**Web框架介绍---SpringMVC**

作者：   来源：http://blog.ccidnet.com   发布者：admin   
时间：2009-04-20 11:44:53   点击：625

12.1. Web框架介绍  
  
Spring 的web框架是围绕分发器（DispatcherServlet）设计的，DispatcherServlet将请求分发到不同的处理器，框架还包括可配置的处理器映射，视图解析，本地化，主题解析，还支持文件上传。缺省的处理器是一个简单的控制器（Controller）接口，这个接口仅仅定义了ModelAndView handleRequest(request,response)方法。你可以实现这个接口生成应用的控制器，但是使用Spring提供的一系列控制器实现会更好一些，比如AbstractController，AbstractCommandController，和SimpleFormController。应用控制器一般都从它们继承。注意你需要选择正确的基类：如果你没有表单，你就不需要一个FormController。这是和Structs的一个主要区别。  
  
你可以使用任何对象作为命令对象或表单对象：不必实现某个接口或从某个基类继承。Spring的数据绑定相当灵活，例如，它认为类型不匹配这样的错误应该是应用级的验证错误，而不是系统错误。所以你不需要为了处理无效的表单提交，或者正确地转换字符串，在你的表单对象中用字符串类型重复定义你的业务对象属性。你应该直接绑定表单到业务对象上。这是和Struts的另一个重要不同，Struts是围绕象Action和ActionForm这样的基类构建的，每一种行为都是它们的子类。  
  
和WebWork 相比，Spring将对象细分成不同的角色：它支持的概念有控制器（Controller），可选的命令对象（Command Object）或表单对象（Form Object），以及传递到视图的模型（Model）。模型不仅包含命令对象或表单对象，而且也包含任何引用数据。但是，WebWork的Action将所有的这些角色都合并在一个单独的对象里。WebWork允许你在表单中使用现有的业务对象，但是只能把它们定义成不同Action类的bean属性。更重要的是，在运算和表单赋值时，使用的是同一个处理请求的Action实例。因此，引用数据也需要被定义成Action的bean属性。这样在一个对象就承担了太多的角色。  
  
对于视图：Spring的视图解析相当灵活。一个控制器实现甚至可以直接输出一个视图作为响应，这需要使用null返回 ModelAndView。在一般的情况下，一个ModelAndView实例包含视图名字和模型映射表，模型映射表提供了bean的名字及其对象（比如命令对象或表单对象，引用数据等等）的对应关系。视图名解析的配置是非常灵活的，可以通过bean的名字，属性文件或者你自己的ViewResolver 来实现。抽象的模型映射表完全抽象了表现层，没有任何限制：JSP，Velocity，或者其它的技术——任何表现层都可以直接和Spring集成。模型映射表仅仅将数据转换成合适的格式，比如JSP请求属性或者Velocity模版模型。  
12.1.1. MVC实现的可扩展性  
  
许多团队努力争取在技术和工具方面能使他们的投入更有价值，无论是现有的项目还是新的项目都是这样。具体地说，Struts 不仅有大量的书籍和工具，而且有许多开发者熟悉它。因此，如果你能忍受Struts的架构性缺陷，它仍然是web层一个很好的选择。WebWork和其它 web框架也是这样。  
  
如果你不想使用Spring的web MVC框架，而仅仅想使用Spring提供的其它功能，你可以很容易地将你选择的web框架和Spring结合起来。只要通过Spring的 ContextLoadListener启动一个Spring的根应用上下文，并且通过它的ServletContext属性（或者Spring的各种帮助方法）在Struts或WebWork的Action中访问。注意到现在没有提到任何具体的“plugins”，因此这里也没有提及如何集成：从web 层的角度看，你可以仅仅把Spring作为一个库使用，根应用上下文实例作为入口。  
  
所有你注册的bean和Spring的服务可以在没有 Spring的web MVC下被访问。Spring并没有在使用方法上和Struts或WebWork竞争，它只是提供单一web框架所没有的功能，从bean的配置到数据访问和事务处理。所以你可以使用Spring的中间层和（或者）数据访问层来增强你的应用，即使你只是使用象JDBC或Hibernate事务抽象这样的功能。  
12.1.2. Spring MVC框架的特点  
  
如果仅仅关注于web方面的支持，Spring有下面一些特点：  
  
  \*  
  
    清晰的角色划分：控制器，验证器，命令对象，表单对象和模型对象；分发器，处理器映射和视图解析器；等等。  
  \*  
  
    直接将框架类和应用类都作为JavaBean配置，包括通过应用上下文配置中间层引用，例如，从web控制器到业务对象和验证器的引用。  
  \*  
  
    可适应性，但不具有强制性：根据不同的情况，使用任何你需要的控制器子类（普通控制器，命令，表单，向导，多个行为，或者自定义的），而不是要求任何东西都要从Action/ActionForm继承。  
  \*  
  
    可重用的业务代码，而不需要代码重复：你可以使用现有的业务对象作为命令对象或表单对象，而不需要在ActionForm的子类中重复它们的定义。  
  \*  
  
    可定制的绑定和验证：将类型不匹配作为应用级的验证错误，这可以保存错误的值，以及本地化的日期和数字绑定等，而不是只能使用字符串表单对象，手动解析它并转换到业务对象。  
  \*  
  
