# SpringMVC

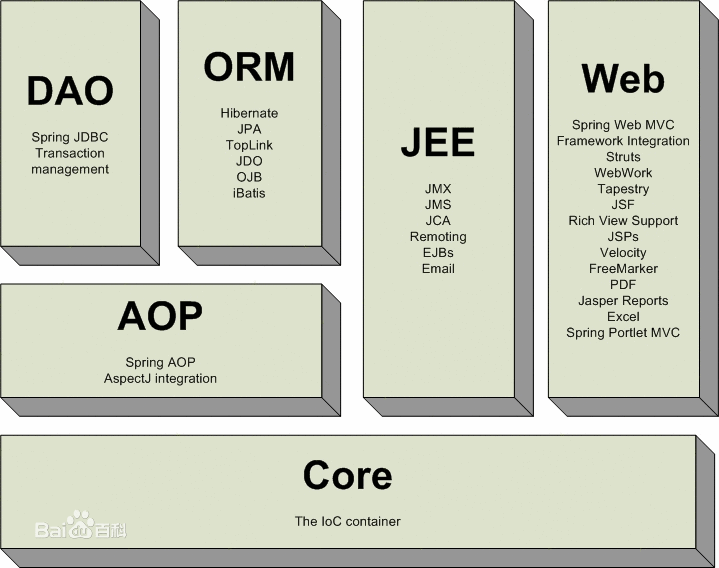
## 简介

Spring是一个独立的框架，它不需要依赖于任何Web服务器或容器。它即可以在独立的JavaSE项目中使用，也可以在JavaWeb项目中使用。

Spring框架提供了构建Web应用程序的全功能MVC模块--SpringMVC。

简言之，SpringMVC是一个Web层面的，基于MVC设计模式的框架。

## Spring模块示例图



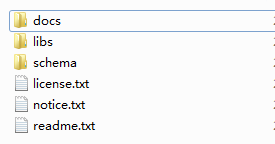
从图中可以看出来，SpringMVC是Spring的一部分。

# 第一个SpringMVC程序

## 下载spring压缩包，并解压缩







Docs:存放spring相关文档，包含开发指南，API参考文档

Libs:该文件夹下的jar分三类：

①：Spring框架class文件的jar包；

②：Spring框架源文件的压缩包；

③：Spring框架API文档的压缩包。

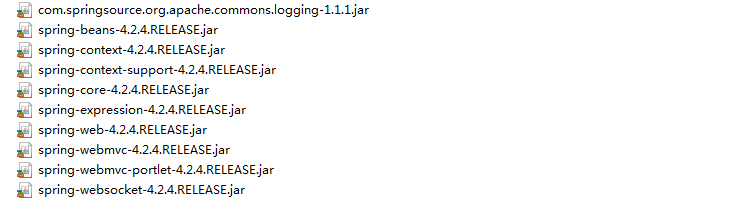
Schema:该文件夹下包含了Spring各种配置文件的XML Schema文档。

readme.txt,notice.txt,licence.txt等说明性文档

## 下载commons-logging.jar

这是Spring核心容器依赖的jar包。Spring虽然依赖它，但是并不提供它的下载。需要自行下载。

## 创建一个web项目，FirstSpringMvc。把如下jar包添加到工程中。



## 创建HelloController

**public** **class** HelloController **implements** Controller{

@Override

**public** ModelAndView handleRequest(HttpServletRequest req, HttpServletResponse res) **throws** Exception {

//ModelAndView用来保存模型和视图信息

ModelAndView modelAndView = **new** ModelAndView();

//添加模型信息

modelAndView.addObject("userName", "大王叫我来巡山");

//添加视图信息【物理视图】

modelAndView.setViewName("/WEB-INF/jsp/welcome.jsp");

**return** modelAndView;

}

}

注意：自定义的Controller要实现Controller接口

## 在/WEB-INF/目录创建jsp目录，jsp目录下创建welcome.jsp

<body>

在页面上使用EL表达式就可以获得ModelAndView中的模型数据

<h2>${userName}</h2>

</body>

## 创建springmvc的配置文件springmvc-servlet.xml

在WEB-INF下创建springmvc-servlet.xml,加入约束信息

<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.xsd"*>

<!-- 配置处理器映射器：它负责将浏览器url映射到需要执行的Controller -->

<bean class=*"org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"* />

<!-- 配置处理器适配器：它负责执行Controller -->

<bean class=*"org.springframework.web.portlet.mvc.SimpleControllerHandlerAdapter"* />

<!-- 配置自定义的HelloController -->

<bean name=*"/hello"* class=*"com.xiaoka.controller.HelloController"*/>

</beans>

## 编辑Web.xml，加载spring-config.xml

配置springmvc的前端控制器DispatcherServlet，它是一个servlet,只需要像配置servlet那样配置它即可。

<!-- SpringMVC默认加载/WEB/INF/<servlet-name>-servlet.xml的配置文件 -->

<!-- 例如：本例<spring-name>的值是springmvc,那么就默认加载WEB-INF/springmvc-servlet.xml文件 -->

<servlet>

<servlet-name>springmvc</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>springmvc</servlet-name>

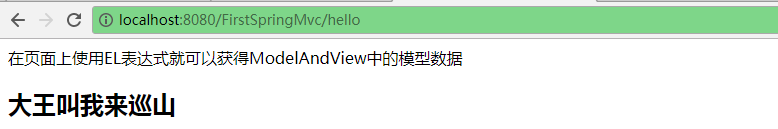
<url-pattern>/</url-pattern>

</servlet-mapping>

## 运行测试

在浏览器输入URL

http://localhost:8080/FirstSpringMvc/hello



# 视图解析器

在第一个SprringMVC章节，我们已经成功访问到Controller，并转发到了视图页面。在Controller中有这样的代码

//添加视图信息【物理视图】

modelAndView.setViewName("/WEB-INF/jsp/welcome.jsp");

实际开发中，我们会用视图解析器来解析逻辑视图到物理视图之间的映射。

## 修改springmvc-servlet.xml配置文件

在springmvc-servlet.xml中配置视图解析

<!-- 配置视图解析器：它将负责解析Controller中的逻辑视图地址 -->

<bean id=*"viewResolver"*

class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

<!-- 逻辑视图的前缀 -->

<property name=*"prefix"* value=*"/WEB-INF/jsp/"* />

<!-- 逻辑视图的后缀 -->

<property name=*"suffix"* value=*".jsp"* />

</bean>

## 修改HelloController

**public** **class** HelloController **implements** Controller {

@Override

**public** ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) **throws** Exception {

// ModelAndView用来保存模型和视图信息

ModelAndView modelAndView = **new** ModelAndView();

// 添加模型信息

modelAndView.addObject("userName", "大王叫我来巡山");

/\*

\* 添加视图信息，这将是一个逻辑视图

\* 视图解析器会将前缀拼接在逻辑视图的前边

\* 将后缀拼接在逻辑视图的后边

\* 拼接结果：/WEB-INF/jsp/welcome.jsp

\*

\*/

modelAndView.setViewName("welcome");

// 添加视图信息【物理视图】

// modelAndView.setViewName("/WEB-INF/jsp/welcome.jsp");

**return** modelAndView;

}

}

使用视图解析器的好处则是可以在Controller中省略前缀和后缀的书写，方便以后代码的统一维护。

## 运行测试

运行2.8章节处的测试URL，可以发现依然能够成功的访问Controller，并转发到视图页面。

# 乱码问题

如果在访问中文中出现了乱码，那么则进行如下配置

## 修改Tomcat的请求编码

Tomcat默认使用ISO-8859-1编码来接受请求，只需要将servet.xml请求编码指定为UTF-8即可。



这个用来解决GET请求乱码

## 添加SpringMVC的过滤器

在web.xml中添加SpringMVC的过滤器，指定使用UTF-8编码

<filter>

<filter-name>characterEncoding</filter-name>

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

<init-param>

<param-name>encoding</param-name>

<param-value>UTF-8</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>characterEncoding</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

