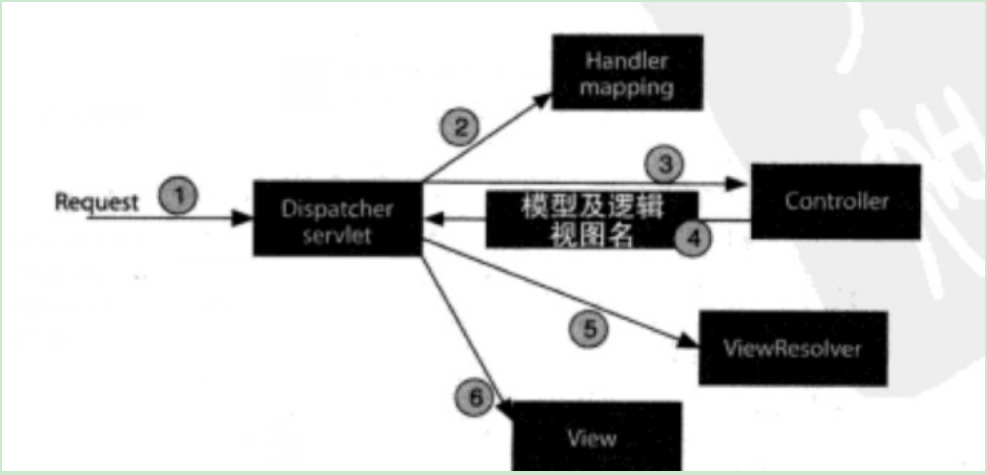
阅读笔记6

第六章Spring mvc

Spring mvc就是mvc层的框架
可以帮助你建立Spring框架那样的灵活与耦合的web应用程序

每当用户在wen浏览器中点击链接或提交表单的时候 请求就开始工作了
请求会由DispactcherServlet分配给控制器 在控制器完成处理后 接着请求会被发送给一个视图
Spring MVC所有的请求都会经过一个前端控制器Servlet
在Spring MVC中 DispatcherServlet就是前端控制器



DispatcherServlet的任务就是将
请求发送给Spring MVC层控制器 控制器就是用于处理请求的Spring组件
通过控制器处理完任务之后 产生了信息 再返回给DispatcherServlet
然后DispatcherServlet来传递到相关的视图层上 列如JSP

首先我们要搭建Spring MVC的环境
在web.xml我们要这么配置

```
<servlet>
    <servlet-name>spitter</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spitter</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

通过将url-pattern这个类映射到/ 声明了它会做为默认的servlet并且会处理所有的请求 包括静态资源的分配

这里的MVC的命名空间有点特殊 所以我就粘贴在下面
这个命名空间还是非常重要的

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:util="http://www.springframework.org/schema/util"
xmlns:mvc="http://www.springframework.org/schema/mvc"
xmlns:p="http://www.springframework.org/schema/p"
xsi:schemaLocation="http://www.springframework.org/schema/mvc http://www.springframework.org/
schema/mvc/spring-mvc-3.0.xsd
      http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans
/spring-beans-3.0.xsd
      http://www.springframework.org/schema/util http://www.springframework.org/schema/util/s
```

```
pring-util-3.0.xsd
      http://www.springframework.org/schema/context http://www.springframework.org/schema/con
text/spring-context-3.0.xsd">



       <mvc:resources mapping="/resources/**"  location="/resources/" />

</beans>
```

在这里我们进行了一个关于<mvc：resources>的注解 这个是用来处理静态资源的请求的 <mvc：resources>建立了一个服务于静态资源的处理器 **属性mapping被设置为/resources/** 表示路径必须从 /resources开始** location表示要提供服务的文件位置 所有以/resources路径开头的请求都会自动由应用程序根目录下的/resource目录提供服务