    可定制的处理器映射，可定制的视图解析：灵活的模型可以根据名字/值映射，处理器映射和视图解析使应用策略从简单过渡到复杂，而不是只有一种单一的方法。  
  \*  
  
    可定制的本地化和主题解析，支持JSP，无论有没有使用Spring标签库，支持JSTL，支持不需要额外过渡的Velocity，等等。  
  \*  
  
    简单而强大的标签库，它尽可能地避免在HTML生成时的开销，提供在标记方面的最大灵活性。  
  
12.2. 分发器（DispatcherServlet）  
  
Spring的web框架——象其它web框架一样——是一个请求驱动的web框架，其设计围绕一个能将请求分发到控制器的servlet，它也提供其它功能帮助web应用开发。然而，Spring的DispatcherServlet所做的不仅仅是这些。它和Spring的ApplicationContext完全集成在一起，允许你使用Spring的其它功能。  
  
DispatcherServlet和其它servlet一样定义在你的web应用的web.xml文件里。DispatcherServlet处理的请求必须在同一个web.xml文件里使用url-mapping定义映射。  
  
<web-app>  
  ...  
  <servlet>  
    <servlet-name>example</servlet-name>  
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
    <load-on-startup>1</load-on-startup>  
  </servlet>  
  <servlet-mapping>  
    <servlet-name>example</servlet-name>  
    <url-pattern>\*.form</url-pattern>  
  </servlet-mapping>     
</web-app>  
  
在上面的例子里，所有以.form结尾的请求都会由DispatcherServlet处理。接下来需要配置DispatcherServlet本身。正如在第 3.10 节 “介绍ApplicationContext”中所描述的，Spring中的ApplicationContexts可以被限制在不同的作用域。在web框架中，每个DispatcherServlet有它自己的WebApplicationContext，它包含了DispatcherServlet配置所需要的bean。DispatcherServlet 使用的缺省BeanFactory是XmlBeanFactory，并且DispatcherServlet在初始化时会在你的web应用的WEB-INF目录下寻找[servlet-name]-servlet.xml文件。DispatcherServlet使用的缺省值可以使用servlet初始化参数修改（详细信息如下）。  
  
WebApplicationContext仅仅是一个拥有web应用必要功能的普通ApplicationContext。它和一个标准的ApplicationContext的不同之处在于它能够解析主题（参考第 12.7 节 “主题使用”），并且它知道和那个servlet关联（通过到ServletContext的连接）。WebApplicationContext被绑定在ServletContext上，当你需要的时候，可以使用RequestContextUtils找到WebApplicationContext。  
  
Spring 的DispatcherServlet有一组特殊的bean，用来处理请求和显示相应的视图。这些bean包含在Spring的框架里，（可选择）可以在 WebApplicationContext中配置，配置方式就象配置其它bean的方式一样。这些bean中的每一个都在下面被详细描述。待一会儿，我们就会提到它们，但这里仅仅是让你知道它们的存在以便我们继续讨论DispatcherServlet。对大多数bean，都提供了缺省值，所有你不必要担心它们的值。  
  
表 12.1. WebApplicationContext中特殊的bean  
名称     解释  
处理器映射（handler mapping(s)）     (第 12.4 节 “处理器映射”) 前处理器，后处理器和控制器的列表，它们在符合某种条件下才被执行（例如符合控制器指定的URL）  
控制器（controller(s)）     (第 12.3 节 “控制器”) 作为MVC三层一部分，提供具体功能（或者至少能够访问具体功能）的bean  
视图解析器（view resolver）     (第 12.5 节 “视图与视图解析”) 能够解析视图名，在DispatcherServlet解析视图时使用  
本地化信息解析器（locale resolver）     (第 12.6 节 “使用本地化信息”) 能够解析用户正在使用的本地化信息，以提供国际化视图  
主题解析器（theme resolver）     (第 12.7 节 “主题使用”) 能够解析你的web应用所使用的主题,比如,提供个性化的布局  
multipart解析器     (第 12.8 节 “Spring对multipart（文件上传）的支持”) 提供HTML表单文件上传功能  
处理器异常解析器（handlerexception resolver）     (第 12.9 节 “处理异常”) 将异常对应到视图,或者实现某种复杂的异常处理代码  
  
当DispatcherServlet被安装配置好，DispatcherServlet一接收到请求，处理就开始了。下面的列表描述了DispatcherServlet处理请求的全过程：  
  
  1.  
  
    搜索WebApplicationContext，并将它绑定到请求的一个属性上，以便控制器和处理链上的其它处理器能使用WebApplicationContext。缺省它被绑定在DispatcherServlet.WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE这个关键字上  
  2.  
  
    绑定本地化信息解析器到请求上，这样使得处理链上的处理器在处理请求（显示视图，准备数据等等）时能解析本地化信息。如果你不使用本地化信息解析器，它不会影响任何东西，忽略它就可以了  
  3.  
  
    绑定主题解析器到请求上，使得视图决定使用哪个主题（如果你不需要主题，可以忽略它，解析器仅仅是绑定，如果你不使用它，不会影响任何东西）  
  4.  
  
    如果multipart解析器被指定，请求会被检查是否使用了multipart，如果是，multipart解析器会被保存在MultipartHttpServletRequest中以便被处理链中的其它处理器使用（下面会讲到更多有关multipart处理的内容）  
  5.  
  
    搜索合适的处理器。如果找到，执行和这个处理器相关的执行链（预处理器，后处理器，控制器），以便准备模型数据  
  6.  
  
