# Spring

Spring教程：<http://www.yiibai.com/spring/load-multiple-spring-bean-configuration-file.html>

ApplicationContext有三个实现类：

ClassPathXmlApplicationContext：从classpath检索配置文件并装载，创建bean实例。

FileSystemXmlApplicationContext：从classpath以外的地方检索配置文件并装载，创建实例。

XmlWebApplicationContext：可以在web.xml中配置

## 外部配置文件,properties引用

<http://www.cnblogs.com/leiOOlei/archive/2013/12/25/3491042.html>

### 外部Spring bean的引用

如果bean在不同的xml文件中，需要先将其所在的xml文件引入（import）;

<**import** resource="spring-bean.xml"/>  
<**bean** id="userDao" class="com.duan.springweb.dao.UserDaoImpl">  
 <**property** name="user" ref="user2"/>  
</**bean**>

在同一个文件中时，直接引用即可，也可放在独立的<ref local=””/>标签中（不作为<property/>标签的属性）。

注意：可以不显示的import，直接引用，但这需要在装载配置文件时装载所需的所有配置文件，即：

String[] cs = {  
 "/spring/spring-mvc.xml",  
 "/spring/spring-bean.xml",  
};  
ApplicationContext context = new ClassPathXmlApplicationContext(cs);

或者在一个配置文件中<import/>用到的所有配置文件，创建ApplicationContext只传入那个即可。

### Properties引用

<http://blog.csdn.net/eson_15/article/details/51365707>

1. Xml文件中引用

<context:property-placeholder location=””/>

如：

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

或者<bean/>标签

如：

<!-- 用于读取properties文件，此处将数据库连接的用户名密码等存在db-config.properties文件中 -->  
<**bean** id="propertyConfigurer"  
 class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">  
 <**property** name="locations">  
 <**list**>  
 <**value**>db-config.properties</**value**>  
 </**list**>  
 </**property**>  
</**bean**>

## Spring bean三种写法

1. 普通方法

<**bean** id="user" class="com.duan.springweb.entity.User">  
 <**property** name="name">  
 <**value**>"张二"</**value**>  
 </**property**>  
 <**property** name="password">  
 <**value**>"张二aabc"</**value**>  
 </**property**>  
 <**property** name="state">  
 <**value**>0</**value**>  
 </**property**>  
 <**property** name="age">  
 <**value**>23</**value**>  
 </**property**>  
</**bean**>

1. 快捷方法

<**bean** id="user" class="com.duan.springweb.entity.User">  
 <**property** name="name" value="张二"/>  
 <**property** name="password" value="张二aabc"/>  
 <**property** name="state" value="0"/>  
 <**property** name="age" value="23"/>  
</**bean**>

1. “p”方法
2. <**bean** id="user" class="com.duan.springweb.entity.User"  
    p:name="张二"  
    p:password="张二aabc"  
    p:state="0"  
    p:age="23"/>

# Spring MVC

Spring文档：<https://docs.spring.io/spring/docs/>

SpringMVC中文教程：<http://www.yiibai.com/spring_mvc/>

第二章 Spring MVC入门 —— 跟开涛学SpringMVC：

<http://jinnianshilongnian.iteye.com/blog/1594806>

## Idea配置

<http://blog.csdn.net/zy0528smile/article/details/52922378>

<http://www.cnblogs.com/Sinte-Beuve/p/5730553.html>

## Spring请求处理流程和架构

1. 关键对象：

前端控制器：DispatcherServlet

请求-处理器映射：HandlerMapping

处理器适配器：HandlerAdapter

视图解析器：ViewResolve

1. 请求处理流程





流程：用户发送请求 ——> 前端控制器（DispatcherServlet）—*根据请求信息选择页面控制器并委托处理*—> 页面控制器/处理器 —*将请求参数包装为****命令对象****，进行验证，传递给****业务对象****处理（获得****模型数据****）*—> 页面控制器/处理器 —*直接返回ModelAndView*—>前端控制器 —*根据回传数据装载页面,渲染****视图***—> 用户收到回馈

1. 架构



DispatcherServlet的核心代码在doDispatch方法中，doDispatch获取页面处理器，然后根据页面处理器获取对应的HandlerAdapter（以支持多种类型的请求），最后由HandlerAdapter调用页面处理器的方法。

HandlerMapping：找到保存url请求和处理函数间的映射关系。（如RequestMappingHandlerMapping，扫描当前请求和被@ReauestMapping注解的Controller中的方法，将方法和url关联）

HandlerAdapter：处理请求，适配器模式。（如RequestMappingHandlerAdapter，根据HandlerMapping找到对应的方法，并调用）

1. 解析用户请求(**图中第2点**) **——>前端控制器**收到请求后交给其它**解析器处理**，作为统一访问点，进行全局流程控制。通过HandlerMapping找到url和method间的映射关系（利于添加新的映射关系），并包装为Handler**Execution**Chain、Handler以及多个HandlerInterceptor。
2. 处理用户请求(**图中第3点**)**——>**前端控制器成功解析用户请求，将Handler交给**处理适配器**（HandlerAdapter），适配器将根据HandlerMapping获得的映射结果调用**页面处理器**（利于支持多种类型的处理器）的**相应方法**（**图中第4点**），完成处理；并返回一个ModelAndView对象（包含处理结果、逻辑视图名）。
3. 响应用户请求（**图中第5/6点**）**——>**ViewResolver将把逻辑视图解析为具体的View； **——>** View将传入的Model数据渲染到视图上（这里包装后的Model实际上是一个Map（HashMap）数据结构）。
4. 将控制权交回给DispatcherServlet，流程结束。

上述流程的1~4在DispatcherServlet的doDispatch中完成。5、6调用processDispatchResult7方法中完成。

## SpringMVC使用步骤

1. 在web.xml中注册DispatcherServlet并指定拦截所有请求（或指定请求）；
2. 指定配置文件：不指定的话默认加载“(servlet-name)-servlet.xml”，如：