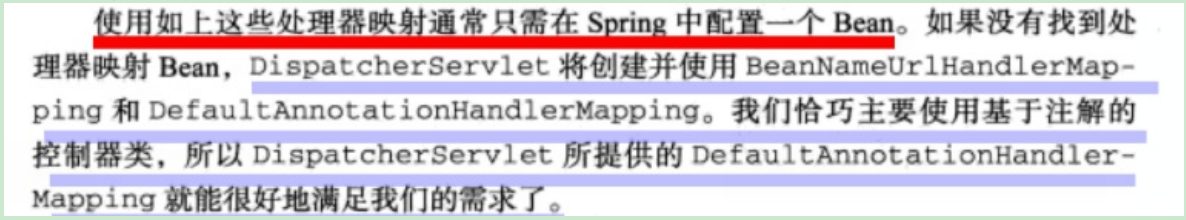
配置注解驱动的Spring MVC

在Spring MVC中里内置了多个处理器映射 其中在这里我们使用的是DefaultAnnotationHandlerMapping这个基于注解的 这个也是同BeanNameUrlHandlerMapping

- BeanNameUrlHandlerMapping：根据控制器 Bean 的名字将控制器映射到 URL。

- ControllerBeanNameHandlerMapping：与 BeanNameUrlHandlerMapping 类似，根据控制器 Bean 的名字将控制器映射到 URL。使用该处理器映射实现，Bean 的名字不需要遵循 URL 的约定。
- ControllerClassNameHandlerMapping：通过使用控制器的类名作为 URL 基础将控制器映射到 URL。
- DefaultAnnotationHandlerMapping：将请求映射给使用 @RequestMapping 注解的控制器和控制器方法。
- SimpleUrlHandlerMapping：使用定义在 Spring 应用上下文的属性集合将控制器映射到 URL。

一共有这些处理器映射



使用如上这些处理器映射通常只需在 Spring 中配置一个 Bean。如果没有找到处理器映射 Bean，DispatcherServlet 将创建并使用 BeanNameUrlHandlerMapping 和 DefaultAnnotationHandlerMapping。我们恰巧主要使用基于注解的控制器类，所以 DispatcherServlet 所提供的 DefaultAnnotationHandlerMapping 就能很好地满足我们的需求了。

在构建控制器的时候 我们还需要**使用注解将请求参数绑定到控制器的方法参数上进行校验以及信息转换**
我们只需要在spitter-servlet.xml上添加一行就能达到这个效果

```
<mvc:annotation-driven/>
```

这个mvc:annotation-driven 标签有足够的威力 它注册了多 个特性 包括JSP-303校验支持 信息转换以及对域格式化的支持
定义首页的控制器

```
@Controller
public class HomeController {


public  static  final  int DEFAULT_SPITTLES_PER_PAGE=25;
    private SpitterService  spitterService;


@Inject
public HomeController(SpitterService spitterService) {
this.spitterService = spitterService;
}

@RequestMapping({"/","/home"})
public  String showHomePage(Map<String,Object> model){
       model.put("spittles",spitterService.getRecentSpittles(DEFAULT_SPITTLES_PER_PAGE));
```

```
        return  "home";
}
}
```

这样就是一个关于首页的控制器了
这里我们主要是声明了三个注解 这三个注解分别是@Inject（自动注入）@RequestMapping（指明了这个方法是处理“/“和”/home“路径的请求） 以及开头的@Controller注明这是一个Bean
在XML的具体配置中 我们将其具体的配置注明

注意这里的Map集合 这个Map代表了模型---控制器与视图之间传递的数据 这里的Map中的数据通过spitterService.getRecentSpiiles（）方法得到Spittle列表后 将这个数据置于模型Map中 这样就能在视图成显示出来了 这个就是个视图层与控制器传递的关键 这个可以传输数据 但是同时
我们也可以使用一些本来就有request这些属性来传值 还有这里有一个问题 在有的web.xml版本中 EL表达式不能使用 其原因是被禁用掉了
需要加入

```
<%@page isELIgnored="false"%>
```

