**Spring MVC**

# spring MVC简介

Spring MVC属于SpringFrameWork的后续产品，已经融合在Spring Web Flow里面。Spring 框架提供了构建 Web 应用程序的全功能 MVC 模块。使用 Spring 可插入的 MVC 架构，从而在使用Spring进行WEB开发时，可以选择使用Spring的SpringMVC框架或集成其他MVC开发框架，如Struts1，Struts2等。

# 使用 spring MVC

## 导入jar包

Spring整体，以及logging.jar

## 在web.xml中配置核心控制器(前端控制器) DispatcherServlet

<servlet>

<servlet-name>springMVC</servlet-name>

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

<init-param>

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

<param-value>/WEB-INF/spring-servlet.xml</param-value>

</init-param>

<load-on-startup>2</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>springMVC</servlet-name>

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

</servlet-mapping>

## 在WEB-INF下配置spring MVC的配置文件

文件名任意，通常情况下，文件名为 核心控制器配置时<servlet-name>的值 加-servlet.xml 例如 spring-servlet.xml。当不这样命名时，需要在配置核心控制器是通过下面方式指定

### 配置web.xml,指定配置文件

<param-value>/WEB-INF/spring-servlet.xml</param-value>

### 启动spring的注解

<context:annotation-config />

### 启动bean扫描

<context:component-scan base-package="web.controller" />

### 启动spring MVC 注解方式

<bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter" />

或<mvc:annotation-driven />

# Controller（控制器）

## @Controller 表示这个类是控制层

## @RequestMapping配置Controller的访问路径

加在类上，也要加在方法上

例如：

@Controller

@RequestMapping(value="/RegisterController.action")

**public** **class** RegisterController{

这个controller的访问路径为 RegisterController.action

## controller中的方法：

返回值类型 任意

方法名 任意

参数列表 任意

建议： 返回String类型---- 用来做跳转路径

方法要想被访问，必须在方法也添加@RequestMapping

@RequestMapping

**public** String register(){

**return** "index.jsp";

}

## 关于@RequestMapping

### 加在类上@RequestMapping(value="RegisterController.action")

配置controller的访问路径

### 加在方法上@RequestMapping

使得方法可以被访问

### 加在方法上 @RequestMapping 一些其他属性的配置

@RequestMapping(method=RequestMethod.GET)

表示这个方法只有 是GET请求时才能访问

### 一个controller中有多个方法时，如何区分访问

@RequestMapping(method=RequestMethod.GET,params="method=aaa")

这个方法的访问路径为：

HelloWorldController.action?method=aaa

### 不在class上配置@RequestMapping,只在 方法上配置 @RequestMapping

访问路径就是 项目名/helloworld/test.action

@RequestMapping(value="/helloworld/test")

## controller中 获取客户端提交的数据的方式：

### 方法中定义 HttpServletRequest类型的参数

**public** String register(HttpServletRequest request){

String name = request.getParameter(“name”);

**return** "index.jsp";

}

@RequestParam("name")

### 在方法参数中使用 @RequestParam 注解

**public** String register(@RequestParam("name") String name,@RequestParam("age") int age){

System.out.println(name);

System.out.println(age);

@RequestParam("name") 中的 "name" 要和请求参数的key值相同

**return** "index.jsp";

}

注意：当请求参数和 方法参数同名时，@RequestParam注解可以省略

### 使用 @PathVariable 注解 --这种用法比较特殊，(少用)

@RequestMapping(value="/hello/{name}/{age}")

public String hello(@PathVariable("name") String name,@PathVariable("age") int age){

//省略了 class 上的@RequestMapping

// 访问这个方法？？？ hello/zhangsan/100.action

System.out.println(name+"\t"+age);

return "/helloworld.jsp";

}

### .直接在方法参数中获得（\*\*\*\*最常这样用）

@RequestMapping(value="/login.action")

**public** String login(String name,String pwd){

System.***out***.println(name + "~" + pwd);