…<servlet-name>MySpring<servlet-name/>，则会寻找MySpring-servlet.xml的配置文件，找不到就会出错（file doesn’t exists）。

可按如下方式配置：

<!--前端控制器DispatcherServlet的配置-->  
<**servlet**>  
 <**servlet-name**>springMVC</**servlet-name**>  
 <**servlet-class**>org.springframework.web.servlet.DispatcherServlet</**servlet-class**>  
 <**init-param**>  
 <!--Sources标注的文件夹下需要新建一个spring文件夹-->  
 <**param-name**>contextConfigLocation</**param-name**>  
 <**param-value**>classpath:spring/spring-mvc.xml</**param-value**>  
 </**init-param**>  
 <**load-on-startup**>1</**load-on-startup**>  
 <**async-supported**>true</**async-supported**>  
</**servlet**>  
  
<**servlet-mapping**>  
 <**servlet-name**>springMVC</**servlet-name**>  
 <**url-pattern**>/</**url-pattern**>  
</**servlet-mapping**>

1. 在spring-mvc.xml中配置视图处理器和其它杂项：

选择注解配置或xml配置；

视图处理器配置于<bean/>标签中；

配置自动扫描位置；

配置静态资源映射；

etc.

1. 开发处理器（Handler）/页面控制器

页面处理器有两种实现方式：参看[Spring MVC注解方式和实现接口方式实现页面控制器的不同](#_Spring_MVC注解方式和实现接口方式实现页面控制器的不同)

接口方式请求流程示例：



1. Java代码中引用bena
2. String[] cs = {  
    "/spring/spring-mvc.xml",  
   // "/spring/spring-bean.xml",  
    };  
    ApplicationContext context = new ClassPathXmlApplicationContext(cs);  
    UserService userService = (UserService) context.getBean("userService");  
    UserDao dao = userService.getUserDao();  
    *p*.accept(dao.queryUserById(2));

关键对象为ApplicationContext。

## Spring MVC注解方式和实现接口方式实现页面控制器的不同

参考文章：<http://blog.csdn.net/eson_15/article/details/51699103>

1. 注解方式（spring2.5之后支持）：

注解方式在配置文件（spring-mvc.xml）要进行如下声明：

<!--注解方式的映射器-->  
<**bean** class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"/>  
<!--注解方式的适配器-->  
<**bean** class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"/>

这种方式需将**页面处理器**对应的java类用@Controller注解，并在**具体方法**上用@ReuestMapping注解以指明该动作的访问url。

方法可返回ModelAndView对象将目标视图和数据包装其中，或返回String类型，仅仅指明目标视图名称。

使用注解方式时要**注意**：注解适配器和注解映射器必须配对使用

在SpringMVC中可以更简单的在配置文件中声明，<mvc:annotation-driven/>，此时注解适配器和映射器可以省略。

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

1. 实现接口方式

1 其中适配器有两种（可以同时使用），对应两种不同的接口实现；

<!--实现接口方式的适配器（HandlerAdapter） 表示所有实现了org.springframework.web.servlet.mvc.Controller-->  
<**bean** class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"/>  
<!--实现接口方式的适配器（HandlerAdapter） 表示所有实现了org.springframework.web.HttpRequestHandler-->  
<**bean** class="org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter"/>

接口方式要求一个页面控制器**只能有一个动作**（处理器），因为实现接口，其动作只能在那个方法中实现。（区别于注解方式的：一个页面控制器多个处理器）

2 映射器也有两种，拥有不同的灵活性：

* url和bean名字（bean将名字和页面控制器对应）对应，

<!--实现接口方式的映射器（HandlerMapping） 无需配置，springmvc可以默认启动-->  
<!-- HandlerMapping 表示将请求的URL和Bean名字（name）映射，如URL为 “上下文/hello”，则Spring配置文件必须有一个名字为“/hello”的Bean，上下文默认忽略。-->  
<**bean** class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

需额外配置<bean name=”” class=””/>将bean的名字和页面控制器对应，访问时就可通过name访问控制器。

* 简单映射器（指定url和页面控制器的对应关系）

<**bean** id="item1" class="com.duan.springweb.controller.Item1Controller"/>  
<**bean** id="item2" class="com.duan.springweb.controller.Item2Controller"/>  
<**bean** class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">  
 <**property** name="mappings">  
 <**props**>  
 <**prop** key="/item\_1">item1</**prop**>  
 <**prop** key="/item\_2">item2</**prop**>  
 </**props**>  
 </**property**>  
</**bean**>

将每一个页面控制器声明为bean，在SimpleUrlHandlerMapping中将访问url和bean对应；

两个映射器也可以一起使用，互不冲突，匹配到哪个就用哪个。

## <context:component-scan/> 自动扫描页面控制器

如果页面控制器很多的话，就需要配置很多的bean，因此spring-mvc提供了扫描包的方式进行自动配置。

<!--自动扫描该包，使SpringMVC认为包下用了@controller 注解的类是控制器 -->  
<**context:component-scan** base-package="com.duan.springweb.controller">  
 <**context:include-filter** type="annotation" expression="org.springframework.stereotype.Controller"/>  
</**context:component-scan**>

这样SpringMVC就能自动扫描@Controller注解的类（或者实现接口的类），该类就被认为是页面控制器，类中被@RequestMapping注解的方法则被认为具体的处理器（动作）；

## 视图名称的解析

<!-- 对模型视图名称的解析，即在模型视图名称添加前后缀(如果最后一个还是表示文件夹,则最后的斜杠不要漏了) 使用JSP  
ViewResolver 用于支持Servlet、JSP视图解析-->  
<**bean** id="defaultViewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">  
 <**property** name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>  
 <**property** name="prefix" value="/WEB-INF/views/"/><!--设置JSP文件的目录位置-->  
 <**property** name="suffix" value=".jsp"/>  
</**bean**>