这个用来解决POST乱码

# 处理器映射器

处理器映射器负责映射URL到具体的Contriller，SpingMVC提供了三个常用的处理器映射器来供开发者使用。

## BeanNameUrlHandlerMapping

该处理器映射器是根据SpringMVC的配置文件中，Controller的name属性值来进行访问。在第一个SpringMVC程序中，我们的示例代码就使用了该处理器映射器。

<bean name=*"/hello"* class=*"com.xiaoka.controller.HelloController"*/>

这里的Controll配置的name属性值为hello，所以我们的访问路径是

http://localhost:8080/FirstSpringMvc/hello

此处不再赘述

## SimpleUrlHandlerMaping

该处理器可以集中对Controller进行配置，需要结合Controller的id属性一起使用。

### 修改SpringMVC的配置文件

在SpringMVC配置文件中添加该处理器映射器，并给HelloController添加上id属性

<!-- 简单URL处理器映射器 -->

<!-- ①prop的key是在浏览器的访问路径URL-->

<!-- ②prop的标签体的helloController将映射到id值为该值的Controller -->

<bean class=*"org.springframework.web.servlet.handler.SimpleUrlHandlerMapping"*>

<property name=*"mappings"*>

<props>

<prop key=*"/hello1"*>helloController</prop>

<prop key=*"/hello2"*>helloController</prop>

</props>

</property>

</bean>

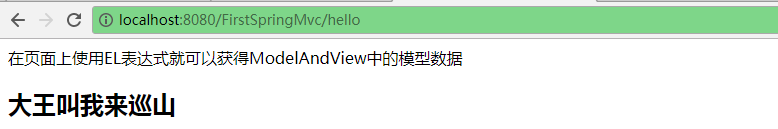
<!-- 配置自定义的HelloController -->

<bean id=*"helloController"* name=*"/hello"*class=*"com.xiaoka.mvc.controller.HelloController"* />

### 运行测试

[http://localhost:8080/FirstSpringMvc/hello1](http://localhost:8080/FirstSpringMvc/hello2)

<http://localhost:8080/FirstSpringMvc/hello2>



可以发现无论是hello1还是hello2的URL，都可以正确的映射到HelloController。

## ControllerClassNameHandlerMapping

该处理器映射器根据Controller类名来访问,类名首字母小写

### 修改SpringMVC的配置文件

在SpringMVC配置文件中添加该处理器映射器,它的规则是可以让Controller使用类名称来访问。

<!-- 配置根据类名称访问的处理器映射器，所有的Controller都可以根据类名称来访问[类名首字母小写] -->

<bean class=*"org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping"*/>

### 运行测试

<http://localhost:8080/FirstSpringMvc/helloController>

可以发现，使用类名称的小驼峰式能够正确的映射到Controller。

## 小结

这三个处理器映射器可以共存，无论是分别映射某个Controller还是同时映射多个Controlle规则都是生效的。

# 处理器适配器

处理器适配器负责执行Controller，SpringMVC提供了两个常用的处理器适配器

## SimpleControllerHandlerAdapter

该处理器适配器是SpringMVC的默认处理器适配器，负责执行实现了Controller接口的Controller。在第一个SpringMVC程序中，本示例代码则是使用该适配器。此处不再赘述。

## HttpRequestHandlerAdapter

该处理器适配器负责执行实现了HttpRequestHandler接口的Controller，它无限接近Servlet开发。

### 创建MyHttpController

/\*该Controller是无限接近Servlet开发的一个Controller\*/

**public** **class** MyHttpController **implements** HttpRequestHandler {

@Override

**public** **void** handleRequest(HttpServletRequest request, HttpServletResponse response)

**throws** ServletException, IOException {

request.setAttribute("userName", "巡了南山巡北山");

request.getRequestDispatcher("WEB-INF/jsp/welcome.jsp").forward(request, response);

}

}

### 修改SpringMVC的配置文件

#### 配置Controller

<bean name=*"/myHttp"* class=*"com.xiaoka.mvc.controller.MyHttpController"* />

#### 修改简单URL处理器映射器

<bean class=*"org.springframework.web.servlet.handler.SimpleUrlHandlerMapping"*>

<property name=*"mappings"*>

<props>

<prop key=*"/hello1"*>helloController</prop>

<prop key=*"/httpController"*>myHttpController</prop>

</props>

</property>

</bean>

#### 在配置文件中添加处理器适配器

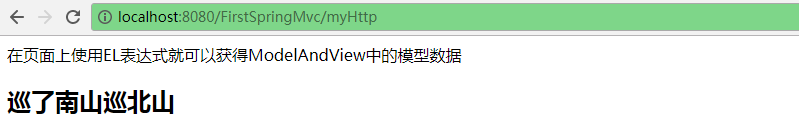
<!-- 该处理器适配器负责执行实现了HttpRequestHandler接口的Controller -->

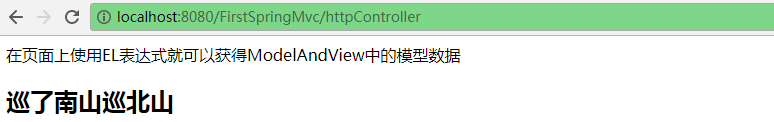
<bean class=*"org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter"* />

### 运行测试

<http://localhost:8080/FirstSpringMvc/myHttp>

<http://localhost:8080/FirstSpringMvc/httpController>





## 小结

这个两个处理器适配器可以共存。

# 命令控制器

在前面的六个章节中，我们主要的认识了SpringMVC的映射器，适配器和视图解析器，已经可以正常的访问到Controller，本章节我们将学习在Controller中接受页面传过来的参数。

SpringMVC通过命令设计模式来接受页面参数

## 创建JavaBean

**public** **class** User {

**private** Integer id;

**private** String userName;

**private** Date birthDay;

//getter/setter和toString()...略...

}

## 创建UserController

**public** **class** UserController **extends** ~~AbstractCommandController~~ {

**public** UserController() {

**this**.~~setCommandClass~~(User.**class**);

}

@Override

**protected** ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object object, BindException arg3) **throws** Exception {

User user = (User) object;

System.***out***.println(user);

ModelAndView modelAndView = **new** ModelAndView();

modelAndView.addObject("user",user);

modelAndView.setViewName("showUser");//待会儿再创建该jsp文件

**return** modelAndView;

}

}

这里可以看到，接受页面参数需要继承~~AbstractCommandController，~~该类简介实现了Controller接口。然而该类在当前4.2版本中却没有，所以需要将3.2版本的Spring包添加到项目中。

## 添加3.2版本jar包

spring-context-3.2.0.RELEASE.jar

spring-context-support-3.2.0.RELEASE.jar

spring-core-3.2.0.RELEASE.jar

spring-web-3.2.0.RELEASE.jar

spring-webmvc-3.2.0.RELEASE.jar

spring-webmvc-portlet-3.2.0.RELEASE.jar

## 修改SpringMVC配置文件

在SpringMVC中配置UserController

<bean name=*"/userController"*class=*"com.xiaoka.mvc.controller.UserController"*/>