**return** "/WEB-INF/jsp/index.jsp";

}

## **controller 方法中 在作用域中设置值带回给客户端**

### 通过 request.setAttribute();

### 通过方法的参数 ModelMap model,在页面可以使用el表达式获得

**public** String register(ModelMap modelMap){

modelMap.put(“key”,Object); --放在request作用域

}

## controller中使用Session

### 通过request 获取session request.getSession()

### 使用 @SessionAttributes(必须加载类上)

@SessionAttributes({"user","name",.....}) @RequestMapping(value="RegisterController.action")

**public** **class** RegisterController{

**public** String register(ModelMap modelMap){

modelMap.put("user",user); --放在request作用域

modelMap.put("name","中软高科");

}

}

modelMap中存的值 将会在session中也保存一份

## **关于跳转**

### 返回字符串 默认是转发方式

### 重定向 return “redirect:index.jsp”

### 跳转路径 前缀后缀的设置

<!-- 配置视图

<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"

p:prefix="/WEB-INF/view/" p:suffix=".jsp">

<property name="viewClass">

<value>org.springframework.web.servlet.view.JstlView</value>

</property>

</bean>

-->

return “index” 将会跳转到 /WEB-INF/view/index.jsp

9. 关于使用ModelAndView

它是这个包的类--org.springframework.web.servlet.ModelAndView;

通常用于方法的返回值

**public** ModelAndView test2(){

String name = “中软高科”;

return new ModelAndView(new Redirect(“index.jsp”));

return new ModelAndView(new Redirect(“index.jsp”),”name”,name); -- 另一种带值得方式

return new ModelAndView(new RedirectViews(“index.jsp”),map); --map 为Map 也是用来带值

}

或

**public** ModelAndView test2(){

ModelAndView mav = new ModelAndView();

mav.setViewName(viewName); //设置跳转路径

mav.addObject(obj) // 这种设置方式，带的值 key为这个对象所属类的 类名首字母小写

mav.addObject(“key”,obj)

return mav;

}

## @ModelAttribute注解

作用一：获取请求参数：

public String register(@ModelAttribute User user,@RequestParam("birthday111") String birthday)

客户端提交的数据会直接封装 user中，要保证请求参数的key和user的属性相同

还会将user存入request作用域

对于Date的birthday无法完成直接的赋值，因此单独进行配置

另一个作用：加在方法上，方法将会在controller中要调用的方法之前执行

@ModelAttribute("myList")

public List<User> getList(){ //这个方法将在 register方法之前执行

System.out.println("getList......");

User u1 = new User(1,"aaa",20,new Date());

User u2 = new User(2,"bbb",30,new Date());

User u3 = new User(3,"ccc",40,new Date());

List<User> list = new ArrayList<User>();

list.add(u1);

list.add(u2);

list.add(u3);

return list;

}

而且页面可以直接根据 myList使用 方法返回的list

使用C标签遍历即可。

# spring MVC的声明式异常处理，在web层中，一般可以作为所有controller的超类，在其中定义多个异常

@ExceptionHandler({Exception.**class**})//异常处理

**public** String exceptionHandler(Exception e){

System.*out*.println(e.getMessage());

**return** "exception.jsp";

}

# Spring MVC 文件上传

## 添加jar包

**commons-fileUpload.jar**

**commons-io.jars**

## 在spring-servlet.xml配置

<!-- 配置文件上传 -->

<bean id=*"multipartResolver"*

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

p:defaultEncoding=*"utf-8"*>

</bean>

## 单个文件上传

<form action=*"*${pageContext.request.contextPath }*/user/upload.action"*

method=*"post"*

enctype=*"multipart/form-data"*>

<input type=*"file"* name=*"file"* />

<input type=*"submit"*/>

</form>

@RequestMapping

**public** String upload(HttpServletRequest request) **throws**

Exception{

MultipartHttpServletRequest req = (MultipartHttpServletRequest)request;

MultipartFile mf = req.getFile("myFile");

System.out.println(mf.getName());

System.out.println(mf.getOriginalFilename());

System.out.println(mf.getSize());

InputStream is = mf.getInputStream();

String folderPath =

request.getSession().getServletContext().getRealPath("/upload");

OutputStream os = **new**

FileOutputStream(folderPath+"/"+mf.getOriginalFilename());