在页面控制器的相应处理方法中直接返回页面名称，或装入ModelAndView中返回，对视图名称的解析将通过上面的方式。（也可以配置其它的方式）

## SpringMVC提供的乱码解决过滤器

<!--表示规定字符编码为UTF-8-->  
<**filter**>  
 <**filter-name**>CharacterEncodingFilter</**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**>CharacterEncodingFilter</**filter-name**>  
 <**url-pattern**>/\*</**url-pattern**>  
</**filter-mapping**>

## SpringMVC主要注解

<http://jinnianshilongnian.iteye.com/blog/1594806>

## Spring MVC 对静态文件的引用

<http://www.cnblogs.com/dflmg/p/6393416.html>

如果在web.xml中配置DispatcherServlet的url-pattern为“/”，那么spring将捕获所有请求，这会使web应用在请求一些本地的静态资源时被spring以相同的方式处理，即在spring-mvc.xml中寻找其页面控制器，找不到就会产生404错误。有三种方法可以解决这种情况：

### 方法一：采用<mvc:default-servlet-handler/>

在springMVC-servlet.xml中配置<mvc:default-servlet-handler />后，会在Spring MVC上下文中定义一个org.springframework.web.servlet.resource.**DefaultServletHttpRequestHandler**，它会像一个检查员，对进入DispatcherServlet的URL进行筛查，如果发现是静态资源的请求，就将该请求转由Web应用服务器默认的Servlet处理，如果不是静态资源的请求，才由DispatcherServlet继续处理。

### 方法二：采用<mvc:resources/>

<mvc:resources location="/,classpath:/META-INF/publicResources/" mapping="/resources/\*\*"/>

由Spring MVC框架自己处理静态资源，并添加一些有用的附加功能。首先，<mvc:resources />允许静态资源放在任何地方，其次，<mvc:resources />依据当前著名的Page Speed、YSlow等浏览器优化原则对静态资源提供优化。

<!--静态资源映射-->  
<!--本项目把静态资源放在了WEB-INF的statics目录下，资源映射如下-->  
<**mvc:resources** mapping="/css/\*\*" location="/WEB-INF/statics/css/"/>  
<**mvc:resources** mapping="/js/\*\*" location="/WEB-INF/statics/js/"/>  
<**mvc:resources** mapping="/image/\*\*" location="/WEB-INF/statics/image/"/>

### 方法三：激活Tomcat的default servlet来处理静态文件

类似<mvc:default-servlet-handler/>方式。

* **<servlet-mapping>**
* **<servlet-name>**default**</servlet-name>**
* **<url-pattern>**\*.jpg**</url-pattern>**
* **</servlet-mapping>**
* **<servlet-mapping>**
* **<servlet-name>**default**</servlet-name>**
* **<url-pattern>**\*.js**</url-pattern>**
* **</servlet-mapping>**

这种方式要写在DispatcherServlet前面，让defaultServlet先拦截，请求就不会进入spring。

# Spring AOP

<http://blog.csdn.net/luoshenfu001/article/details/5816408/>

用注解配置Spring bean

1. Dao层

@Repository

1. 服务层

@Service

1. 控制层（页面控制器）

@Controller

@Scope：指明对象创建后的作用范围。

@Resource：在属性的setter方法上加注解指定注入的对象来源：@Resource(name = "noteDaoImpl2")，这些来源会在xml文件的包扫描（或定义的bean）中寻找。

也可以只使用@Component来指定对象的所在层，Spring会自动归类，但应该首选更具体的注解来指明组件所属层。

## AOP（Aspet Oriented Programming）

<http://blog.csdn.net/luoshenfu001/article/details/5816408/>

aop，spring aop，aspectj区别与联系<http://blog.csdn.net/pingnanlee/article/details/38845955>

AOP可以处理一些具有横切性质的系统服务，如**事务处理、安全检查、缓存、对象池管理**等。

主要功能为：日志记录，性能统计，安全控制，事务处理，异常处理等。对这些行为进行分离，将其独立到非指导业务逻辑的方法中，从而改进这些行为的时候不影响业务逻辑。

面向对象编程（OOP）解决问题的重点在于对具体**领域模型的抽象**，而面向切面编程（AOP）解决问题的关键则在于对**关注点**的抽象。也就是说，系统中对于**一些需要分散在多个不相关的模块中解决的共同问题**，则交由AOP来解决；来实现松散耦合。

1. PointCut中的execution

execution(

modifiers-pattern? return-type-pattern declaring-type-pattern? **name-pattern**(param-pattern) throws-pattern?

)

execution中可指定**方法可见性**、**返回值**、**所在包**、**方法名**、**参数名**、**方法声明的异常**，其中除了name-pattern外，其它都是可选的。

1. Adivce通知

各种类型的通知包括：

@Component  
@Aspect  
@Pointcut()  
@Around()  
@Before()  
@After()  
@AfterReturning  
@AfterThrowing

etc..

## 两种配置方式

### 使用注解配置

1. @Component  
   @Aspect  
   public class PointCuts2 {  
     
    @Pointcut("execution(public \* com.duan.springweb.controller.\*.\*(..))")  
    public void pointcut() {  
    }  
      
    @Around("pointcut()")  
    public Object around(ProceedingJoinPoint pjp) throws Throwable {  
    Utils.*ol*.accept("Around 环绕执行之前");  
    Object obj = pjp.proceed(pjp.getArgs());  
    Utils.*ol*.accept("Around 环绕执行之后");  
    return obj;  
    }  
      
   }

### 使用xml配置

使用AOP需要开启动态代理

<!--开启动态代理-->  
<**aop:aspectj-autoproxy**/>  
  
<**bean** id="aspect" class="com.duan.springweb.aspect.PointCuts">  
</**bean**>  
<**aop:config**>  
 <**aop:aspect** ref="aspect">  
 <**aop:pointcut** id="testAop"  
 expression="execution(public \* com.duan.springweb.controller.NoteController2.testAop(int,String))"/>  
 <**aop:after** method="after"  
 pointcut-ref="testAop"/>  
 <**aop:before** method="before"  
 pointcut-ref="testAop"/>  
 <**aop:after-returning** method="afterReturn"  
 pointcut-ref="testAop"/>  
 <**aop:after-throwing** method="afterThrowing"  
 pointcut-ref="testAop"/>  
 </**aop:aspect**>  
</**aop:config**>