## 创建jsp文件

①：在WebContent下创建user.jsp文件

<form action=*"userController"*>

姓名:<input type=*"text"* name=*"userName"*/>

生日:<input type=*"text"* name=*"birthDay"*>

<input type=*"submit"* value=*"提交"*>

</form>

②：在WEB-INF/jsp/下创建showUser.jsp

<body>

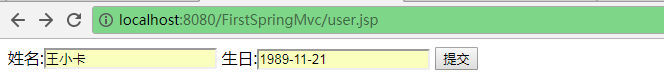
展示用户的jsp页面<br>

${user }

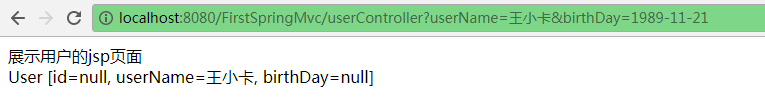
</body>

## 运行测试

①：访问该地址，输入内容，点击提交



②：结果



## 小结

①：String类型的userName正常封装。

②：birthDay则由String转换成Date类型失败。这是因为格式不匹配。如果格式匹配上则可以成功转换。

③：这种命令设计模式封装页面参数是3.X版本的，我们这里作了解。后续学习新版本中的接受参数的方式。

# 日期类型转换

在上一章节，User的birthDay接受失败，解决这个问题很简单，只需要在封装之前拿到页面的birthDay指定好格式转换成Date类型即可。

## 修改UserController

只需要UserController重写initBinder()函数即可。

/\* SpringMVC自动触发该函数，会在handle函数之前执行 \*/

@Override

**protected** **void** initBinder(HttpServletRequest request, ServletRequestDataBinder

binder) **throws** Exception {

**super**.~~initBinder~~(request, binder);

String birthDay = request.getParameter("birthDay");

**if** (birthDay.contains("/")) {

SimpleDateFormat format = **new** SimpleDateFormat("yyyy/MM/dd");

//true：该参数允许为null

CustomDateEditor editor = **new** CustomDateEditor(format, **true**);

binder.registerCustomEditor(Date.**class**, editor);

}

**else** **if** (birthDay.contains("-")) {

SimpleDateFormat format = **new** SimpleDateFormat("yyyy-MM-dd");

CustomDateEditor editor = **new** CustomDateEditor(format, **true**);

binder.registerCustomEditor(Date.**class**, editor);

}

}

# 注解开发

之前的程序中，一个Controller里边只能响应一个请求，而且都需要在SpringMVC的配置文件中进行配置。对于开发工作来讲非常的不方便。

本章我们讲解基于注解的开发方式，注解的开发方式将使开发变得更简洁。

## 第一个注解的SpringMVC项目

### 创建项目

导入spring,springmvc,**aop**,commons-logging.jar等jar包【不要3.X版本的jar包】，总共11个。

### 创建Controller

使用注解方式开发，自定义的Controller不用再继承任何类，不用实现任何接口。

//添加Spring的注解，将Controller类交给Spring来管理

@Controller

**public** **class** UserController {

//使用@RequestMapping定义该函数的访问路径[http://ip:port/project/login]

@RequestMapping(value="/login")

**public** String login(){

//这里返回的loginsuccess是视图的逻辑地址

**return** "loginsuccess";

}

}

### 创建SpringMVC的配置文件

①：创建Source Folder，命名config,那么该路径也将成为类路径的根路径

②：在config目录下创建SpringMVC的配置文件SpringMVC-Config.xml

<!-- 记得添加上下文约束 -->

<!-- 使用上下文扫描器：base-package包下带spring注解的类将全部管理起来 -->

<context:component-scan base-package=*"com.xiaoka.controller"* />

<!-- 配置基于注解的处理器映射器 -->

<bean class=*"org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"* />

<!-- 配置基于直接的处理器适配器 -->

<bean class=*"org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"* />

<!-- 配置视图解析器 -->

<bean

class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

<property name=*"prefix"* value=*"/WEB-INF/jsp/"*/>

<property name=*"suffix"* value=*".jsp"*/>

</bean>

### 配置前端控制器

在web.xml中配置SpringMVC的前端控制器，加载指定目录下的配置文件。

①：配置SpringMVC的过滤器

②：加载指定目录下的指定文件作为配置文件

<servlet>

<servlet-name>springmvc</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<!-- 指定加载类路径下的 SpringMVC-Config.xml配置文件-->

<param-name>contextConfigLocation</param-name>

<param-value>classpath:SpringMVC-Config.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

### 创建loginsuccess.jsp文件

在WEB-INF/下创建jsp目录

在jsp目录下创建loginsuccess.jsp文件

<body>

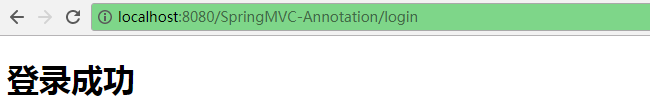
<h1>登录成功</h1>

</body>

### 运行测试

在浏览器输入url

<http://localhost:8080/SpringMVC-Annotation/login>



### 小结

使用注解极大的方便的了开发，提高了开发速度；

注解的粒度更细，它直接定义到方法级别上的访问路径，这使得一个Controller里边可以有多个访问的路径。

# @RequestMapping注解

## 该注解作用于方法上，用于指示该方法用来处理请求动作

### 修改UserController

@Controller

**public** **class** UserController {

// 该函数的访问路径：http://ip:port/project/login

@RequestMapping(value="/login")

**public** String login() {

**return** "loginsuccess";

}

// 该函数的访问路径：http://ip:port/project/register

@RequestMapping(value="/register")

**public** String register() {

**return** "registersuccess";

}

}

### 创建jsp文件

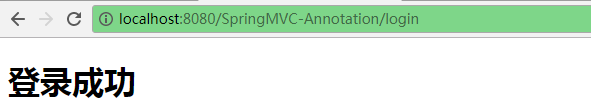
在WEB-INF/jsp/目录下创建registersuccess.jsp文件

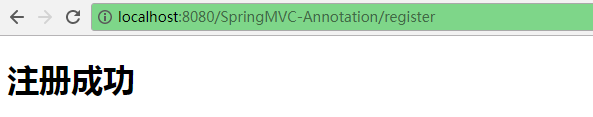
<body>

<h1>注册成功</h1>

</body>

### 运行测试





## 作用类上，Controller中的请求都被映射到类所指示的路径下

### 修改UserController

在UserController类上添加注解@RequestMapping

@Controller

@RequestMapping(value="/user")//当前Controller中所有的请求都必须在前面追加上/user/

**public** **class** UserController {

// 该函数的访问路径：http://ip:port/project/user/login

@RequestMapping(value="/login")

**public** String login() {

**return** "loginsuccess";

}

// 该函数的访问路径：http://ip:port/project/user/register

@RequestMapping(value="/register")

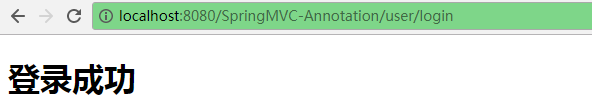
**public** String register() {

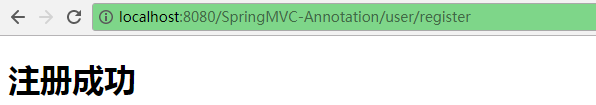
**return** "registersuccess";

}

}

### 运行测试





## 访问方式

@RequestMapping的value属性用于指定访问的路径，它的method属性则用来指定访问的方式

### 指定以GET方式访问

修改UserController的login函数

// 该路径只能通过GET方式访问

@RequestMapping(value="/login",method = RequestMethod.***GET***)

**public** String login() {

**return** "loginsuccess";

}

### 指定以POST方式访问

修改UserController的register函数

// 该路径只能通过POST方式访问

@RequestMapping(value="/register",method = RequestMethod.***POST***)

**public** String register() {

**return** "registersuccess";

}

### 指定同时支持GET和POST访问

修改UserController，在里边创建testGetAndPost()函数

//该路径同时支持GET和POST两种请求方式

@RequestMapping(value="/test",method={RequestMethod.***GET***,RequestMethod.***POST***})