**int** len=0; **byte**[] b = **new** **byte**[1024];

**while**((len=is.read(b))>0){

os.write(b, 0, len);

}

is.close();

os.close();

**return** "success.jsp";

}

或：

@RequestMapping(value="/upload.action")

**public** ModelAndView upload(HttpServletRequest request, MultipartFile file, ModelAndView view){

ServletContext sc = request.getSession().getServletContext();

String realPath = sc.getRealPath("/file/");

String filename = file.getOriginalFilename();

**try** {

FileUtils.*writeByteArrayToFile*(**new** File(realPath, filename), file.getBytes());

} **catch** (IOException e) {

System.***out***.println("文件写入失败");

e.printStackTrace();

}

view.setViewName("index");

view.addObject("imgPath", "/file/" + filename);

**return** view;

}

## 多文件上传：

<form action=*"*${pageContext.request.contextPath }*/user/upload.action"*

method=*"post"*

enctype=*"multipart/form-data"*>

<input type=*"file"* name=*"file"s* />

<input type=*"file"* name=*"file"s* />

<input type=*"file"* name=*"file"s* />

<input type=*"file"* name=*"file"s* />

<input type=*"submit"*/>

</form>

**public** String upload(HttpServletRequest request,ModelMap model) **throws** Exception{

`

MultipartHttpServletRequest req = (MultipartHttpServletRequest)request;

List<MultipartFile> list = req.getFiles("myFile");

String folderPath = request.getSession().getServletContext().getRealPath("/upload");

**for**(MultipartFile mf:list){

**if**(!mf.isEmpty()){

OutputStream os = **new** FileOutputStream(folderPath+"/"+mf.getOriginalFilename());

InputStream is = mf.getInputStream();

**byte**[] b = **new** **byte**[1024];

**int** len = 0;

**while**((len=is.read(b))>0){

os.write(b, 0, len);

}

is.close();

os.close();

System.*out*.println(mf.getName());

System.*out*.println(mf.getOriginalFilename());

}

}

**return** "success.jsp";

}

或

@RequestMapping(value="/uploads.action")

**public** String uploads(HttpServletRequest request){

ServletContext sc = request.getSession().getServletContext();

String realPath = sc.getRealPath("/file/");

MultipartHttpServletRequest req = (MultipartHttpServletRequest)request;

//获得多个文件

List<MultipartFile> files = req.getFiles("files");

**for** (MultipartFile f : files) {

**if**(f.isEmpty()){

**continue**;

}

String filename = f.getOriginalFilename();

**try** {

FileUtils.*writeByteArrayToFile*(**new** File(realPath, filename), f.getBytes());

System.***out***.println("保存一个文件");

} **catch** (IOException e) {

System.***out***.println("文件写入失败");

e.printStackTrace();

}

}

**return** "index";

}

## 验证

<bean id="multipartResolver"

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

p:defaultEncoding="utf-8">

<property name="maxUploadSize">

<value>200000000000</value>

</property>

</bean>

<!-- SpringMVC在超出上传文件限制时，会抛出org.springframework.web.multipart.MaxUploadSizeExceededException -->

<!-- 该异常是SpringMVC在检查上传的文件信息时抛出来的，而且此时还没有进入到Controller方法中 -->

<bean id="exceptionResolver" class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">

<property name="exceptionMappings">

<props>

<!-- 遇到MaxUploadSizeExceededException异常时，自动跳转到/WEB-INF/jsp/error\_fileupload.jsp页面 -->

<prop key="org.springframework.web.multipart.MaxUploadSizeExceededException">error\_fileupload.jsp</prop>

</props>

</property>

</bean>

# spring MVC拦截器

自定义拦截器的两种方式：

## 实现HandlerInterceptor

**public** **class** MyInterceptor **implements** HandlerInterceptor {

@Override

**public** **boolean** X (HttpServletRequest arg0, HttpServletResponse arg1,

Object arg2) **throws** Exception {

//Object arg2 指的是下一个拦截器对象

System.*out*.println("Controller之前执行...");