    如果模型数据被返还，就使用配置在WebApplicationContext中的视图解析器，显示视图，否则（可能是安全原因，预处理器或后处理器截取了请求），虽然请求能够提供必要的信息，但是视图也不会被显示。  
  
在请求处理过程中抛出的异常可以被任何定义在WebApplicationContext中的异常解析器所获取。使用这些异常解析器，你可以在异常抛出时定义特定行为。  
  
Spring的DispatcherServlet也支持返回Servlet API定义的last-modification-date，决定某个请求最后修改的日期很简单。DispatcherServlet会首先寻找一个合适的处理器映射，检查处理器是否实现了LastModified接口，如果是，将long getLastModified(request)的值返回给客户端。  
  
你可以在web.xml文件中添加上下文参数或servlet初始化参数定制Spring的DispatcherServlet。下面是一些可能的参数。  
  
表 12.2. DispatcherServlet初始化参数  
参数     解释  
contextClass     实现WebApplicationContext的类，当前的servlet用它来实例化上下文。如果这个参数没有指定，使用XmlWebApplicationContext  
contextConfigLocation     传给上下文实例（由contextClass指定）的字符串，用来指定上下文的位置。这个字符串可以被分成多个字符串（使用逗号作为分隔符）来支持多个上下文（在多上下文的情况下，被定义两次的bean中，后面一个优先）  
namespace     WebApplicationContext命名空间。缺省是[server-name]-servlet  
12.3. 控制器  
  
控制器的概念是MVC设计模式的一部分。控制器定义了应用的行为，至少能使用户访问到这些行为。控制器解释用户输入，并将其转换成合理的模型数据，从而可以进一步地由视图展示给用户。Spring以一种抽象的方式实现了控制器概念，这样使得不同类型的控制器可以被创建。Spring包含表单控制器，命令控制器，执行向导逻辑的控制器等等。  
  
Spring控制器架构的基础是org.springframework.mvc.Controller接口。  
  
public interface Controller {  
  
  /\*\*  
  \* Process the request and return a ModelAndView object which the DispatcherServlet  
  \* will render.  
  \*/  
  ModelAndView handleRequest(  
    HttpServletRequest request,   
    HttpServletResponse response)   
  throws Exception;  
}  
  
你可以发现Controller接口仅仅声明了一个方法，它能够处理请求并返回合适的模型和视图。Spring MVC实现的基础就是这三个概念：ModelAndView和Controller。 因为Controller接口是完全抽象的，Spring提供了许多已经包含一定功能的控制器。控制器接口仅仅定义了每个控制器提供的共同功能：处理请求并返回一个模型和一个视图。  
12.3.1. AbstractController 和 WebContentGenerator  
  
当然，就一个控制器接口并不够。为了提供一套基础设施，所有的Spring控制器都从 AbstractController 继承，AbstractController 提供缓存和其它比如 mimetype 的设置的功能。  
  
表 12.3. AbstractController提供的功能  
功能     解释  
supportedMethods     指定这个控制器应该接受什么样的请求方法。通常它被设置成GET和POST，但是你可以选择你想支持的方法。如果控制器不支持请求发送的方法，客户端会得到通知（ServletException）  
requiresSession     指定这个控制器是否需要会话。这个功能提供给所有控制器。如果控制器在没有会话的情况下接收到请求，用户被通知ServletException  
synchronizeSession     如果你需要使控制器同步访问用户会话，使用这个参数。具体地说，继承的控制器要重载handleRequestInternal方法，如果你指定了这个变量，控制器就被同步化。  
cacheSeconds     当你需要控制器在HTTP响应中生成缓存指令，用这参数指定一个大于零的整数。缺省它被设置为-1，所以就没有生成缓存指令  
useExpiresHeader     指定你的控制器使用HTTP 1.0兼容的"Expires"。缺省为true，所以你可以不用修改它  
useCacheHeader     指定你的控制器使用HTTP 1.0兼容的"Cache-Control"。缺省为true，所以你也可以不用修改它  
  
最后的两个属性是WebContentGenerator定义的，WebContentGenerator是AbstractController的超类……  
  
当使用AbstractController作为你的控制器基类时（一般不推荐这样做，因为有许多预定义的控制器你可以选择），你只需要重载handleRequestInternal(HttpServletRequest, HttpServletResponse)这个方法，实现你自己的逻辑，并返回一个ModelAndView对象。下面这个简单例子包含一个类和在web应用上下文中的定义。  
  
package samples;  
  
public class SampleController extends AbstractController {  
  
  public ModelAndView handleRequestInternal(  
    HttpServletRequest request,  
    HttpServletResponse response)  
  throws Exception {  
    ModelAndView mav = new ModelAndView("foo", new HashMap());  
  }  
}  
  
<bean id="sampleController" class="samples.SampleController">  
  <property name="cacheSeconds"><value>120</value</property>  
</bean>  
  
除了这个类和在web应用上下文中的定义，还需要设置处理器映射（参考第 12.4 节 “处理器映射”），这样这个简单的控制器就可以工作了。这个控制器将生成缓存指令告诉客户端缓存数据2分钟后再检查状态。这个控制器还返回了一个硬编码的视图名（不是很好）（详情参考第 12.5 节 “视图与视图解析”）。  
12.3.2. 其它的简单控制器  
  
除了AbstractController——虽然有许多其他控制器可以提供给你更多的功能，但是你还是可以直接继承AbstractController——有许多简单控制器，它们可以减轻开发简单MVC应用时的负担。ParameterizableViewController基本上和上面例子中的一样，但是你可以指定返回的视图名，视图名定义在web应用上下文中（不需要硬编码的视图名）  
  
FileNameViewController检查URL并获取文件请求的文件名（<http://www.springframework.org/index.html>的文件名是index），把它作为视图名。仅此而已。  
12.3.3. MultiActionController  
  