**public** String testGetAndPost(){

**return** "loginsuccess";

}

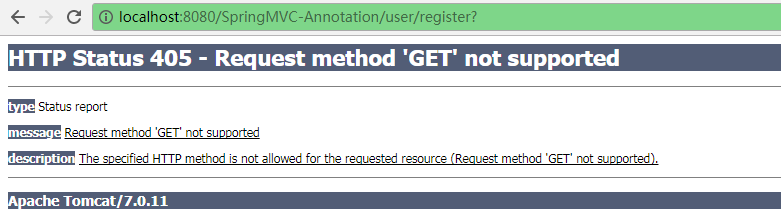
### 运行测试

10.3.1定义login()函数只能通过GET方式访问，那么用POST方式访问将报错



注意：仅当使用表单提交，且指定表单method=POST，才是使用POST访问。

10.3.1定义register()函数只能通过POST方式访问，那么用GET方式访问将报错



# 接受请求参数

SpringMVC接受请求参数遵循一个基本的规律：只要请求参数的名称和Controller中函数的形式参数名称一致，那么就可以自动封装成功。

由于Controller中的函数是完全自定义的，所以形式参数可以声明任何类型，十分的方便。

## 基本类型-Integer

### 修改UserController

修改login函数，在里边声明Integer类型的empno

@RequestMapping(value="/login",method = RequestMethod.***GET***)

**public** String login(Integer empno) {

System.***out***.println(empno);

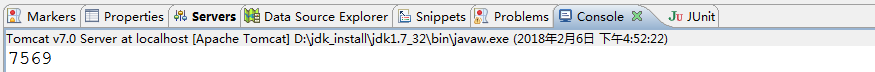
**return** "loginsuccess";

}

### 运行测试

<http://localhost:8080/SpringMVC-Annotation/user/login?empno=7569>

### 控制台日志



## 基本类型-String

### 修改UserController

修改testGetAndPost函数，在里边声明String类型的形式参数eName

@RequestMapping(value="/test",method={RequestMethod.***GET***,RequestMethod.***POST***})

**public** String testGetAndPost(String eName){

System.***out***.println(eName);

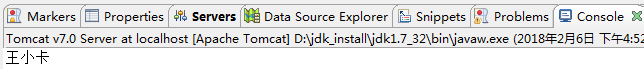
**return** "loginsuccess";

}

### 运行测试

[http://localhost:8080/SpringMVC-Annotation/user/test?eName=](http://localhost:8080/SpringMVC-Annotation/user/test?eName=%E7%8E%8B%E5%B0%8F%E5%8D%A1)王小卡

### 控制台日志



## POJO类型

### 创建JavaBean

**public** **class** User {

**private** Integer id;

**private** String userName;

**private** Date birthDay;

//getter/setter和toString()...略...

}

### 在UserController中添加函数

@RequestMapping(value="/testPojo",method={RequestMethod.***GET***,RequestMethod.***POST***})

**public** String testPojo(User user){

System.***out***.println(user);

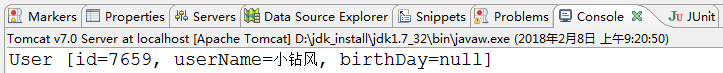
**return** "loginsuccess";

}

### 运行测试

[http://localhost:8080/SpringMVC-Annotation/user/testPojo?id=7659&userName=](http://localhost:8080/SpringMVC-Annotation/user/testPojo?id=7659&userName=%E5%B0%8F%E9%92%BB%E9%A3%8E)小钻风

### 控制台日志



可以看到，凡是参数名称和User对象属性名称一致的，都可以自动封装到User对象中。

## 包装类型

包装类型其实就是对象的嵌套。

### 创建User包装类

**public** **class** DecorationUser {

**private** User user;

}

### 在UserController中添加函数

@RequestMapping(value="/testDecPojo",method=RequestMethod.***POST***)

**public** String testDecPojo(DecorationUser decorationUser){

System.***out***.println(decorationUser);

**return** "loginsuccess";

}

### 创建test.jsp

<form action=*"user/testDecPojo"* method=*"POST"*>

ID:<input type=*"text"* name=*"user.id"*><br>

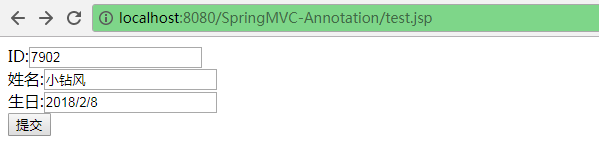
姓名:<input type=*"text"* name=*"user.userName"*><br>

生日:<input type=*"text"* name=*"user.birthDay"*><br>

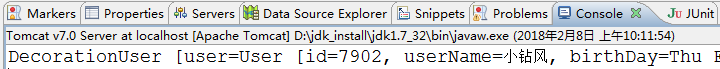
<input type=*"submit"* value=*"提交"*>

</form>

### 运行测试



### 控制台日志



## 数组类型

### 在UserControler中添加函数

@RequestMapping(value="/testArray",method=RequestMethod.***POST***)

**public** String testArray(Integer[] ids){

**for** (Integer integer : ids) {

System.***out***.println(integer);

}

**return** "loginsuccess";

}

### 创建testArray.jsp

<form action=*"user/testArray"* method=*"POST"*>

ID:<input type=*"text"* name=*"ids"*><br>

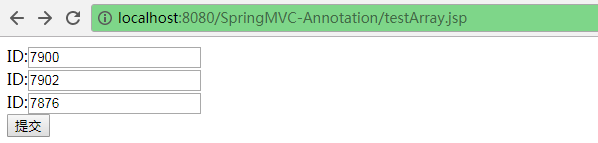
ID:<input type=*"text"* name=*"ids"*><br>

ID:<input type=*"text"* name=*"ids"*><br>

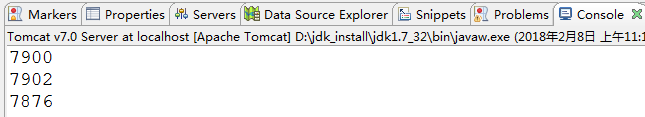
<input type=*"submit"* value=*"提交"*>

</form>

### 运行测试



### 控制台日志



## 集合类型

集合类型无法直接在Controller中接受，必须使用包装类来接受。

### 修改包装类

**public** **class** DecorationUser {

**private** User user;

**private** List<String> list;

**private** List<User> userList;

**private** Map<String, String> map;

//getter/setter和toString()...略...

}

### 在UserController中添加函数

@RequestMapping(value="/testCollection",method=RequestMethod.***POST***)

**public** String testCollection(DecorationUser decorationUser){

System.***out***.println(decorationUser);

**return** "loginsuccess";

}

### 创建testCollection.jsp

<form action=*"user/testCollection"* method=*"POST"*>

姓名1:<input type=*"text"* name=*"list[0]"*><br>

姓名2:<input type=*"text"* name=*"list[1]"*><br>

姓名3：<input type=*"text"* name=*"userList[0].userName"*><br>

姓名5：<input type=*"text"* name=*"userList[1].userName"*><br>

<!-- "username"就是map集合的key，value则是输入的值 -->

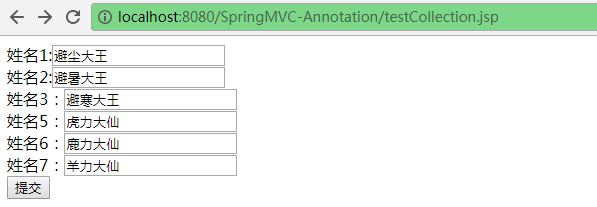
姓名6：<input type=*"text"* name=*"map['username']"*><br>

姓名7：<input type=*"text"* name=*"map['username']"*><br>

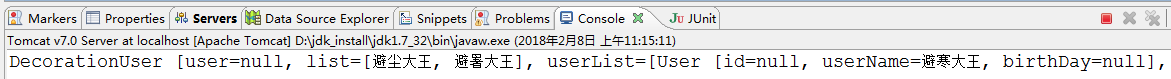
<input type=*"submit"* value=*"提交"*>

</form>

### 运行测试



### 控制台日志



# 页面数据回显

SpringMVC使用Model对象来存储模型数据，在页面直接使用EL表达式就可以获取。

## 在UserController中添加函数

@RequestMapping(value = "/testBackData", method = RequestMethod.***GET***)