这个标签

```
  <context:component-scan base-package="com.test.Spitter.mvc"/>
```

扫描并注册这个我们在具体类中声明为Bean的类

由于这个类只有三个注解
是可以进行相关的测试 因为基本没有相关的Spring代码 非常的易于测试

```
package com.test.Spitter.mvc;


import static java.util.Arrays.*;

import static org.junit.Assert.*;
import static org.mockito.Mockito.*;


import java.util.HashMap;
import java.util.List;

import com.test.Spitter.orm.SpitterService;
import com.test.Spitter.orm.Spittle;
import org.junit.Test;


public class HomeControllerTest {

@Test
public void shouldDisplayRecentSpittles() {
    List<Spittle> expectedSpittles =
asList(new Spittle(), new Spittle(), new Spittle());

SpitterService spitterService = mock(SpitterService.class);//<co id="co_mockSpitterService"/>
    /*模拟SpitterService*/

when(spitterService.getRecentSpittles(HomeController.DEFAULT_SPITTLES_PER_PAGE )).
        thenReturn(expectedSpittles);

HomeController controller =
new HomeController(spitterService); //<co id="co_createController"/>
    /*创建控制器*/
HashMap<String, Object> model = new HashMap<String, Object>();
String viewName = controller.showHomePage(model); //<co id="co_callShowHomePage"/>
    /*调用处理方法*/

    /*对结果进行断言*/
assertEquals("home", viewName);
assertSame(expectedSpittles, model.get("spittles")); //<co id="co_assertResults"/>
verify(spitterService).getRecentSpittles(HomeController.DEFAULT_SPITTLES_PER_PAGE );
}
}
```

这里牵扯到一个关于测试的jar包

```
import static org.mockito.Mockito.*;
```

这个的具体的一些使用方法就在这个类中了

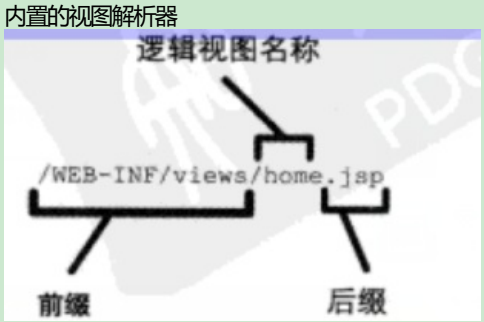
解析视图

处理的最后一步就是要将用户的渲染输出 就是将我们需要输出的东西显示在视图层 确定具体的使用哪个视图层的 就是由DispatcherServlet中来决定的 DispatcherServlet会查找一个视图解析器来将控制器返回的逻辑视图名称转换为实际我们需要的视图

视图解析器 将逻辑视图与org.springframework.web.servlet.View的实现进行相应的匹配

视图解析器	描述
BeanNameViewResolver	查找 <bean> ID 与逻辑视图名称相同 View 的实现
ContentNegotiatingViewResolver	委托给一个或多个视图解析器，而选择哪一个取决于请求的内容类型（在第 11 章将介绍这个视图解析器）
FreeMarkerViewResolver	查找一个基于 FreeMarker 的模板，它的路径根据加完前缀和后缀的逻辑视图名称来确定
InternalResourceViewResolver	在 Web 应用程序的 WAR 文件中查找视图模板，视图模板的路径根据加完前缀和后缀的逻辑视图名称来确定
JasperReportsViewResolver	根据加完前缀和后缀的逻辑视图名称来查找一个 Jasper Report 报表文件
ResourceBundleViewResolver	根据属性文件（properties file）来查找 View 实现
TilesViewResolver	查找通过 Tiles 模板定义的视图，模板的名字与逻辑视图名称相同
UrlBasedViewResolver	这是一些其他视图解析器（如 InternalResourceViewResolver）的基类。它可以单独使用，但是没有它的子类强大。例如，UrlBasedViewResolver 不能基于当前的语言环境来解析视图
VelocityLayoutViewResolver	它是 VelocityViewResolver 的子类，它支持通过 Spring 的 VelocityLayoutView（模仿 VelocityVelocityLayoutServlet 的 View 实现）来进行页面的组合
VelocityViewResolver	解析基于 Velocity 的视图，Velocity 模板的路径根据加完前缀和后缀的逻辑视图名来确定
XmlViewResolver	查找在 XML 文件（/WEB-INF/views.xml）中声明的 View 实现。这个视图解析器与 BeanNameViewResolver 很类似，但在这里视图 <bean> 的声明与应用程序 Spring 上下文的其他 <bean> 是分开的
XsltViewResolver	解析基于 XSLT 的视图，XSLT 样式表的路径根据加完前缀和后缀的逻辑视图名称来确定