Spring提供一个多动作控制器，使用它你可以将几个动作合并在一个控制器里，这样可以把功能组合在一起。多动作控制器存在在一个单独的包中——org.springframework.web.mvc.multiaction——它能够将请求映射到方法名，然后调用正确的方法。比如当你在一个控制器中有很多公共的功能，但是想多个入口到控制器使用不同的行为，使用多动作控制器就特别方便。  
  
表 12.4. MultiActionController提供的功能  
功能     解释  
delegate     MultiActionController 有两种使用方式。第一种是继承MultiActionController，并在子类中指定由MethodNameResolver解析的方法（这种情况下不需要这个配置参数），第二种是你定义了一个代理对象，由它调用Resolver解析的方法。如果你是这种情况，你必须使用这个配置参数定义代理对象  
methodNameResolver     由于某种原因，MultiActionController需要基于收到的请求解析它必须调用的方法。你可以使用这个配置参数定义一个解析器  
  
一个多动作控制器的方法需要符合下列格式：  
  
// actionName can be replaced by any methodname  
ModelAndView actionName(HttpServletRequest, HttpServletResponse);  
  
由于MultiActionController不能判断方法重载（overloading），所以方法重载是不允许的。此外，你可以定义exception handlers，它能够处理从你指定的方法中抛出的异常。包含异常处理的动作方法需要返回一个ModelAndView对象，就象其它动作方法一样，并符合下面的格式：  
  
// anyMeaningfulName can be replaced by any methodname  
ModelAndView anyMeaningfulName(HttpServletRequest, HttpServletResponse, ExceptionClass);  
  
ExceptionClass可以是任何异常，只要它是java.lang.Exception或java.lang.RuntimeException的子类。  
  
MethodNameResolver根据收到的请求解析方法名。有三种解析器可以供你选择，当然你可以自己实现解析器。  
  
  \*  
  
    ParameterMethodNameResolver － 解析请求参数，并将它作为方法名（<http://www.sf.net/index.view?testParam=testIt>的请求就会调用testIt(HttpServletRequest,HttpServletResponse)）。使用paramName配置参数可以调整所检查的参数  
  \*  
  
    InternalPathMethodNameResolver － 从路径中获取文件名作为方法名（<http://www.sf.net/testing.view>的请求会调用testing(HttpServletRequest, HttpServletResponse)方法）  
  \*  
  
    PropertiesMethodNameResolver － 使用用户定义的属性对象将请求的URL映射到方法名。当属性定义/index/welcome.html=doIt，并且收到/index/welcome.html的请求，就调用doIt(HttpServletRequest, HttpServletResponse)方法。这个方法名解析器需要使用PathMatcher（参考 第 12.10.1 节 “关于pathmatcher的小故事”）所以如果属性包含/\*\*/welcom?.html，该方法也会被调用！  
  
我们来看一组例子。首先是一个使用ParameterMethodNameResolver和代理属性的例子，它接受包含参数名的请求，调用方法retrieveIndex:  
  
<bean id="paramResolver" class="org....mvc.multiaction.ParameterMethodNameResolver">  
  <property name="paramName"><value>method</value></property>  
</bean>  
  
<bean id="paramMultiController" class="org....mvc.multiaction.MultiActionController">  
  <property name="methodNameResolver"><ref bean="paramResolver"/></property>  
  <property name="delegate"><ref bean="sampleDelegate"/>  
</bean>  
  
<bean id="sampleDelegate" class="samples.SampleDelegate"/>  
  
## together with  
  
public class SampleDelegate {  
  
  public ModelAndView retrieveIndex(  
    HttpServletRequest req,   
    HttpServletResponse resp) {  
  
    rerurn new ModelAndView("index", "date", new Long(System.currentTimeMillis()));  
  }  
}  
  
当使用上面的代理对象时，我们也可以使用PropertiesMethodNameRseolver来匹配一组URL，将它们映射到我们定义的方法上：  
  
<bean id="propsResolver" class="org....mvc.multiaction.PropertiesMethodNameResolver">  
  <property name="mappings">  
    <props>  
        <prop key="/index/welcome.html">retrieveIndex</prop>  
        <prop key="/\*\*/notwelcome.html">retrieveIndex</prop>  
        <prop key="/\*/user?.html">retrieveIndex</prop>  
    </props>  
  </property>  
</bean>  
  
<bean id="paramMultiController" class="org....mvc.multiaction.MultiActionController">  
  <property name="methodNameResolver"><ref bean="propsResolver"/></property>  
  <property name="delegate"><ref bean="sampleDelegate"/>  
</bean>  
  
12.3.4. 命令控制器  
  
Spring的CommandControllers是Spring MVC包的重要部分。命令控制器提供了一种和数据对象交互的方式，并动态将来自HttpServletRequest的参数绑定到你指定的数据对象上。和Struts的actonform相比，在Spring中，你不需要实现任何接口来实现数据绑定。首先，让我们看一下有哪些可以使用的命令控制器，以便有一个清晰的了解：  
  
  \*  
  
    AbstractCommandController - 你可以使用这个命令控制器来创建你自己的命令控制器，它能够将请求参数绑定到你指定的数据对象。这个类并不提供任何表单功能，但是它提供验证功能，并且让你在控制器中定义如何处理包含请求参数的数据对象。  
  \*  
  