**public** String testBackData(Model model) {

ArrayList<User> userList = **new** ArrayList<>();

ArrayList<String> list = **new** ArrayList<>();

**for** (**int** i = 1; i < 6; i++) {

User user = **new** User();

user.setId(i);

user.setUserName("小钻风" + i);

userList.add(user);

list.add("王小卡" + i);

}

model.addAttribute("userList", userList);

model.addAttribute("list", list);

model.addAttribute("name", "小钻风");

**return** "showData";

}

## 创建.jsp页面

在WEB-INF/jsp/下创建showData.jsp

<body>

<font><b>展示集合数据：泛型是User</b></font></br>

${userList[0].userName}</br>

<font><b>展示集合数据：泛型是String</b> </font></br>

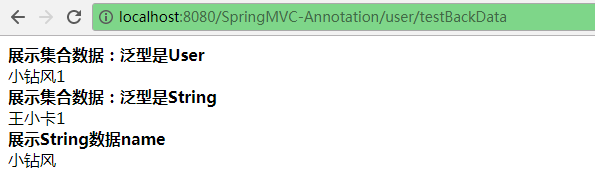
${list[0]}</br>

<font><b>展示String数据name</b></font></br>

${name}

</body>

## 运行测试



# @PathVariable注解

该注解可以非常方便的获取请求URL中的动态参数，它只有一个属性value，类型为String,表示绑定的名称。如果省略则默认绑定同名参数。

使用该注解主要是为了实现Restfull风格设计，让请求中的URL中不再有.do,.action等扩展名，不仅可以提高安全性，而且方便搜索引擎收录。

## 修改UserController

/\*@RequestMapping(value = "/xxx/{x}" 中的{x}表示一个占位符

\*@RequestMapping("/xxx/{x}" 中的{x}要和@PathVariable(value = "y")中的{y}保持一致

\*请求URL【http://ip:port/x/y/z】里边的z则会赋值给x，而x则对应了y,y则是注解了userId

\*最终则是x赋值给userId

\* x,y,userId，完全一致，可以省略@PathVariable中的value属性

\*

\*/

@RequestMapping(value = "/testPathVariable/{id}", method = RequestMethod.***GET***)

**public** String testPathVariable(@PathVariable(value = "id") String userId) {

System.***out***.println(userId);

**return** "loginsuccess";

}

## 修改showData.jsp

<a href="${pageContext.request.contextPath }/user/testPathVariable/${userList[0].id}">删除用户</a>

## 运行测试

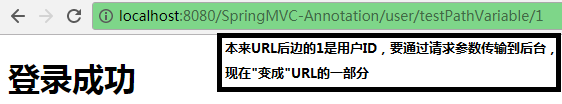
①：访问

[http://localhost:8080/SpringMVC-Annotation/user/testBackData](http://localhost:8080/SpringMVC-Annotation/user/testBackData，)

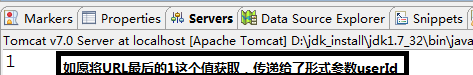
让showData.jsp中具有数据

②：单击“删除用户”超链接

**浏览器**



**控制台日志**



## 小结

SpringMVC使用@PathVariable注解来获取请求URL中的动态参数，这也是Restfull设计风格。

Restfull增加了程序的安全性，它将请求参数隐藏到了URL中

Restfull可以方便搜素引擎收录

# 请求转发和重定向

在SpringMVC中转发请求和响应重定向很简单，只需要在返回的逻辑视图字符串前加入forward或者redirect即可。

注意：当加入了forward或者redirect以后，返回的字符串将不再是逻辑视图，而是Controller的访问路径

## 响应重定向测试

### 修改UserController

①：修改testBackData()函数，在返回视图前拼接redirect。

@RequestMapping(value = "/testBackData", method = RequestMethod.***GET***)

**public** String testBackData(Model model) {

ArrayList<User> userList = **new** ArrayList<>();

ArrayList<String> list = **new** ArrayList<>();

**for** (**int** i = 1; i < 6; i++) {

User user = **new** User();

user.setId(i);

user.setUserName("小钻风" + i);

userList.add(user);

list.add("王小卡" + i);

}

model.addAttribute("userList", userList);

model.addAttribute("list", list);

model.addAttribute("name", "小钻风");

//这里返回的不再是逻辑视图，而是一个URL

**return** "redirect:showData";

}

②：加入函数，用当前函数返回逻辑视图

@RequestMapping(value = "/showData")

**public** String testBackDataController(Model model) {

System.***out***.println("------------");

//这里返回则是逻辑视图

**return** "showData";

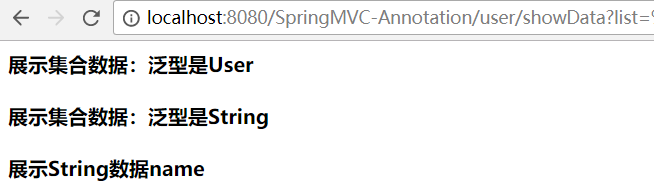
}

### 运行测试

1. 在地址栏输入，并回车



**运行结果**



### 小结

可以看到，地址栏地址已经更改，页面上也没有数据，这是因为使用了重定向的缘故。

## 请求转发测试

### 修改UserController

修改testBackData()函数：所有逻辑不变，只是在返回视图前拼接的是forward

@RequestMapping(value = "/testBackData", method = RequestMethod.***GET***)

**public** String testBackData(Model model) {

ArrayList<User> userList = **new** ArrayList<>();

ArrayList<String> list = **new** ArrayList<>();

**for** (**int** i = 1; i < 6; i++) {

User user = **new** User();

user.setId(i);

user.setUserName("小钻风" + i);

userList.add(user);

list.add("王小卡" + i);

}

model.addAttribute("userList", userList);

model.addAttribute("list", list);

model.addAttribute("name", "小钻风");

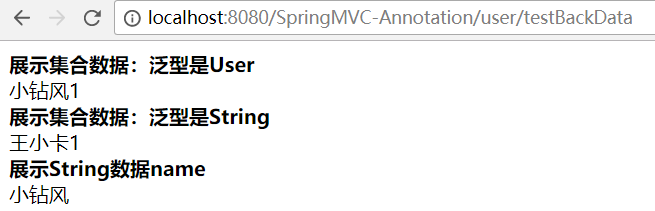
// 这里返回的不再是逻辑视图，而是一个URL

**return** "forward:showData";

}

### 运行测试

在地址栏输入请求URL,并回车



### 小结

使用了请求转发以后，数据依然可以得到，地址栏地址保持不变。

## 路径小结

注意：在返回的路径中，如果用斜杠开头，则是使用绝对路径，否则则是使用相对路径。无论是请求转发还是重定向，推荐使用绝对路径。

绝对路径

返回到当前Controller

Return “forward:/当前Controller根路径/函数访问路径”

返回到其它Controller

Return”forward:/目标Controller根路径/函数访问路径”

相对路径

返回到当前Controller

Return “forward:函数访问路径”

返回到其它Controller

在实际开发中，如果要访问的是不是当前Controller，那么不建议使用相对路径。

# @RequestParam

该注解用于参数绑定，它可以将指定的请求参数赋值给方法中的指定形参。下表是@RequestParam注解支持的属性。

|  |  |  |  |
| --- | --- | --- | --- |
| 属性 | 类型 | 是否必要 | 说明 |
| name | String | 否 | 指定请求头绑定的名称 |
| value | String | 否 | 请求参数的name名称 |
| required | boolean | 否 | 指示参数是否必须绑定，默认值为ture |
| defaultValue | String | 否 | 指定参数的默认值 |

## 修改UserController

在UserController中加入测试函数

//请求参数name的值会赋值给形式参数userName

@RequestMapping("testReuqestParam")