这就是Spring
内置的视图解析器



这样配置就是配置视图解析器 当DispatcherServlet要求这个InternalResourceViewResolver解析视图时 将获取一个逻辑视图名称 我们在这里配置的俩个属性就是上图表示的前缀与后缀

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/views/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

由于在这里我们使用了一些JSTL标签 所以通过设置viewClass属性来将InterabkResourceViewResoverView来替换为JstlView

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/views/"/>
    <property name="suffix" value=".jsp"/>
    <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
```

```
</bean>
```

这样JstlView就将请求传给JSP 但是 这个还会公布特定的JSTL的请求属性 这有就可使用JSTL的国际化支持了

其他的一大堆视图解析器都与这个差不多 都是为了逻辑视图添加前缀 与后缀 然后再去查找视图模板

对没有复杂构造的Web简单应用来讲 这样是足够的 但是我们复杂的话 就涉及到一个工具 就是Apache Tiles
这是个布局管理器 将页面分成片段并在运行的时候进行组装成完整的页面

```
<bean class="org.springframework.web.servlet.view.tiles2.TilesViewResolver"></bean>

 <bean class="org.springframework.web.servlet.view.tiles2.TilesConfigurer">
     <property name="definitions">
         <list>
             <value>/WEB-INF/views/**/views.xml</value>
         </list>
     </property>
 </bean>
```

首先 我们配置一个TilesViewResolver这个视图解析器 会查找逻辑视图名称与Tiles定义名称相同的Tiles模板定义 并将其作为视图
然后 由于Spring缺少知道是怎么Tiles定义的 我们需要配置一个TilesConfigurer这个Bean来进行记录这个信息 通过这个 TilesConfigurer
加载一个或者多个Tiles定义 通过definitions属性来装配我们需要的信息 也就是views.xml的定义文件 这里的**是ANT风格代表在views 这个文件下查找所有的名为views.xml文件

这样在TilesViewResolver在解析逻辑视图名称时首先会找到home模板
DispatcherServlet会将这个请求传递给Tiles 用Home定义来渲染结果

```
<!DOCTYPE tiles-definitions PUBLIC
      "-//Apache Software Foundation//DTD Tiles Configuration 2.1//EN"
      "http://tiles.apache.org/dtds/tiles-config_2_1.dtd">

<tiles-definitions>
<!--<start id="tile_template"/>-->
<definition name="template"
template="/WEB-INF/views/main_template.jsp">
     <put-attribute name="top"
value="/WEB-INF/views/tiles/spittleForm.jsp" />
     <put-attribute name="side"
value="/WEB-INF/views/tiles/signinsignup.jsp" />
   </definition>
<!--<end id="tile_template"/>-->

<!--<start id="tile_home"/>-->
<definition name="home" extends="template">
     <put-attribute name="content" value="/WEB-INF/views/home.jsp" />
   </definition>
<!--<end id="tile_home"/>-->

<definition name="login" extends="template">
     <put-attribute name="content" value="/WEB-INF/views/login.jsp" />
     <put-attribute name="side" value="/WEB-INF/views/tiles/alreadyamember.jsp" />
   </definition>