## 原理

AOP的源码中用到了两种动态代理来实现拦截切入功能：**jdk动态代理和cglib动态代理**。

两种方法同时存在，各有优劣。jdk动态代理是由java内部的反射机制来实现的，cglib动态代理底层则是借助asm来实现的。

注意：jdk动态代理的应用前提，必须是目标类基于统一的**接口**。如果没有上述前提，jdk动态代理不能应用。由此可以看出，jdk动态代理有一定的局限性，cglib这种第三方类库实现的动态代理应用更加广泛，且在效率上更有优势。

参考：设计模式.docx 代理模式

# Spring 常用注解

## Spring常用注解汇总

<https://www.cnblogs.com/xingzc/p/5777814.html>

## @RequestMapping 用法详解之地址映射

<http://www.cnblogs.com/qq78292959/p/3760560.html>

### value的uri值为以下三类

1. 可以指定为普通的具体值；
2. 可以指定为含有某变量的一类值(URI Template Patterns with Path Variables)；

@RequestMapping(value="/owners/{ownerId}", method=RequestMethod.*GET*)  
public String findOwner(@PathVariable String ownerId, Model model) {  
 Owner owner = ownerService.findOwner(ownerId);  
 model.addAttribute("owner", owner);  
 return "displayOwner";  
}

1. 可以指定为含正则表达式的一类值( URI Template Patterns with Regular Expressions);

@RequestMapping("/spring-web/{symbolicName:[a-z-]+}-{version:\d\.\d\.\d}.{extension:\.[a-z]}")  
 public void handle(@PathVariable String version, @PathVariable String extension) {  
 // ...  
 }  
}

### parames和headers示例

仅处理请求中包含了名为“myParam”，值为“myValue”的请求；

@RequestMapping(value = "/pets/{petId}", method = RequestMethod.*GET*, params="myParam=myValue")  
public void findPet(@PathVariable String ownerId, @PathVariable String petId, Model model) {  
 // implementation omitted  
}

仅处理request的header中包含了指定“Referer”请求头和对应值为“http://www.ifeng.com/”的请求。即请求来自于指定网址。

@RequestMapping(value = "/pets", method = RequestMethod.*GET*, headers="Referer=http://www.ifeng.com/")  
public void findPet(@PathVariable String ownerId, @PathVariable String petId, Model model) {  
 // implementation omitted  
}

Java WEB高并发秒杀系统: <https://github.com/codingXiaxw/seckill>

## @RequestBody

<http://www.cnblogs.com/qq78292959/p/3760651.html>

作用：

该注解用于读取Request请求的body（**请求体）部分数据**，使用系统默认配置的HttpMessageConverter进行解析，然后把相应的数据绑定到要返回的对象上；再把HttpMessageConverter返回的对象数据绑定到 **controller中方法的参数**上。

说明：request的body部分的数据编码格式由header请求头部分的Content-Type指定；

## @ResponseBody

作用：

该注解用于将**Controller的方法返回的对象**，通过适当的HttpMessageConverter转换为**指定格式**后，写入到**Response对象的body数据区**。

## @ModelAttribute

@ModelAttrbuite：<http://hbiao68.iteye.com/blog/1948380> <http://blog.csdn.net/hejingyuan6/article/details/49995987>

1 绑定请求参数到模型对象（Model），放在处理方法形参上时，用于将多个请求参数绑定到一个模型对象，从而简化绑定流程。

如：参数中传入 (Model model,int id)：model.addAttruibute(“id”,23);

可简化为：(@ModelAttribute(“id”) int id)

**将数据保存到request域中**

2 将@RequestMapping方法返回值绑定到模型对象中，暴露返回值。

@ModelAttribute(“data”)

public Map<String,String> getData(){

….

}

所在页面控制器的同级方法被访问时该方法将首先调用，在jsp中就可通过${data}使用返回的数据。

如果不指明添加到模型对象中数据的名字，则使用返回类型的首字母小写类名代替：

@ModelAttribute

Public List<String> getData(){

…

}

Jsp中使用${list}使用，**最好指定，也是将数据保存到request域中。**

使用@ModelAttribute将公用的取数据的方法返回值传到页面，不用在每一个controller方法通过Model将数据传到页面。

## @RequestMapping的组合变种

<https://www.baidu.com/link?url=7UYXPYZVvY14sak36CAjFzHATUvRQFciEdQzVm4biUZ9rn_QcTTiWAhlE8H-5VHKgME_l-9KgREzPOK3eLafF_&wd=&eqid=d414aaae0000b214000000055a0d6189>

@GetMapping

@PostMapping

@DeleteMapping

@PatchMapping

## @ExceptionHandler

<https://www.cnblogs.com/shuimuzhushui/p/6791600.html>

统一处理某一类异常，从而能够减少代码重复率和复杂度，该注解作用对象为方法，并且在运行时有效，value()可以指定异常类。

由该注解注释的方法可以具有灵活的输入参数。不指明异常类，会自动映射。

1. 异常参数：包括一般的异常或特定的异常（即自定义异常），如果注解没有指定异常类，会默认进行映射。
2. 请求或响应对象 (Servlet API or Portlet API)： 你可以选择不同的类型，如ServletRequest/HttpServletRequest或PortleRequest/ActionRequest/RenderRequest。
3. Session对象(Servlet API or Portlet API)： HttpSession或PortletSession。
4. WebRequest或NativeWebRequest
5. Locale
6. InputStream/Reader
7. OutputStream/Writer
8. Model