**public** String testReuqestParam(

@RequestParam(defaultValue = "小钻风", value = "name", required = **true**) String userName) {

System.***out***.println(userName);

**return** "loginsuccess";

}

## 运行测试

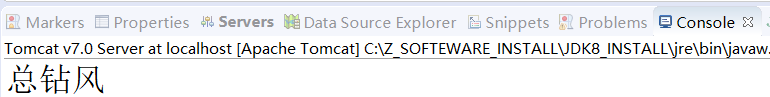
### 传入指定参数

在浏览器地址栏输入请求URL，传入指定必须的name参数

<http://localhost:8080/SpringMVC-Annotation/user/testReuqestParam?name>=总钻风



控制台日志



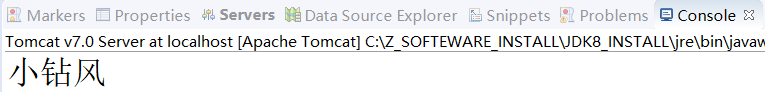
当传入了指定的参数以后，那么Controller里边就会收到该值

### 没有传入指定参数

在浏览器地址栏输入请求URL，没有传入指定必须的name参数

<http://localhost:8080/SpringMVC-Annotation/user/testReuqestParam?namexxx>=总钻风

控制台日志



没有传入指定的参数，那么Controller里边就会使用默认值

### 报错400

当指定required=true【该属性默认为true】,且不指定defaultValue，这个时候如果没有传入指定的请求参数，则会报错400。

#### 修改Controller

将testReuqestParam()函数只保留value属性，这个时候意味着请求参数必须有name,且没有了默认值

@RequestMapping("testReuqestParam")

**public** String testReuqestParam(

@RequestParam(value = "name") String userName) {

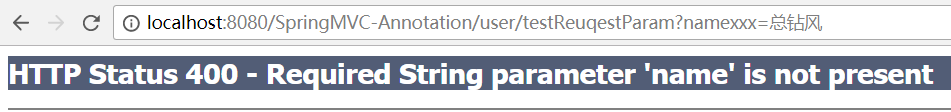
System.***out***.println(userName);

**return** "success";

}

#### 运行测试

在地址栏输入请求URL，刻意不传入所必须的请求参数name



# 注解驱动

使用注解驱动，可以简化SpringMVC的配置。

## 修改配置文件

<context:component-scan base-package=*"com.xiaoka.controller"* />

<!-- 开启注解驱动 -->

<!-- 注解驱动将自动装配基于注解的处理器映射器RequestMappingHandlerMapping

和处理器适配器RequestMappingHandlerAdapter

-->

<!-- 注解驱动也将自动开启JSON支持 -->

<!-- 记得添加mvc约束 -->

<mvc:annotation-driven />

<bean

class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

<property name=*"prefix"* value=*"/WEB-INF/jsp/"* />

<property name=*"suffix"* value=*".jsp"* />

</bean>

## 运行测试

略。【所有的基于注解的Controller都将可以被正常访问。】

# SpringMVC对JSON的支持

SpringMVC默认使用第三方开源库Jackson来处理json数据.所以当要接受json或者返回json数据的时候,需要下载第三方jar包Jackson。

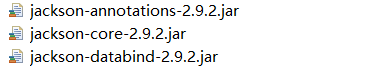
注解驱动默认开启了JSON支持

## @RequestBody

该注解用于接收页面发送过来的json数据，并将json格式的数据自动封装到JavaBean中。

**需求：页面发送json格式数据，后台使用JavaBean接受**

### 下载jackson的jar包导入到当前项目



### 修改配置文件



### 修改UserController

在UserController中添加接受json数据的函数

@RequestMapping("testRequestJson")

/\*@RequestBody：该注解用于将页面发送过来的json数据封装到User中\*/

**public** **void** testRequestJson(@RequestBody User user, HttpServletResponse response) **throws** IOException {

System.***out***.println(user);

response.getWriter().write("{msg:'success'}");

}

### 创建jsp页面，发送json数据

在WebContent下创建发送json数据的jsp页面

①：创建sendJson.jsp

②：导入Jquery的js文件

<script src=*"*${pageContext.request.contextPath}*/js/jquery-2.1.0.js"*type=*"text/javascript"* charset=*"utf-8"*></script>

<script type=*"text/javascript"*>

**var** url = '${pageContext.request.contextPath}/user/testRequestJson';

**var** jsonData = {'id' : 7965,'userName' : '小钻风'};

**var** content = JSON.stringify(jsonData);

$.ajax({

type:'POST',

url : url,

data : content,

contentType : 'application/json',

success : **function**(data) {

alert(data);

},

error : **function**() {

alert('error');

}

})

</script>

</head>

<body>

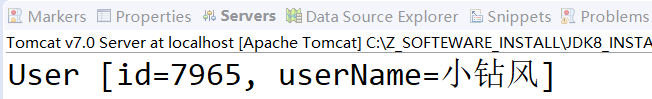
</body>

### 运行测试

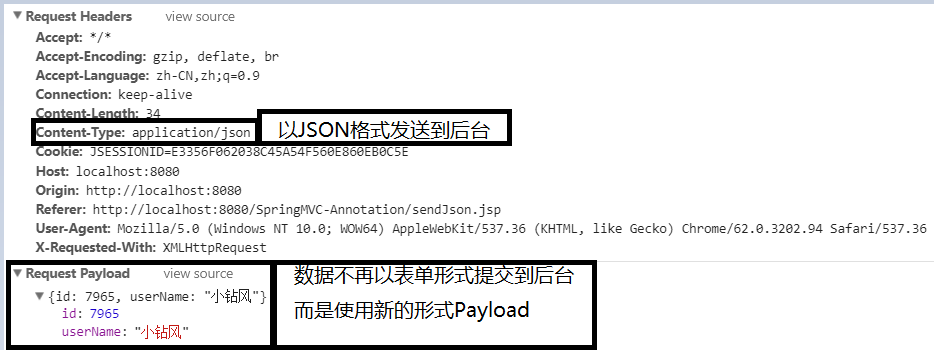
直接使用浏览器访问sendJson.jsp页面，即可直接发送json数据到Controller。



控制台日志



### 分析



## @ResponseBody

该注解用于将后台返回的数据转换成json对象。注意：页面上拿到的数据已经是json对象。

### 创建JSP页面

在WebContent下创建接受json数据的页面receiveJson.jsp

<script src=*"*${pageContext.request.contextPath}*/js/jquery-2.1.0.js"*type=*"text/javascript"* charset=*"utf-8"*></script>

<script type=*"text/javascript"*>

**var** url = '${pageContext.request.contextPath}/user/testReceiveJson';

$.ajax({

type:'POST',

url : url,

success : **function**(data) {

alert(data.userName);

},

error : **function**() {

alert('error');

}

})

</script>

### 修改Controller

在UserController中添加函数，返回java对象。

@RequestMapping("/testReceiveJson")

/\*@ResponseBody：该注解用于将User转换成json返回到页面\*/

**public** @ResponseBody User testReceiveJson() **throws** IOException {

User user = **new** User();

user.setId(7956);

user.setUserName("小钻风");

**return** user;

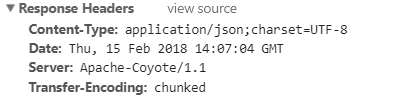
}

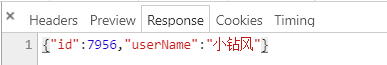
### 运行测试

直接在浏览器访问receiveJson.jsp，将会直接访问Controller，即直接拿到返回的json对象数据



### 分析



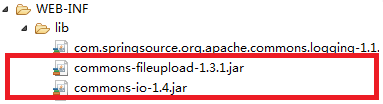


# SpringMVC上传下载

## SpringMVC上传

### 添加jar包

SpringMVC依然使用了Apache的Commons FileUpload组件,因此首先要把Apache Commons FileUpload.jar添加到项目路径下。