   <definition name="admin" extends="template">
     <put-attribute name="content" value="/WEB-INF/views/admin.jsp" />
   </definition>
</tiles-definitions>
```

这个就是关于views.xml的配置情况 定义了通用的模板 也定义了我们要使用的home 并且将home使用了这个模板 这个模板是通用的

在这里我进行调试了2天 jar包的问题很严重 一定要注意jar包匹配是否正确 不然就会出错

注意这里的@Inject（这个有问题 我们采用的是@Autowired ）这里我们要进行测试的时候必须要进行关闭 不然的话就会出错 因为我们这里没有能够进行装配的值 就会导致出现HTTP 500错误

要解决这样的错误 我们必须就将上一章我们写的其他配置文件加载进来
我们只需要在web.xml中进行如下配置就好

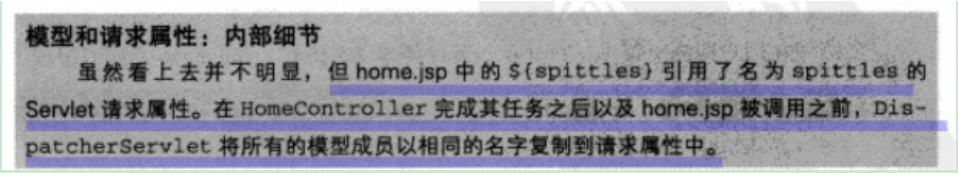
我们可以加入ContextLoadListener能够发挥作用的地方了 这是个Servlet监听器 同时 由于默认的是applicantContext.xml 我们需要重写默认 于是我们使用了

```
<context-param>
    <param-name>contextConfigLocation</param-name>
<!--<param-value>classpath:orm/Spring-tx.xml</param-value>
    <param-value>classpath:orm/Spring_AOPTx.xml</param-value>-->
<param-value>
/WEB-INF/spitter-spitterService.xml
</param-value>
</context-param>
```

下面是总代码 在web.xml中的

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



<context-param>
    <param-name>contextConfigLocation</param-name>
<!--<param-value>classpath:orm/Spring-tx.xml</param-value>
    <param-value>classpath:orm/Spring_AOPTx.xml</param-value>-->
<param-value>
/WEB-INF/spitter-spitterService.xml
</param-value>
</context-param>
```

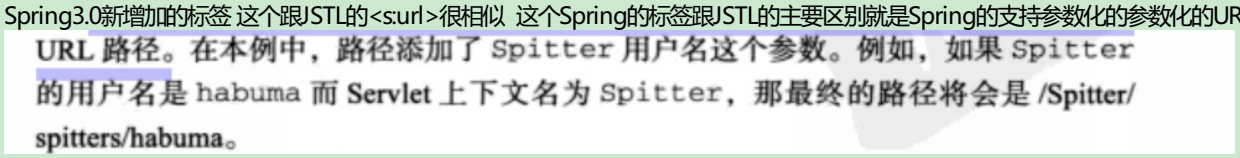
```
<div>
  <h2>A global community of friends and strangers spitting out their
  inner-most and personal thoughts on the web for everyone else to
  see.</h2>
  <h3>Look at what these people are spitting right now...</h3>

  <ol class="spittle-list">
<c:forEach var="spittle" items="${spittles}"><!--<co id="cp_foreach_spittles"/>-->

<s:url value="/spitters/{spitterName}"
var="spitter_url" ><!--<co id="cp_spitter_url"/>-->
<s:param name="spitterName"
value="${spittle.spitter.username}" />
</s:url>

<li><span class="spittleListImage">
        <img src=
"http://s3.amazonaws.com/spitterImages/${spittle.spitter.id}.jpg"
width="48"
border="0"
align="middle"
onError=
"this.src='<s:url value="/resources/images"/>/spitter_avatar.png';"/>
      </span>
      <span class="spittleListText">
        <a href="${spitter_url}">              <!--<co id="cp_spitter_properties"/>-->
<c:out value="${spittle.spitter.username}" /></a>
- <c:out value="${spittle.text}" /><br/>
         <small><fmt:formatDate value="${spittle.when}"
pattern="hh:mma MMM d, yyyy" /></small>
      </span></li>
</c:forEach>
</ol>
</div>
```