    AbstractFormController - 一个提供表单提交支持的控制器。使用这个控制器，你可以定义表单，并使用你从控制器获取的数据对象构建表单。当用户输入表单内容， AbstractFormController将用户输入的内容绑定到数据对象，验证这些内容，并将对象交给控制器，完成适当的动作。它所支持的功能有无效表单提交（再次提交），验证，和正确的表单工作流。你可以控制将什么视图绑定到你的AbstractFormController。如果你需要表单，但不想在应用上下文中指定显示给用户的视图，就使用这个控制器。  
  \*  
  
    SimpleFormController - 这是一个更具体的FormCotnroller，它能用相应的数据对象帮助你创建表单。SimpleFormController让你指定一个命令对象，表单视图名，当表单提交成功后显示给用户的视图名等等。  
  \*  
  
    WizardFormController - 最后一个也是功能最强的控制器。WizardFormController 允许你以向导风格处理数据对象，当使用大的数据对象时，这样的方式相当方便。  
  
12.4. 处理器映射  
  
使用处理器映射，你可以将web请求映射到正确的处理器上。有很多处理器映射你可以使用，例如：SimpleUrlHandlerMapping或者BeanNameUrlHandlerMapping，但是先让我们看一下HandlerMapping的基本概念。  
  
一个基本的HandlerMapping所提供的功能是将请求传递到HandlerExecutionChain上，首先HandlerExecutionChain包含一个符合输入请求的处理器。其次（但是可选的）是一个可以拦截请求的拦截器列表。当收到请求，DispatcherServlet将请求交给处理器映射，让它检查请求并获得一个正确的HandlerExecutionChain。然后，执行定义在执行链中的处理器和拦截器（如果有拦截器的话）  
  
包含拦截器（处理器执行前，执行后，或者执行前后）的可配置的处理器映射功能非常强大。许多功能被放置在自定义的HandlerMappings中。一个自定义的处理器映射不仅根据请求的URL，而且还可以根据和请求相关的会话状态来选择处理器。  
  
我们来看看Spring提供的处理器映射。  
12.4.1. BeanNameUrlHandlerMapping  
  
BeanNameUrlHandlerMapping是一个简单但很强大的处理器映射，它将收到的HTTP请求映射到在web应用上下文中定义的bean的名字上。如果我们想要使用户插入一个账户，并且假设我们提供了FormController（关于CommandController和FormController请参考第 12.3.4 节 “命令控制器”）和显示表单的JSP视图（或Velocity模版）。当使用BeanNameUrlHandlerMapping时，我们用下面的配置能将包含URL <http://samples.com/editaccount.form>的HTTP请求映射到合适的FormController上：  
  
<beans>  
  <bean id="handlerMapping"   
      class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>  
      
  <bean name="/editaccount.form"  
      class="org.springframework.web.servlet.mvc.SimpleFormController">  
    <property name="formView"><value>account</value></property>  
    <property name="successView"><value>account-created</value></property>  
    <property name="commandName"><value>Account</value></property>  
    <property name="commandClass"><value>samples.Account</value></property>  
  </bean>  
<beans>     
  
所有/editaccount.form的请求就会由上面的FormController处理。当然我们得在web.xml中定义servlet-mapping，接受所有以.form结尾的请求。  
  
<web-app>  
  ...  
  <servlet>  
    <servlet-name>sample</servlet-name>  
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
    <load-on-startup>1</load-on-startup>  
  </servlet>  
  
  <!-- Maps the sample dispatcher to /\*.form -->  
  <servlet-mapping>  
    <servlet-name>sample</servlet-name>  
    <url-pattern>\*.form</url-pattern>  
  </servlet-mapping>  
  ...  
</web-app>  
  
注意：如果你使用BeanNameUrlHandlerMapping，你不必在web应用上下文中定义它。缺省情况下，如果在上下文中没有找到处理器映射，DispatcherServlet会为你创建一个BeanNameUrlHandlerMapping！  
12.4.2. SimpleUrlHandlerMapping  
  
另一个——更强大的处理器映射——是SimpleUrlHandlerMapping。它在应用上下文中可以配置，并且有Ant风格的路径匹配功能（参考第 12.10.1 节 “关于pathmatcher的小故事”）。下面几个例子可以帮助理解：  
  
<web-app>  
  ...  
  <servlet>  
    <servlet-name>sample</servlet-name>  
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
    <load-on-startup>1</load-on-startup>  
  </servlet>  
  
  <!-- Maps the sample dispatcher to /\*.form -->  
  <servlet-mapping>  
    <servlet-name>sample</servlet-name>  
    <url-pattern>\*.form</url-pattern>  
  </servlet-mapping>  
  <servlet-mapping>  
    <servlet-name>sample</servlet-name>  
    <url-pattern>\*.html</url-pattern>  
  </servlet-mapping>  
  ...  
</web-app>  
  
允许所有以.html和.form结尾的请求都由这个示例dispatchservelt处理。  
  
<beans>  
  <bean id="handlerMapping"   
      class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">           
    <property name="mappings">  
        <props>  
          <prop key="/\*/account.form">editAccountFormController</prop>  
          <prop key="/\*/editaccount.form">editAccountFormController</prop>  
          <prop key="/ex/view\*.html">someViewController</prop>  
          <prop key="/\*\*/help.html">helpController</prop>  
        </props>  
    </property>  
  </bean>  
    
  <bean id="someViewController"  
      class="org.springframework.web.servlet.mvc.FilenameViewController"/>  
    
  <bean id="editAccountFormController"  
      class="org.springframework.web.servlet.mvc.SimpleFormController">  
    <property name="formView"><value>account</value></property>  
    <property name="successView"><value>account-created</value></property>  
    <property name="commandName"><value>Account</value></property>  
    <property name="commandClass"><value>samples.Account</value></property>  
  </bean>  
<beans>  
  