### 创建上传的jsp文件

在WebContent下创建用于上传的jsp文件upload.jsp

<!-- 上传文件的表达必须使用POST提交，设置enctype值为二进制表单数据 -->

<form action=*"upload"* enctype=*"multipart/form-data"* method=*"POST"*>

文件描述<input type=*"text"* name=*"description"*><br>

请选择文件<input type=*"file"* name=*"fileName"*><br>

<input type=*"submit"* value=*"上传"*>

</form>

### 创建上传文件的Controller

@Controller

**public** **class** UploadController {

@RequestMapping("upload")

**public** String upLoad(String description, MultipartFile fileName) **throws** Exception{

System.***out***.println(description);

//获得上传文件的名称

String originalFilename = fileName.getOriginalFilename();

System.***out***.println(originalFilename);

File file = **new** File(originalFilename);

//将上传的文件保存到指定路径

fileName.transferTo(file);

**return** "loginsuccess";

}

### 修改配置文件

<!-- 配置MultipartResolver，这样才可以使用SpringMVC的上传功能 -->

<bean id=*"multipartResolver"*

class=*"org.springframework.web.multipart.commons.CommonsMultipartResolver"*>

<!-- 上传文件大小限制，单位为字节 -->

<property name=*"maxUploadSize"*>

<value>5120000</value>

</property>

<!-- 请求的编码格式，必须和jsp的pageEncoding值一致 -->

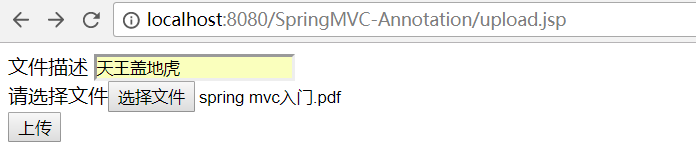
<property name=*"defaultEncoding"*>

<value>UTF-8</value>

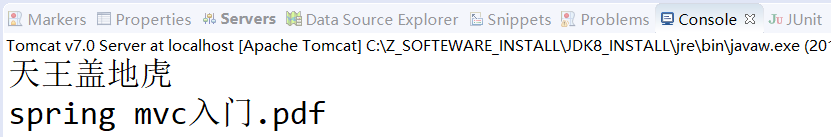
</property>

</bean>

### 运行测试



控制台日志



可以看到，文件已经上传到指定目录。

### 小结

本例使用最简单的代码实现了SpringMVC的文件上传。而文件上传中有诸多问题亟待解决留成练习作业。

①：文件上传的路径问题

②：文件名称重复问题

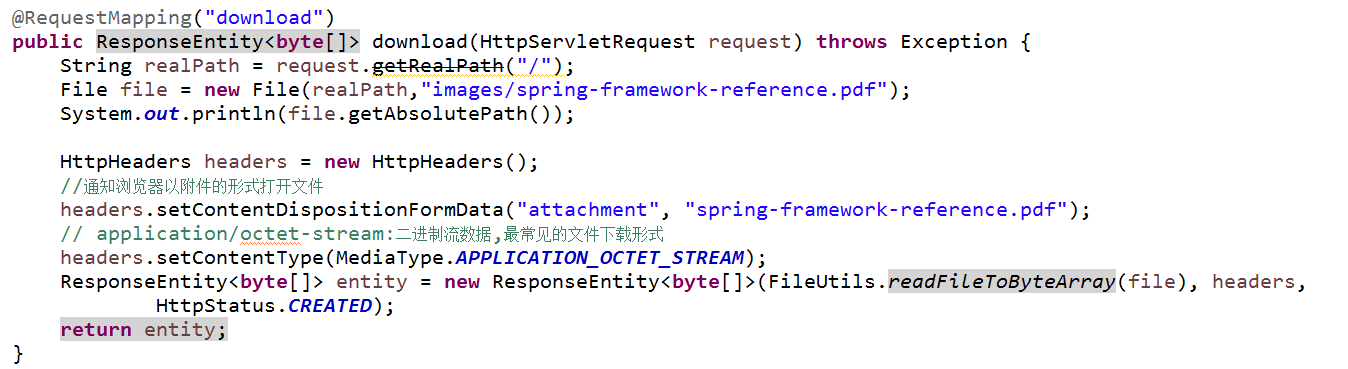
③：如何使用Ajax上传文件

④：如何上传到远程服务器

⑤：使用Ajax上传图片，图片要求回显

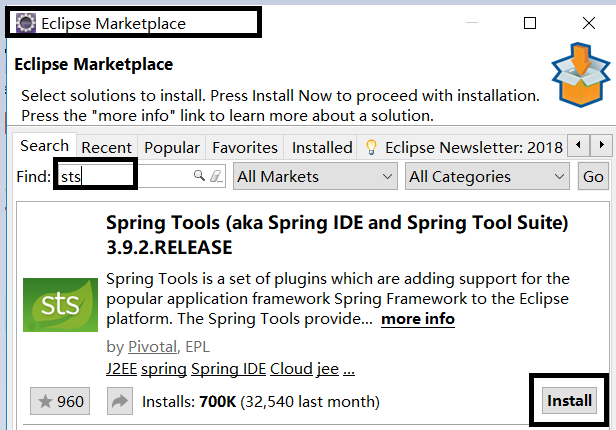
## SpringMVC下载

使用SpringMVC进行下载,则很方便,只需要使用ResponseEntity即可。



# 附录

## Eclipse安装插件



## Jackson简介

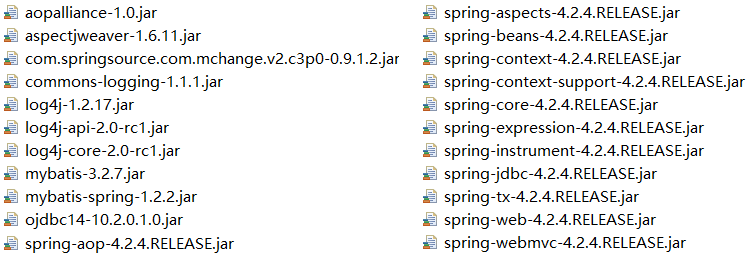
<http://blog.csdn.net/java_huashan/article/details/46375857>

## SSM整合

### 创建工程

创建Web工程，并导入相应的jar包。所需jar包：

Spring,SpringMVC,Mybatis,mybatis-spring,数据库驱动包,C3P0数据库连接池等。



### 创建配置文件

创建资源目录config,依次创建如下配置文件在config中。

#### 创建log4j.properties文件

#### 创建db.properties文件

db.driverClass=oracle.jdbc.OracleDriver

db.url=jdbc:oracle:thin:@127.0.0.1:1521:orcl

db.user=scott

db.password=tiger

#### 创建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"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:jdbc=*"http://www.springframework.org/schema/jdbc"*

xmlns:tx=*"http://www.springframework.org/schema/tx"*

xmlns:mybatis-spring=*"http://mybatis.org/schema/mybatis-spring"*

xsi:schemaLocation=*"http://www.springframework.org/schema/jdbc*

*http://www.springframework.org/schema/jdbc/spring-jdbc-4.2.xsd*

*http://mybatis.org/schema/mybatis-spring*

*http://mybatis.org/schema/mybatis-spring-1.2.xsd*

*http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-4.2.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-4.2.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx.xsd"*>

<!-- 记载db.properties -->

<context:property-placeholder location=*"classpath:db.properties"* />

<!-- 配置数据源 -->

<bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>

<property name=*"driverClass"* value=*"${db.driverClass}"* />

<property name=*"jdbcUrl"* value=*"${db.url}"* />

<property name=*"user"* value=*"${db.user}"* />

<property name=*"password"* value=*"${db.password}"* />

</bean>

<!-- 配置SqlSessionFactory -->

<bean id=*"sqlSessionFactory"* class=*"org.mybatis.spring.SqlSessionFactoryBean"*>