方法返回值可以为：

1. ModelAndView对象
2. Model对象
3. Map对象
4. View对象
5. String对象
6. 还有@ResponseBody、HttpEntity<?>或ResponseEntity<?>，以及void

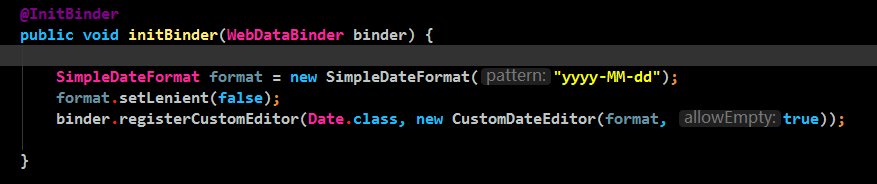
## @InitBinder

<https://www.cnblogs.com/android-blogs/p/5441808.html>

在使用SpringMVC的时候，经常会遇到表单中的日期字符串和JavaBean的Date类型的转换，而SpringMVC默认不支持这个格式的转换，所以需要手动配置，自定义数据的绑定才能解决这个问题。

在需要日期转换的Controller中使用SpringMVC的注解@initbinder和Spring自带的WebDateBinder类来操作。

WebDataBinder是用来绑定请求参数到指定的属性编辑器，由于前台传到controller里的值是String类型的，当往Model里Set这个值的时候，如果set的这个属性是个对象，Spring就会去找到对应的editor进行转换，然后再SET进去。



拓展：

spring mvc在绑定表单之前，都会先注册这些编辑器，Spring自己提供了大量的实现类，诸如CustomDateEditor ，CustomBooleanEditor，CustomNumberEditor等

## @ResponseStatus

可以将某种异常映射为HTTP状态码，通常与@ExceptionHandler一起使用。

@ResponseStatus 用于修饰一个类或者一个方法

### 修饰类

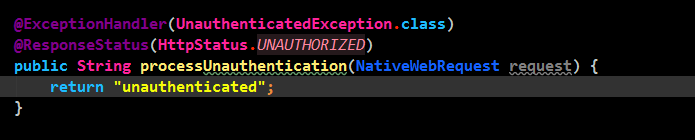
@ResponseStatus(value=HttpStatus.FORBIDDEN,reason="用户不匹配")

public class UserNotMatchException extends RuntimeException{}

它有两个属性，value属性是http状态码，比如404，500等。reason是错误信息

### 修饰方法

与@ExceptionHandler注解配合使用



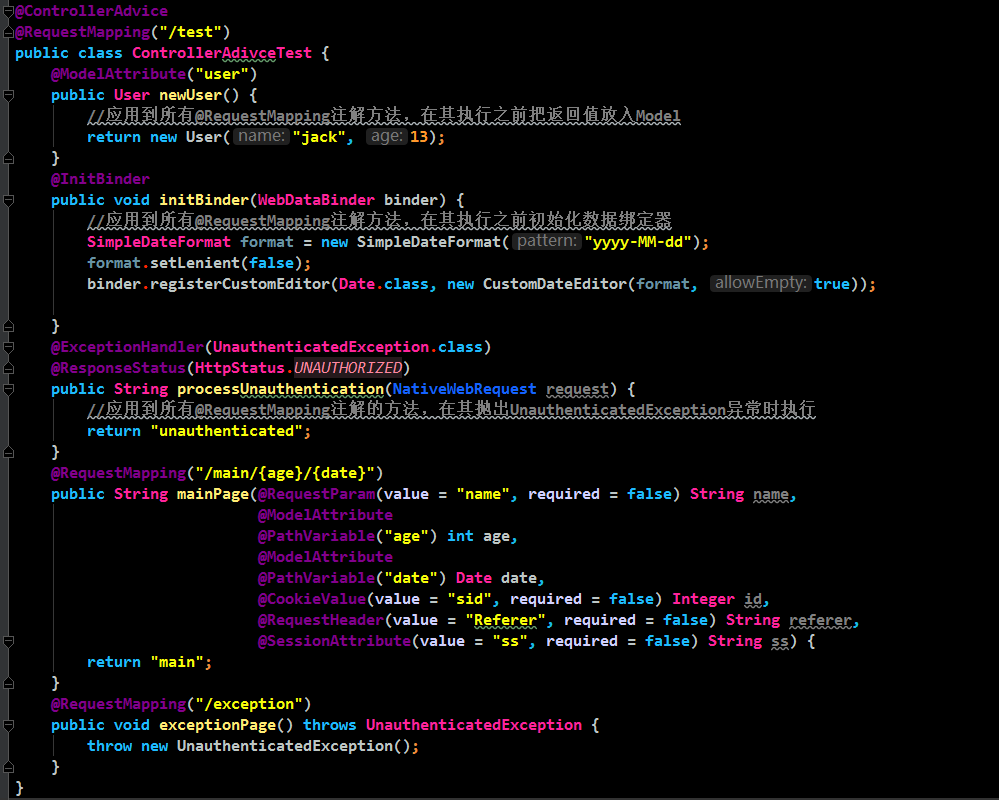
## [@ControllerAdvice](mailto:Spring3.2新注解@ControllerAdvice)(控制器增强 3.2新增)

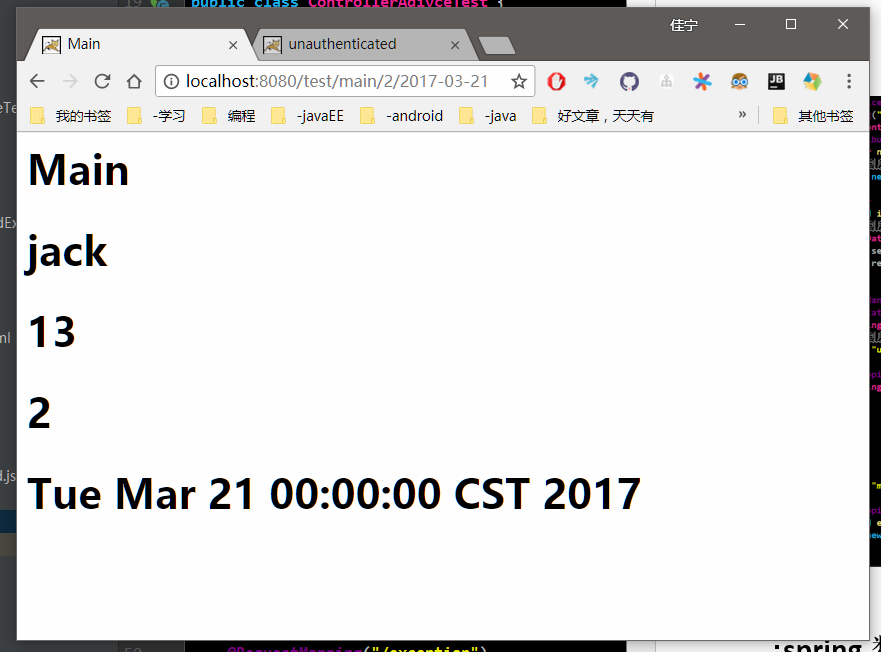
<http://jinnianshilongnian.iteye.com/blog/1866350>