模型和请求属性：内部细节

虽然看上去并不明显，但 home.jsp 中的 ${spittles} 引用了名为 spittles 的 Servlet 请求属性。在 HomeController 完成其任务之后以及 home.jsp 被调用之前，DispatcherServlet 将所有的模型成员以相同的名字复制到请求属性中。

在这里的home.jsp的核心是在<c:forEach>标签 在这里遍历了Spittle的列并在此过程中渲染了每个记录的细节
注意这里的<s:url>标签 我们使用这个标签来创建一个相对于Servlet的上下文的URL 来指向每个Spittle所属的Spitter <s:url>是Spring3.0新增加的标签 这个跟JSTL的<s:url>很相似 这个Spring的标签跟JSTL的主要区别就是Spring的支持参数化的参数化的URL路径



URL 路径。在本例中，路径添加了 Spitter 用户名这个参数。例如，如果 Spitter 的用户名是 habuma 而 Servlet 上下文名为 Spitter，那最终的路径将会是 /Spitter/spitters/habuma。

但是这个HomeController太简单 是不能进行处理控制器的输入 控制器往往基于URL参数或者表单数据所传递进来的信息执行一些逻辑 我们需要编写处理输入的控制器

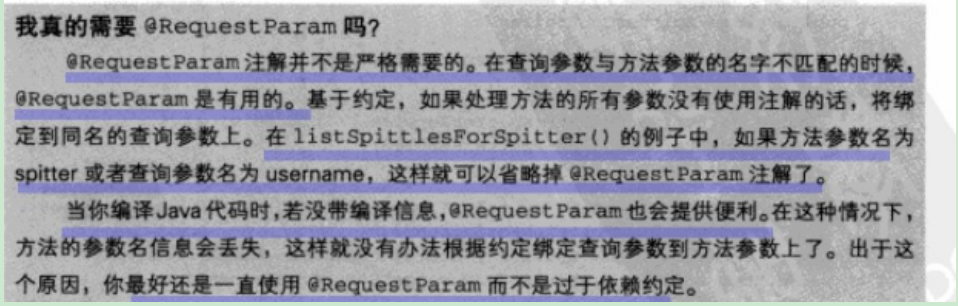
首先实现SpitterController控制器

下面的代码就是SpitterController控制器

```
@RequestMapping("/spitter")
@Controller
public class SpitterController {

private  final SpitterService spitterService;

@Autowired
public SpitterController(SpitterService spitterService) {
this.spitterService = spitterService;
}

@RequestMapping(value = "/spittles" ,method = RequestMethod.GET)
public  String listSpittlesForSpitter(
@RequestParam("spitter")String username,Model model){
        Spitter spitter=spitterService.getSpitter(username);
model.addAttribute(spitter);
model.addAttribute(spitterService.getSpittlesForSpitter(username));
        return "spittles/list";

}

}
```

这里的代码相对于上个控制器来说有些不一样的地方 比如类的定义处也定义了一个@RequestMapping这个注解 这个注解代表着这个控制器的根URL路径 最终会在SpitterController中添加多个处理方法 每个处理方法会对应不同的请求 但是这里的类的@RequestMapping这个注解代表每个请求前面都必须加上/spitter开头

方法级别的还有一个在前面我们没有说过的注解 这就是 @RequestParam这个注解 这个注解表明它的值应该是根据请求中名为spitter的参数的查询参数来获取 listSpittlesForSpitter()将使用这个参数来查找Spitter对象及其Spittle列表