<!-- 设置数据源 -->

<property name=*"dataSource"* ref=*"dataSource"* />

<property name=*"configLocation"* value=*"classpath:sqlMapConfig.xml"* />

</bean>

<!-- 配置Mapper接口代理 -->

<bean class=*"org.mybatis.spring.mapper.MapperScannerConfigurer"*>

<property name=*"basePackage"* value=*"com.xiaoka.mapper"* />

</bean>

<!-- 事务 -->

<bean id=*"transactionManager"*

class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>

<property name=*"dataSource"* ref=*"dataSource"* />

</bean>

<!-- 通知：切面类 -->

<tx:advice id=*"txAdvice"* transaction-manager=*"transactionManager"*>

<tx:attributes>

<tx:method name=*"save\*"* propagation=*"REQUIRED"* />

<tx:method name=*"delete\*"* propagation=*"REQUIRED"* />

<tx:method name=*"update\*"* propagation=*"REQUIRED"* />

<tx:method name=*"insert\*"* propagation=*"REQUIRED"* />

</tx:attributes>

</tx:advice>

<!-- 切入点 -->

<aop:config>

<aop:pointcut expression=*"execution(\* com.xiaoka.service.\*.\*(..))"*

id=*"txPoint"* />

<aop:advisor advice-ref=*"txAdvice"* pointcut-ref=*"txPoint"* />

</aop:config>

</beans>

#### 创建springmvc的配置文件springmvc-config.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"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:mvc=*"http://www.springframework.org/schema/mvc"*

xsi:schemaLocation=*"http://www.springframework.org/schema/mvc*

*http://www.springframework.org/schema/mvc/spring-mvc-4.2.xsd*

*http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-4.2.xsd"*>

<!-- 加载spring配置文件 -->

<import resource=*"applicationContext.xml"*/>

<!-- 开启全局扫描 -->

<context:component-scan base-package=*"com.xiaoka"*></context:component-scan>

<!-- 注解驱动 -->

<mvc:annotation-driven />

<!-- 视图解析器 -->

<bean

class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

<property name=*"suffix"* value=*".jsp"* />

<property name=*"prefix"* value=*"/WEB-INF/jsp/"* />

</bean>

</beans>

#### 创建Mybatis的配置文件sqlMapConfig.xml

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE configuration

PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

<settings>

<setting name=*"lazyLoadingEnabled"* value=*"true"* />

<setting name=*"aggressiveLazyLoading"* value=*"false"* />

<setting name=*"cacheEnabled"* value=*"true"*/>

</settings>

<typeAliases>

<package name=*"com.xiaoka.bean"* />

</typeAliases>

</configuration>

### 加载配置文件

在web.xml中加载springmvc的配置文件

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns=*"http://java.sun.com/xml/ns/javaee"*

xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"*

id=*"WebApp\_ID"* version=*"2.5"*>

<filter>

<filter-name>encodingFilter</filter-name> <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

<init-param>

<param-name>encoding</param-name>

<param-value>UTF-8</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>encodingFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

<servlet>

<servlet-name>springmvc</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:springmvc-config.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>springmvc</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

### 创建JavaBean

@Component

**public** **class** Emp {

**private** Short empno;

**private** String ename;

**private** String job;

**private** Short mgr;

**private** Date hiredate;

**private** BigDecimal sal;

**private** BigDecimal comm;

**private** Short deptno;

}

### 创建Mapper接口

**public** **interface** EmpMapper {

Emp selectByPrimaryKey(Short empno);

List<Emp> selectByExample(EmpExample example);

**int** deleteByPrimaryKey(Short empno);

}

### 创建Mapper配置文件

### 创建UserService.java

@Service

**public** **class** UserService {

@Resource

**private** EmpMapper empMapper;

**public** Emp queryUser(**short** empno){

Emp emp = empMapper.selectByPrimaryKey(empno);

**return** emp;

}

**public** List<Emp> queryEmpLikeName(){

EmpExample example = **new** EmpExample();

Criteria criteria = example.createCriteria();

criteria.andEnameLike("%O%");

List<Emp> list = empMapper.selectByExample(example);

**return** list;

}

**public** **void** deleteEmpByEmpNo(**short** empno){

empMapper.deleteByPrimaryKey(empno);

**int** k = 10/0;

}

}

### 创建UserController.java

@Controller

**public** **class** UserController {

@Resource

**private** UserService userService;

@RequestMapping("/login")

**public** ModelAndView login(**short** empno){

Emp user = userService.queryUser(empno);

List<Emp> list = userService.queryEmpLikeName();

ModelAndView modelAndView = **new** ModelAndView();

modelAndView.addObject("emp", user);

modelAndView.addObject("list", list);

modelAndView.setViewName("success");

**return** modelAndView;

}

@RequestMapping("deleteEmp")

**public** **void** delete(**short** empno){

userService.deleteEmpByEmpNo(empno);

System.***out***.println("================="+empno);

}

}

### 创建success.jsp

在WEB-INF/下创建jsp目录，在该目录下创建success.jsp文件

<body>

${list}

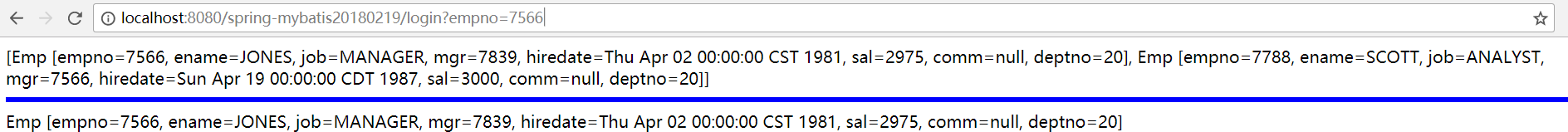
<hr size=*"5"* color=*"blue"*>

${emp}

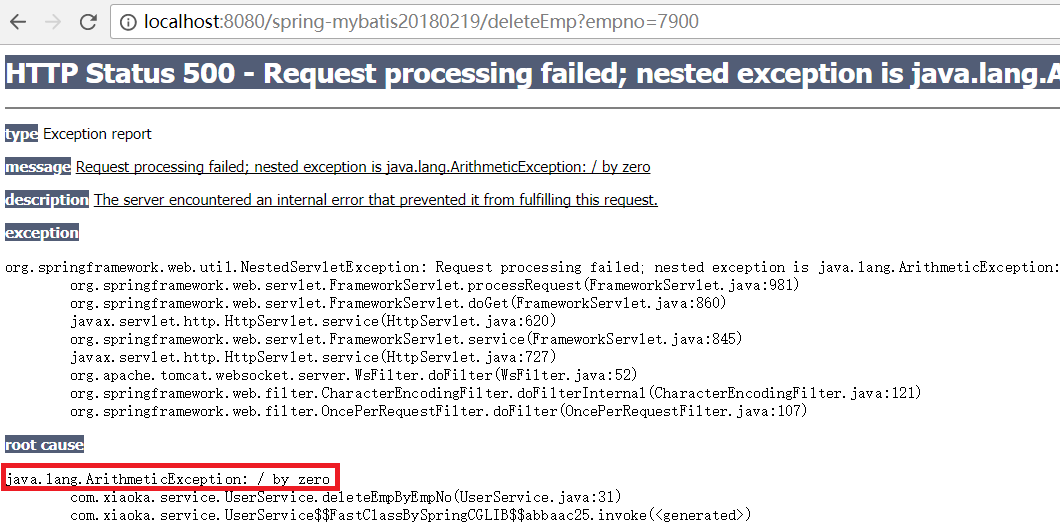
</body>

### 运行测试

对查询的测试



对事务的测试



查询数据库，发现7900号员工依然存在，说明事务生效。