这个处理器映射首先将所有目录中文件名为help.html的请求传递给helpController（译注，原文为someViewController），someViewController是一个FilenameViewController（更多信息请参考第 12.3 节 “控制器”）。所有ex目录中资源名以view开始，.html结尾的请求都会被传递给控制器。这里定义了两个使用editAccountFormController的处理器映射。  
12.4.3. 添加HandlerInterceptors  
  
处理器映射提供了拦截器概念，当你想要为所有请求提供某种功能时，例如做某种检查，这就非常有用。  
  
处理器映射中的拦截器必须实现org.springframework.web.servlet包中的HandlerInterceptor接口。这个接口定义了三个方法，一个在处理器执行前被调用，一个在处理器执行后被调用，另一个在整个请求处理完后调用。这三个方法提供你足够的灵活度做任何处理前和处理后的操作。  
  
preHandle方法有一个boolean返回值。使用这个值，你可以调整执行链的行为。当返回true时，处理器执行链将继续执行，当返回false时，DispatcherServlet认为拦截器本身将处理请求（比如显示正确的视图），而不继续执行执行链中的其它拦截器和处理器。  
  
下面的例子提供了一个拦截器，它拦截所有请求，如果当前时间是在上午9点到下午6点，将重定向到某个页面。  
  
<beans>  
  <bean id="handlerMapping"   
      class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">           
    <property name="interceptors">  
        <list>  
          <ref bean="officeHoursInterceptor"/>  
        </list>  
    </property>  
    <property name="mappings">  
        <props>  
          <prop key="/\*.form">editAccountFormController</prop>  
          <prop key="/\*.view">editAccountFormController</prop>  
        </props>  
    </property>  
  </bean>  
    
  <bean id="officeHoursInterceptor"   
      class="samples.TimeBasedAccessInterceptor">  
    <property name="openingTime"><value>9</value></property>  
    <property name="closingTime"><value>18</value></property>  
  </bean>     
<beans>  
  
package samples;  
  
public class TimeBasedAccessInterceptor extends HandlerInterceptorAdapter {  
  
  private int openingTime;     
  private int closingTime;     
  public void setOpeningTime(int openingTime) {  
    this.openingTime = openingTime;  
  }     
  public void setClosingTime(int closingTime) {  
    this.closingTime = closingTime;  
  }     
  public boolean preHandle(  
        HttpServletRequest request,  
        HttpServletResponse response,  
        Object handler)  
  throws Exception {  
    Calendar cal = Calendar.getInstance();  
    int hour = cal.get(HOUR\_OF\_DAY);  
    if (openingTime <= hour < closingTime) {  
        return true;  
    } else {  
        response.sendRedirect("<http://host.com/outsideOfficeHours.html>");  
        return false;  
    }  
  }     
}  
  
任何收到的请求，都将被TimeBasedAccessInterceptor截获，如果当前时间不在上班时间，用户会被重定向到一个静态html页面，比如告诉他只能在上班时间才能访问网站。  
  
你可以发现，Spring提供了adapter，使你很容易地使用HandlerInterceptor。  
12.5. 视图与视图解析  
  
所有web应用的MVC框架都会有它们处理视图的方式。Spring提供了视图解析器，这使得你在浏览器显示模型数据时不需要指定具体的视图技术。Spring允许你使用Java Server Page，Velocity模版和XSLT视图。第 13 章 集成表现层详细说明了如何集成不同的视图技术。  
  
Spring处理视图的两个重要的类是ViewResolver和View。View接口为请求作准备，并将请求传递给某个视图技术。ViewResolver提供了一个视图名和实际视图之间的映射。  
12.5.1. ViewResolvers  
  
正如前面所讨论的，SpringWeb框架的所有控制器都返回一个ModelAndView实例。Spring中的视图由视图名识别，视图解析器解析。Spring提供了许多视图解析器。我们将列出其中的一些，和它们的例子。  
  
表 12.5. 视图解析器  
ViewResolver     描述  
AbstractCachingViewResolver     抽象视图解析器，负责缓存视图。许多视图需要在使用前作准备，从它继承的视图解析器可以缓存视图。  
ResourceBundleViewResolver     使用ResourceBundle中的bean定义实现ViewResolver，这个ResourceBundle由bundle的basename指定。这个bundle通常定义在一个位于classpath中的一个属性文件中  
UrlBasedViewResolver     这个ViewResolver实现允许将符号视图名直接解析到URL上，而不需要显式的映射定义。如果你的视图名直接符合视图资源的名字而不需要任意的映射，就可以使用这个解析器  
InternalResourceViewResolver     UrlBasedViewResolver 的子类，它很方便地支持InternalResourceView（也就是Servlet和JSP），以及JstlView和TilesView的子类。由这个解析器生成的视图的类都可以通过setViewClass指定。详细参考UrlBasedViewResolver的javadocs  
VelocityViewResolver     UrlBasedViewResolver的子类，它能方便地支持VelocityView（也就是Velocity模版）以及它的子类  
  
例如，当使用JSP时，可以使用UrlBasedViewResolver。这个视图解析器将视图名翻译成URL，并将请求传递给RequestDispatcher显示视图。  
  
<bean id="viewResolver"   
    class="org.springframework.web.servlet.view.UrlBasedViewResolver">  
  <property name="prefix"><value>/WEB-INF/jsp/</value></property>  
  <property name="suffix"><value>.jsp</value></property>  
</bean>  
  