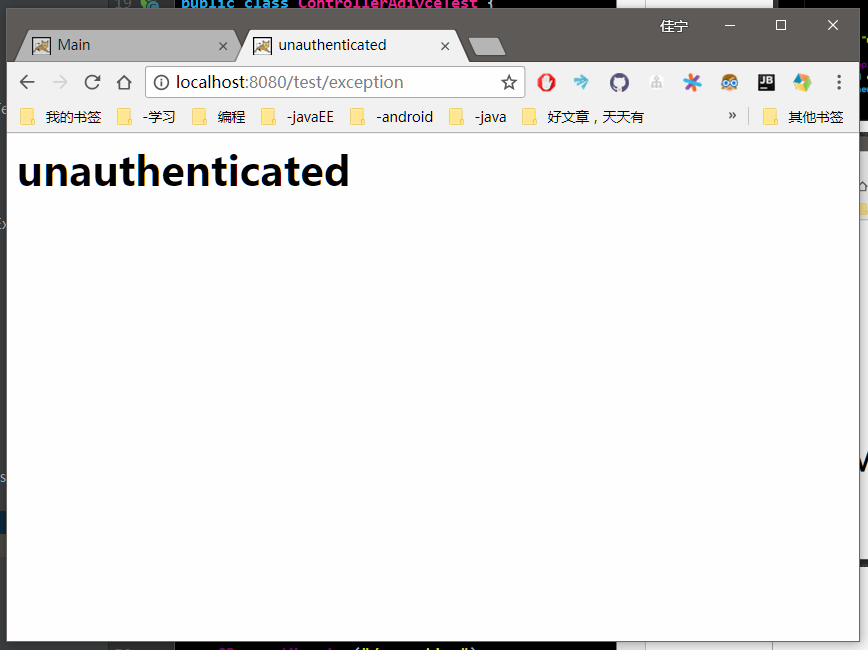
把@ControllerAdvice注解内部使用@ExceptionHandler、@InitBinder、@ModelAttribute注解的方法应用到所有的 @RequestMapping注解的方法。非常简单，不过只有当使用@ExceptionHandler最有用，另外两个用处不大。

@ExceptionHandler和@ControllerAdvice能够集中异常，使异常处理与业务逻辑分离

本文重点理解两种注解方式的使用







## spring数据绑定

<http://blog.csdn.net/li_xiao_ming/article/details/8365013>

1. @RequestParam绑定单个请求参数值；
2. @PathVariable绑定URI模板变量值；
3. @CookieValue绑定Cookie数据值
4. @RequestHeader绑定请求头数据；
5. @ModelValue绑定参数到命令对象；
6. @SessionAttributes绑定命令对象到session；
7. @RequestBody绑定请求的内容区数据并能进行自动类型转换等。
8. @RequestPart绑定“multipart/data”数据，除了能绑定@RequestParam能做到的请求参数外，还能绑定上传的文件等。

# Spring（IOC-DI）

Inversion of Controller Dependency Injection

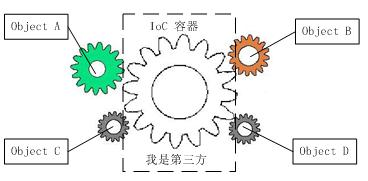
## 什么是控制反转

<http://mp.weixin.qq.com/s?__biz=MzAxOTc0NzExNg==&mid=2665513179&idx=1&sn=772226a5be436a0d08197c335ddb52b8&mpshare=1&scene=23&srcid=1128fMwbuYa57CExgOe46NMD#rd>

深入理解IOC和DI（依赖注入）<http://www.importnew.com/13619.html>

IOC和DI中包含的面向对象思想：开闭原则（O），里氏替换原则（L），依赖倒置原则（D），迪米特原则。

把复杂系统分解成相互合作的对象，这些对象类通过封装以后，内部实现对外部是透明的，从而降低了解决问题的复杂度，而且可以灵活地被重用和扩展。IOC理论提出的观点大体是这样的：借助于“**第三方**”实现具有**依赖关系的对象之间的解耦**。



由于引进了中间位置的“第三方”，也就是IOC容器，使得A、B、C、D这4个对象没有了耦合关系，齿轮之间的传动全部依靠“第三方”了，全部对象的控制权全部上缴给“第三方”IOC容器，所以，IOC容器成了整个系统的关键核心，它起到了一种类似“粘合剂”的作用，把系统中的所有对象粘合在一起发挥作用，如果没有这个“粘合剂”，对象与对象之间会彼此失去联系。

## 为什么需要注入依赖

普通的编程模式：

public class A {

private InterfaceB cb;

public A(){

cb = new ClassBImplementsInterfaceB();

}

public void method(){

cd.doThings();

