext(       "applicationContext.xml");     // 如果配置文件中将startQuertz bean的lazy-init设置为false 则不用实例化     // context.getBean("startQuertz");     System.out.print("测试任务调度结束!\n");  }  }   测试结果：  测试任务调度开始...  - Refreshing [org.springframework.context.support.ClassPathXmlApplicationContext@29428e](mailto:org.springframework.context.support.ClassPathXmlApplicationContext@29428e): display name [org.springframework.context.support.ClassPathXmlApplicationContext@29428e]; startup date [Sun Mar 01 13:13:21 CST 2009]; root of context hierarchy  - Loading XML bean definitions from class path resource [applicationContext.xml]  - Bean factory for application context [org.springframework.context.support.ClassPathXmlApplicationContext@29428e]: [org.springframework.beans.factory.support.DefaultListableBeanFactory@9be79a](mailto:org.springframework.beans.factory.support.DefaultListableBeanFactory@9be79a)  - Pre-instantiating singletons in [org.springframework.beans.factory.support.DefaultListableBeanFactory@9be79a](mailto:org.springframework.beans.factory.support.DefaultListableBeanFactory@9be79a): defining beans [quartzJob,quartzJobTwo,jobtask,jobtask2,doTime,doTime2,startQuertz]; root of factory hierarchy  - Job execution threads will use class loader of thread: main  - Quartz Scheduler v.1.5.2 created.  - RAMJobStore initialized.  - Quartz scheduler 'DefaultQuartzScheduler' initialized from default resource file in Quartz package: 'quartz.properties'  - Quartz scheduler version: 1.5.2  - Starting Quartz scheduler now  - Scheduler DefaultQuartzScheduler\_$\_NON\_CLUSTERED started.  测试任务调度结束!  吴理琪提示您：Spring Quartz的任务调度1被调用！  吴理琪提示您：Spring Quartz的任务调度1被调用！  吴理琪提示您：Spring Quartz的任务调度1被调用！  吴理琪提示您：Spring Quartz的任务调度1被调用！  吴理琪提示您：Spring Quartz的任务调度1被调用！  吴理琪提示您：Spring Quartz的任务调度1被调用！  吴理琪提示您：Spring Quartz的任务调度1被调用！  吴理琪提示您：Spring Quartz的任务调度1被调用！  吴理琪提示您：Spring Quartz的任务调度1被调用！  吴理琪提示您：Spring Quartz的任务调度1被调用！  吴理琪提示您：Spring Quartz的任务调度1被调用！  吴理琪提示您：Spring Quartz的任务调度1被调用！  吴理琪提示您：Spring Quartz的任务调度1被调用！  吴理琪提示您：Spring Quartz的任务调度1被调用！  吴理琪提示您：Spring Quartz的任务调度1被调用！  吴理琪提示您：Spring Quartz的任务调度1被调用！  吴理琪提示您：Spring Quartz的任务调度1被调用！  吴理琪提示您：Spring Quartz的任务调度2被调用 ！  吴理琪提示您：Spring Quartz的任务调度1被调用！   关于cron表达式（参考资料）： Cron 表达式包括以下 7 个字段：   * 秒 * 分 * 小时 * 月内日期 * 月 * 周内日期 * 年（可选字段）   特殊字符 Cron 触发器利用一系列特殊字符，如下所示：   * 反斜线（/）字符表示增量值。例如，在秒字段中“5/15”代表从第 5 秒开始，每 15 秒一次。 * 问号（?）字符和字母 L 字符只有在月内日期和周内日期字段中可用。问号表示这个字段不包含具体值。所以，如果指定月内日期，可以在周内日期字段中插入“?”，表示周内日期值无关紧要。字母 L 字符是 *last* 的缩写。放在月内日期字段中，表示安排在当月最后一天执行。在周内日期字段中，如果“L”单独存在，就等于“7”，否则代表当月内周内日期的最后一个实例。所以“0L”表示安排在当月的最后一个星期日执行。 * 在月内日期字段中的字母（W）字符把执行安排在最靠近指定值的工作日。把“1W”放在月内日期字段中，表示把执行安排在当月的第一个工作日内。 * 井号（#）字符为给定月份指定具体的工作日实例。把“MON#2”放在周内日期字段中，表示把任务安排在当月的第二个星期一。 * 星号（\*）字符是通配字符，表示该字段可以接受任何可能的值。   字段 允许值 允许的特殊字符  秒 0-59 , - \* /  分 0-59 , - \* /  小时 0-23 , - \* /  日期 1-31 , - \* ? / L W C  月份 1-12 或者 JAN-DEC , - \* /  星期 1-7 或者 SUN-SAT , - \* ? / L C #  年（可选） 留空, 1970-2099 , - \* / 表达式意义  "0 0 12 \* \* ?" 每天中午12点触发  "0 15 10 ? \* \*" 每天上午10:15触发  "0 15 10 \* \* ?" 每天上午10:15触发  "0 15 10 \* \* ? \*" 每天上午10:15触发  "0 15 10 \* \* ? 2005" 2005年的每天上午10:15触发  "0 \* 14 \* \* ?" 在每天下午2点到下午2:59期间的每1分钟触发  "0 0/5 14 \* \* ?" 在每天下午2点到下午2:55期间的每5分钟触发  "0 0/5 14,18 \* \* ?" 在每天下午2点到2:55期间和下午6点到6:55期间的每5分钟触发  "0 0-5 14 \* \* ?" 在每天下午2点到下午2:05期间的每1分钟触发  "0 10,44 14 ? 3 WED" 每年三月的星期三的下午2:10和2:44触发  "0 15 10 ? \* MON-FRI" 周一至周五的上午10:15触发  "0 15 10 15 \* ?" 每月15日上午10:15触发  "0 15 10 L \* ?" 每月最后一日的上午10:15触发  "0 15 10 ? \* 6L" 每月的最后一个星期五上午10:15触发  "0 15 10 ? \* 6L 2002-2005" 2002年至2005年的每月的最后一个星期五上午10:15触发  "0 15 10 ? \* 6#3" 每月的第三个星期五上午10:15触发  每天早上6点  0 6 \* \* \*  每两个小时  0 \*/2 \* \* \*  晚上11点到早上8点之间每两个小时，早上八点  0 23-7/2，8 \* \* \*  每个月的4号和每个礼拜的礼拜一到礼拜三的早上11点  0 11 4 \* 1-3  1月1日早上4点  0 4 1 1 \* |