当返回test作为视图名时，这个视图解析器将请求传递给RequestDispatcher，RequestDispatcher将请求再传递给/WEB-INF/jsp/test.jsp。  
  
当在一个web应用中混合使用不同的视图技术时，你可以使用ResourceBundleViewResolver：  
  
<bean id="viewResolver"  
    class="org.springframework.web.servlet.view.ResourceBundleViewResolver">  
  <property name="baseName"><value>views</value></property>  
  <property name="defaultParentView"><value>parentView</value></property  
</bean>  
  
12.6. 使用本地化信息  
  
Spring架构的绝大部分都支持国际化，就象Spring的web框架一样。SpringWeb框架允许你使用客户端本地化信息自动解析消息。这由LocaleResolver对象完成。  
  
当收到请求时，DispatcherServlet寻找一个本地化信息解析器，如果找到它就使用它设置本地化信息。使用RequestContext.getLocale()方法，你总可以获取本地化信息供本地化信息解析器使用。  
  
除了自动本地化信息解析，你还可以将一个拦截器放置到处理器映射上（参考第 12.4.3 节 “添加HandlerInterceptors”），以便在某种环境下，比如基于请求中的参数，改变本地化信息。  
  
本地化信息解析器和拦截器都定义在org.springframework.web.servlet.i18n包中，并且在你的应用上下文中配置。你可以选择使用Spring中的本地化信息解析器。  
12.6.1. AcceptHeaderLocaleResolver  
  
这个本地化信息解析器检查请求中客户端浏览器发送的accept-language头。通常这个头信息包含客户端操作系统的本地化信息。  
12.6.2. CookieLocaleResolver  
  
这个本地化信息解析器检查客户端中的cookie是否本地化信息被指定了。如果指定就使用该本地化信息。使用这个本地化信息解析器的属性，你可以指定cookie名，以及最大生存期。  
  
<bean id="localeResolver">  
  <property name="cookieName"><value>clientlanguage</value></property>  
  <!-- in seconds. If set to -1, the cookie is not persisted (deleted when browser shuts down) -->  
  <property name="cookieMaxAge"><value>100000</value></property>  
</bean>  
  
这个例子定义了一个CookieLocaleResolver。  
  
表 12.6. WebApplicationContext中的特殊bean  
属性     缺省值     描述  
cookieName     classname + LOCALE     cookie名  
cookieMaxAge     Integer.MAX\_INT     cookie在客户端存在的最大时间。如果该值是-1，这个cookie一直存在，直到客户关闭它的浏览器  
cookiePath     /     使用这个参数，你可以限制cookie只有你的一部分网站页面可以访问。当cookiePath被指定，cookie只能被该目录以及子目录的页面访问  
12.6.3. SessionLocaleResolver  
  
SessionLocaleResolver允许你从用户请求相关的会话中获取本地化信息。  
12.6.4. LocaleChangeInterceptor  
  
你可以使用LocaleChangeInterceptor修改本地化信息。这个拦截器需要添加到处理器映射中（参考第 12.4 节 “处理器映射”），并且它会在请求中检查参数修改本地化信息（它在上下文中的LocaleResolver中调用setLocale()）。  
  
<bean id="localeChangeInterceptor"   
    class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">  
  <property name="paramName"><value>siteLanguage</value></property>  
</bean>  
      
<bean id="localeResolver"  
    class="org.springframework.web.servlet.i18n.CookieLocaleResolver"/>  
        
<bean id="urlMapping"   
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">   
  <property name="interceptors">  
    <list>  
        <ref local="localeChangeInterceptor"/>  
    </list>  
  </property>  
  <property name="mappings">  
    <props>  
        <prop key="/\*\*/\*.view">someController</prop>  
    </props>  
  </property>  
</bean>  
  
所有包含参数siteLanguage的\*.view资源的请求都会改变本地化信息。所以<http://www.sf.net/home.view?siteLanguage=nl>的请求会将网站语言修改为荷兰语。  
12.7. 主题使用  
  
空段落  
12.8. Spring对multipart（文件上传）的支持  
12.8.1. 介绍  
  
Spring由内置的multipart支持web应用中的文件上传。multipart支持的设计是通过定义org.springframework.web.multipart包中的插件对象MultipartResovler来完成的。Spring提供MultipartResolver可以支持Commons FileUpload (<http://jakarta.apache.org/commons/fileupload>)和COS FileUpload (<http://www.servlets.com/cos>)。本章后面的部分描述了文件上传是如何支持的。  
  
缺省，Spring是没有multipart处理，因为一些开发者想要自己处理它们。如果你想使用Spring的multipart，需要在web应用的上下文中添加multipart解析器。这样，每个请求就会被检查是否包含multipart。然而，如果请求中包含multipart，你的上下文中定义的MultipartResolver就会解析它。这样，你请求中的multipart属性就会象其它属性一样被处理。  
12.8.2. 使用MultipartResolver  
  
下面的例子说明了如何使用CommonsMultipartResolver：  
  
<bean id="multipartResolver"   
  class="org.springframework.web.multipart.commons.CommonsMultipartResolver">  
  
  <!-- one of the properties available; the maximum file size in bytes -->  
  <property name="maximumFileSize">  
    <value>100000</value>  
  </property>  
</bean>  
  
这个例子使用CosMultipartResolver：  
  
<bean id="multipartResolver"   
  class="org.springframework.web.multipart.cos.CosMultipartResolver">  
  