}

}

其中InterfaceB接口定义了doThings方法。此时类A的依赖情况为：

A即依赖于接口（InterfaceB），又同时依赖实现（ClassBImplementsInterfaceB），因为它需要在编译阶段就确定使用哪种实现。

**使类A对于选择哪个具体InterfaceB实现类的权利从其中移除，转交给外部决定（Spring）（控制反转）。即它所依赖的对象由外部注入（决定）（依赖注入）**

开发中一个功能的完成往往需要多个类配合

直接写死组合类会使单元测试速度慢，不可重复，需要手工干预，不能独立运行

DI符合面向接口编程的思想。

## 传统编程和IoC的对比

传统编程：决定使用哪个具体的实现类的控制权在类本身，在编译阶段就确定了。

IoC模式：调用类只依赖接口，而不依赖具体的实现类，减少了耦合。控制权交给了容器，在运行的时候才由容器决定将具体的实现，动态的“注入”到调用类的对象中。

## IOC目的

降低类间耦合，增强代码复用性；

面向接口编程，实施依赖倒置原则；

提高系统可插入、可测试和可修改的特性。

## SpringIOC如何注入

1. 解析xml, 获取各种元素
2. 通过Java反射把各个bean 的实例创建起来
3. 通过Java反射调用注入实例

IOC容器负责容纳bean，并对bean进行管理，在Spring中，BeanFactory是IOC的核心接口，其职责包括：实例化、定位、配置引用程序中的对象，并建立这些对象间的依赖。常见的BeanFactory有XMLBeanFactory。

### Setter注入

创建bean对应的类（必须有一个空参可访问的构造器），为类中需要注入的成员变量创建setter方法；

在配置文件中创建注入配置；

<**bean** id="user" class="com.duan.springweb.entity.User">  
 <**property** name="name" value="张二"/>  
 <**property** name="password" value="张二aabc"/>  
 <**property** name="state" value="0"/>  
 <**property** name="age" value="23"/>  
</**bean**>

其中property的name对应成员变量的名字（调用其setter方法），value为要注入的值（setter传入的实参）。

### 构造器注入

为Bean对应的类创建构造器，定义好传入的参数；

在配置文件中配置：

<**bean** id="user2" class="com.duan.springweb.entity.User">  
 <**constructor-arg** type="java.lang.String">  
 <**value**>王五</**value**>  
 </**constructor-arg**>  
 <**constructor-arg** type="java.lang.String">  
 <**value**>王五0001</**value**>  
 </**constructor-arg**>  
 <**constructor-arg** type="int">  
 <**value**>2</**value**>  
 </**constructor-arg**>  
 <**constructor-arg** type="int">  
 <**value**>23</**value**>  
 </**constructor-arg**>  
</**bean**>

<constructor-arg/>在构造器参数有歧义时需指定type属性以进行区分，否则会注入失败。

构造器歧义示例：

User(String name, int state, String password, int age) {

User(String name, String password, int state, int age) {

## IOC原理

<https://www.cnblogs.com/ITtangtang/p/3978349.html>

IOC中最基本的技术就是“反射(Reflection)”编程。

可以把IOC容器的工作模式看做是工厂模式的升华，可以**把IOC容器看作是一个工厂，这个工厂里要生产的对象都在配置文件中给出定义，然后利用编程语言的的反射编程，根据配置文件中给出的类名生成相应的对象。**从实现来看，IOC是把以前在工厂方法里写死的对象生成代码，改变为由配置文件来定义，也就是把工厂和对象生成这两者独立分隔开来，目的就是提高灵活性和可维护性。

### Spring IOC结构体系

Spring Bean的创建是典型的工厂模式，这一系列的Bean工厂，也即IOC容器为开发者管理对象间的依赖关系提供了很多便利和基础服务，在Spring中有许多的IOC容器的实现供用户选择和使用，其相互关系如下：

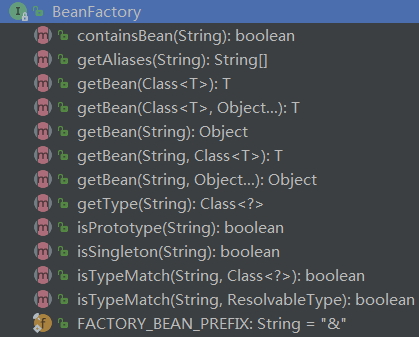


### BeanFactory

其中BeanFactory作为最顶层的一个接口类，它定义了IOC容器的基本功能规范，BeanFactory 有三个子类：ListableBeanFactory、HierarchicalBeanFactory 和AutowireCapableBeanFactory。但是从上图中我们可以发现最终的默认实现类是 DefaultListableBeanFactory。那为何要定义这么多层次的接口呢？查阅这些接口的源码和说明发现，**每个接口都有他使用的场合**，它主要是为了区分在 Spring 内部在操作过程中对象的传递和转化过程中，对对象的数据访问所做的限制。

1. **ListableBeanFactory** 接口表示这些 Bean 是可列表的
2. **HierarchicalBeanFactory** 表示的是这些 Bean 是有继承关系的，也就是每个Bean 有可能有父 Bean
3. **AutowireCapableBeanFactory** 接口定义 Bean 的自动装配规则。

这四个接口共同定义了 Bean 的集合、Bean 之间的关系、以及 Bean 行为.



在BeanFactory里只对IOC容器的基本行为作了定义，根本不关心你的bean是如何定义怎样加载的。正如我们只关心工厂里得到什么的产品对象，至于工厂是怎么生产这些对象的，这个基本的接口不关心。

spring提供了许多IOC容器的实现。比如**XmlBeanFactory**，**ClasspathXmlApplicationContext**等。其中XmlBeanFactory就是针对最基本的ioc容器的实现，这个IOC容器可以读取XML文件定义的BeanDefinition（XML文件中对bean的描述）,如果说XmlBeanFactory是容器中的屌丝，ApplicationContext应该算容器中的高帅富.