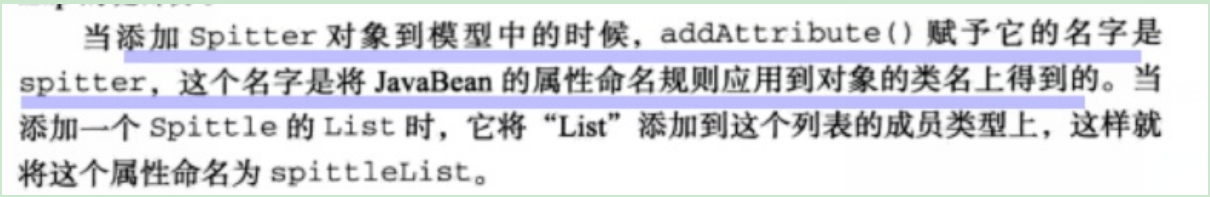


我真的需要 @RequestParam 吗？

@RequestParam 注解并不是严格需要的。在查询参数与方法参数的名字不匹配的时候，@RequestParam 是有用的。基于约定，如果处理方法的所有参数没有使用注解的话，将绑定到同名的查询参数上。在 listSpittlesForSpitter() 的例子中，如果方法参数名为 spitter 或者查询参数名为 username，这样就可以省略掉 @RequestParam 注解了。

当你编译 Java 代码时，若没带编译信息，@RequestParam 也会提供便利。在这种情况下，方法的参数名信息会丢失，这样就没有办法根据约定绑定查询参数到方法参数上了。出于这个原因，你最好还是一直使用 @RequestParam 而不是过于依赖约定。

还有这里的Model参数 这个参数 内部其实很可能是一个Map<String,Onject>但是Model提供了几个更便利的方法来填充模型 如在这里提醒的addAttribute()方法 addAttribute这个方法与Map中put方法的作用是一样的 只不过这个是能够自己进行计算map键值的部分 关于计算的规则如下



然后我们随之将这个页面进行传递到JSP页面上 进行渲染视图 需要注意的是 这里的JSP页面 由于我们在控制器中使用了@RequestParam注解 我们必须在访问这个页面的时候必须传递这个参数才能到达这个页面 不然是访问不到的
当然 我们也同时需要加入views这个Tile定义

```
<definition name="spittles/list" extends="template">
 <put-attribute name="content"
value="/WEB-INF/views/spittles/list.jsp" />
</definition>
```

给它增加模板

```
<div>
  <h2>Spittle for ${spitter.username}</h2>

  <script>
function deleteSpittle(id) {
if(confirm("Are you sure you want to delete this spittle?")) {
document["deleteSpittle_" + id].submit();
}
    }
</script>

  <table cellspacing="15" class="horizontalRuled">
<c:forEach items="${spittleList}" var="spittle">
<s:url value="/spitters/${spittle.spitter.username}" var="spitter_url" />
<tr>
      <td>
          <img src="h   ttp://s3.amazonaws.com/spitterImages/${spittle.spitter.id}.jpg"
width="48"
border="0"
align="middle"
onError="this.src='<s:url value="/resources/images"/>/spitter_avatar.png';"/></td>
      <td><a href="${spitter_url}">${spittle.spitter.username}</a>
          <small><c:out value="${spittle.text}" /><br/><small>
<fmt:formatDate value="${spittle.when}" pattern="hh:mma MMM d, yyyy" />
</small></small>
<s:url value="/spittles/${spittle.id}" var="spittle_url" />
<sf:form method="delete" action="${spittle_url}"
name="deleteSpittle_${spittle.id}"
cssClass="deleteForm">
<input type="submit" value="Delete"/>
</sf:form>
</td>
    </tr>
</c:forEach>
</table>
</div>
```

这个头部就是引用了spitter.username属性 还有其他的大部分也是迭代Spitter列表并且展现他们的详细信息
这里的spitterlist就是Model的addAttribute()方法赋给他的名字

这个是进行访问的spitter用户的页面

下面到笔记7