  <!-- one of the properties available; the maximum file size in bytes -->  
  <property name="maximumFileSize">  
    <value>100000</value>  
  </property>  
</bean>  
  
当然你需要在你的classpath中为multipart解析器提供正确的jar文件。如果是CommonsMultipartResolver，你需要使用commons-fileupload.jar，如果是CosMultipartResolver，使用cos.jar。  
  
你已经看到如何设置Spring处理multipart请求，接下来我们看看如何使用它。当Spring的DispatchServlet发现multipart请求时，它会激活定义在上下文中的解析器并处理请求。它通常做的就是将当前的HttpServletRequest封装到支持multipart的MultipartHttpServletRequest。使用MultipartHttpServletRequest，你可以获取请求所包含的multipart信息，在控制器中获取具体的multipart内容。  
12.8.3. 在一个表单中处理multipart  
  
在MultipartResolver 完成multipart解析后，multipart请求就会和其它请求一样被处理。使用multipart，你需要创建一个带文件上传域的表单，让 Spring将文件绑定到你的表单上。就象其它不会自动转换成String或基本类型的属性一样，为了将二进制数据放到你的bean中，你必须用ServletRequestDatabinder注册一个自定义的编辑器。Spring有许多编辑器可以用来处理文件，以及在bean中设置结果。StringMultipartEditor能将文件转换成String（使用用户定义的字符集），ByteArrayMultipartEditor能将文件转换成字节数组。它们就象CustomDateEditor一样工作。  
  
所以，为了在网站中使用表单上传文件，需要声明解析器，将URL映射到控制器，以及处理bean的控制器本身。  
  
<beans>  
  
  ...  
  
  <bean id="multipartResolver"  
    class="org.springframework.web.multipart.commons.CommonsMultipartResolver"/>  
  
  <bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">  
    <property name="mappings">  
        <props>  
          <prop key="/upload.form">fileUploadController</prop>  
        </props>  
    </property>  
  </bean>  
  
  <bean id="fileUploadController" class="examples.FileUploadController">  
    <property name="commandClass"><value>examples.FileUploadBean</value></property>  
    <property name="formView"><value>fileuploadform</value></property>  
    <property name="successView"><value>confirmation</value></property>  
  </bean>  
  
</beans>  
  
然后，创建控制器和含有文件属性的bean  
  
// snippet from FileUploadController  
public class FileUploadController extends SimpleFormController {  
  
  protected ModelAndView onSubmit(  
    HttpServletRequest request,  
    HttpServletResponse response,  
    Object command,  
    BindException errors)  
    throws ServletException, IOException {  
      
    // cast the bean  
    FileUploadBean bean = (FileUploadBean)command;  
      
    // let's see if there's content there  
    byte[] file = bean.getFile();  
    if (file == null) {  
        // hmm, that's strange, the user did not upload anything  
    }  
  
    // well, let's do nothing with the bean for now and return:       
    return super.onSubmit(request, response, command, errors);  
  }  
    
  protected void initBinder(  
    HttpServletRequest request,  
    ServletRequestDataBinder binder)  
    throws ServletException {  
    // to actually be able to convert Multipart instance to byte[]  
    // we have to register a custom editor (in this case the  
    // ByteArrayMultipartEditor  
    binder.registerCustomEditor(byte[].class, new ByteArrayMultipartFileEditor());  
    // now Spring knows how to handle multipart object and convert them  
  }  
      
}  
  
// snippet from FileUploadBean  
public class FileUploadBean {  
  private byte[] file;  
    
  public void setFile(byte[] file) {  
    this.file = file;  
  }  
    
  public byte[] getFile() {  
    return file;  
  }  
}  
  
你会看到，FileUploadBean有一个byte[]类型的属性来存放文件。控制器注册一个自定义的编辑器以便让Spring知道如何将解析器发现的 multipart对象转换成bean指定的属性。在这些例子中，没有对bean的byte[]类型的属性做任何处理，但是在实际中可以做任何你想做的（将文件存储在数据库中，通过电子邮件发送给某人，等等）。  
  
但是我们还没有结束。为了让用户能真正上传些东西，我们必须创建表单：  
  
<html>  
  <head>  
    <title>Upload a file please</title>  
  </head>  
  <body>  
    <h1>Please upload a file</h1>  
    <form method="post" action="upload.form" enctype="multipart/form-data">  
        <input type="file" name="file"/>  
        <input type="submit"/>  
    </form>  
  </body>  
</html>  
  
你可以看到，我们在bean的byte[]类型的属性后面创建了一个域。我们还添加了编码属性以便让浏览器知道如何编码multipart的域（千万不要忘记！）现在就可以工作了。  
12.9. 处理异常  
  
Spring提供了HandlerExceptionResolvers来帮助处理控制器处理你的请求时所发生的异常。HandlerExceptionResolvers在某种程度上和你在web应用的web.xml中定义的异常映射很相象。然而，它们提供了一种更灵活的处理异常的方式。首先，HandlerExceptionResolver通知你当异常抛出时如何处理。并且，这种可编程的异常处理方式使得在请求被传递到另一个URL前给了你更多的响应选择。（这就和使用servlet特定异常映射的情况一样）。  
  
实现HandlerExceptionResolver需要实现resolveException(Exception, Handler)方法并返回ModelAndView，除了HandlerExceptionResolver，你还可以使用SimpleMappingExceptionResolver。这个解析器使你能够获取任何抛出的异常的类名，并将它映射到视图名。这和servlet API的异常映射在功能上是等价的，但是它还为不同的处理器抛出的异常做更细粒度的映射提供可能