ApplicationContext是Spring提供的一个高级的IoC容器，它除了能够提供IoC容器的基本功能外，还为用户提供了以下的附加服务。从ApplicationContext接口的实现，我们看出其特点：



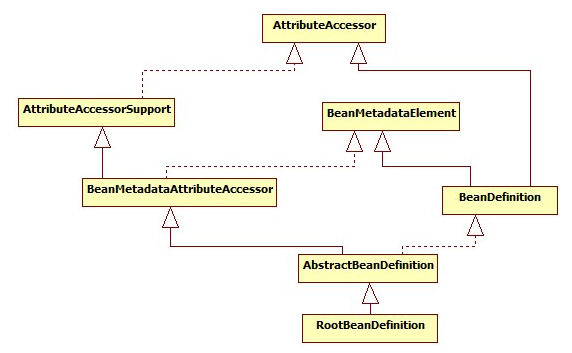
1. 支持信息源，可以实现国际化。（实现**MessageSource**接口）

2. 访问资源。(实现ResourcePatternResolver接口)

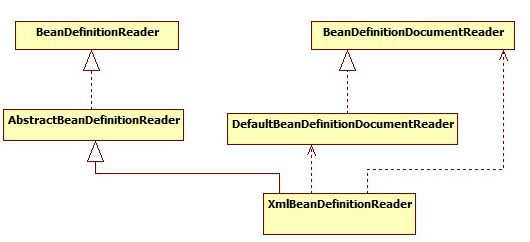
3. 支持应用事件。(实现ApplicationEventPublisher接口)

### BeanDefinition

SpringIOC容器管理了我们定义的各种Bean对象及其相互的关系，Bean对象在Spring实现中是以BeanDefinition来描述的，其继承体系如下：



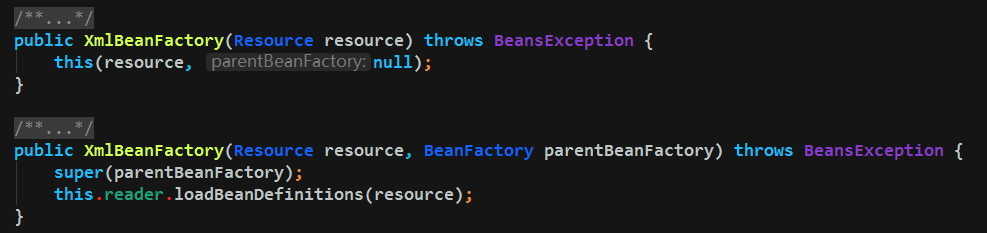
Bean 的解析过程非常复杂，功能被分的很细，因为这里需要被扩展的地方很多，必须保证有足够的灵活性，以应对可能的变化。**Bean 的解析主要就是对 Spring 配置文件的解析**。这个解析过程主要通过下图中的类完成：



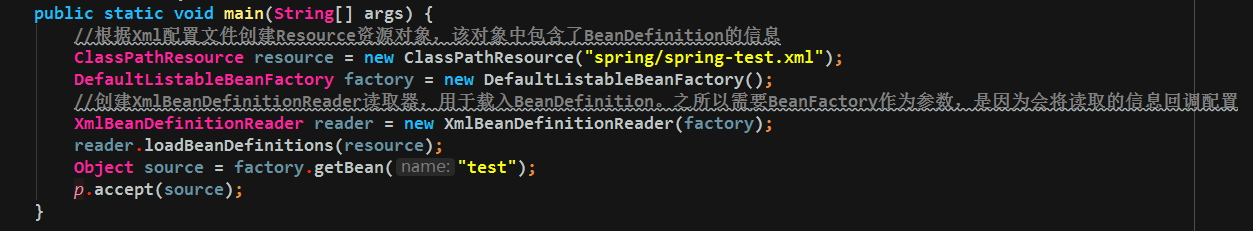
### IoC容器的初始化

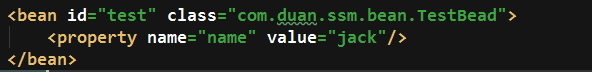
IoC容器的初始化包括BeanDefinition的**Resource定位**、**载入**和**注册**这三个基本的过程。

#### XmlBeanFactoryd容器的创建过程



示例：





输出：

TestBead(name=jack)

# Spring 配置文件beans.xml头部配置解释

<https://www.cnblogs.com/EasonJim/p/6880329.html>

# Spring 中的线程安全性

<http://mp.weixin.qq.com/s?__biz=MjM5NzMyMjAwMA==&mid=2651479529&idx=2&sn=956b7870b4b4ddda694ddb9dd45e4115&chksm=bd2531968a52b880c62882188504e2f69fbc93765a6359988de2861774a6fe6df9b0ee941ec9&mpshare=1&scene=23&srcid=1201lPTSxvajWkRcQCP5vKmp#rd>

Spring作为一个IOC/DI容器，帮助我们管理了许许多多的“bean”。但Spring并没有保证这些对象的线程安全，需要由开发者自己编写解决线程安全问题的代码。

## Spring为bean提供的作用域

1. **singleton**：默认的scope，每个scope为singleton的bean都会被定义为一个单例对象，该对象的生命周期是与Spring IOC容器一致的（但在第一次被注入时才会创建）。
2. **prototype**：bean被定义为在每次注入时都会创建一个新的对象。
3. **request**：bean被定义为在每个HTTP请求中创建一个单例对象，也就是说在单个请求中都会复用这一个单例对象。
4. **session**：bean被定义为在一个session的生命周期内创建一个单例对象。
5. application：bean被定义为在ServletContext的生命周期中复用一个单例对象。
6. websocket：bean被定义为在websocket的生命周期中复用一个单例对象。

## 使用ThreadLocal声明有状态的变量

Spring根本就没有对bean的多线程安全问题做出任何保证与措施。对于每个bean的线程安全问题，根本原因是每个bean自身的设计。不要在bean中声明任何有状态的实例变量或类变量，如果必须如此，那么就使用ThreadLocal把变量变为线程私有的，如果bean的实例变量或类变量需要在多个线程之间共享，那么就只能使用synchronized、lock、CAS等这些实现线程同步的方法了。

# HandlerMapping和HandlerAdapter 原理

<http://shenzhang.github.io/blog/2014/09/19/spring-mvc-qa/>

# 通过AOP的拦截机制实现了事务

tx:adivce 和aop:config配置事务

见《数据库笔记》四种隔离级别，七种传播行为。