**return** **true**;

}

@Override

**public** **void** postHandle(HttpServletRequest arg0, HttpServletResponse arg1,

Object arg2, ModelAndView arg3) **throws** Exception {

System.*out*.println("controller之后，生成视图之前执行");

}

@Override

**public** **void** afterCompletion(HttpServletRequest arg0,

HttpServletResponse arg1, Object arg2, Exception arg3)

**throws** Exception {

System.*out*.println("最后执行，一般用来释放资源");

}

}

## 继承 HandlerInterceptorAdapter

可以指定覆盖哪个方法

拦截器定义完成后，需要在配置文件中声明

<mvc:interceptors>

<!-- 使用bean定义一个Interceptor，直接定义在mvc:interceptors根下面的Interceptor将拦截所有的请求 -->

<bean class="web.interceptor.MyInterceptor"/>

<mvc:interceptor>

<mvc:mapping path="/RegisterController.action"/>

<!-- 定义在mvc:interceptor下面的表示是对特定的请求才进行拦截的 -->

<bean class="web.interceptor.MyInterceptor2"/>

</mvc:interceptor>

</mvc:interceptors>

# spring MVC 返回JSON数据

Controller 返回json格式数据：

## **导入jar包**

jackson-core-asl.jar

jackson-mapper-asl.jar

## 修改spring MVC 配置文件

<!-- 启动Spring MVC的注解功能，完成请求和注解POJO的映射 -->

<bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">

<property name="messageConverters">

<list>

<bean class="org.springframework.http.converter.json.MappingJacksonHttpMessageConverter">

<property name="supportedMediaTypes">

<list>

<value>text/html; charset=UTF-8</value>

<value>application/json;charset=UTF-8</value>

</list>

</property>

</bean>

</list>

</property>

</bean>

## controller：

@RequestMapping

**public** @ResponseBody List<User> test(String name){

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

User u1 = **new** User("aaa","123",20);

User u2 = **new** User("bbb","345",21);

User u3 = **new** User("ccc","234",22);

List<User> list = **new** ArrayList<User>();

list.add(u1);list.add(u2);list.add(u3);

**return** list;

}

或

@RequestMapping

@ResponseBody

**public** List<User> test(String name){

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

User u1 = **new** User("aaa","123",20);

User u2 = **new** User("bbb","345",21);

User u3 = **new** User("ccc","234",22);

List<User> list = **new** ArrayList<User>();

list.add(u1);list.add(u2);list.add(u3);

**return** list;

}

# 其他配置

## 编码过滤器（只对post方法有效）

<filter>

<filter-name>encoding</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>

<init-param>

<param-name>forceEncoding</param-name>

<param-value>true</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>encoding</filter-name>

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

</filter-mapping>

## 静态资源管理 (指定某个文件路径对外开放)

<mvc:annotation-driven/>

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

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

<mvc:resources location=*"/"* mapping=*"/\*.html"*/>

resources用于指定哪些静态资源不需要通过dispatcherServlet。

使用resources标签的时候必须搭配annotation-driven标签。如果没有annotation-driven，resources的内容会被禁止访问。

## POST请求参数

1. 一般的GET请求方法如下:

@GetMapping("/get")

public String get(String name){

return name;

}

http请求报文如下，参数是跟在URL后面:

GET http://127.0.0.1:8080/get?name=123 HTTP/1.1

cache-control: no-cache

Postman-Token: bb5563e8-6a0e-47a2-afa5-1f2b6f885744

User-Agent: PostmanRuntime/7.1.5

Accept: \*/\*

Host: 127.0.0.1:8080

accept-encoding: gzip, deflate

Connection: keep-alive

2. 一般的POST请求

@PostMapping("/post")

public String post1(String name){

return name;

}

http请求报文如下，参数是报文正文，Content-Type是请求类型multipart/form-data。请求的参数以键值对的形式发送到服务端。而服务端可以使用上面的方式获取参数。这种方式存在在POST请求中。

POST http://127.0.0.1:8080/post1 HTTP/1.1

Content-Type: multipart/form-data; boundary=--------------------------950323200802818776636126

cache-control: no-cache

Postman-Token: 598aa925-a5b8-4288-8c99-da4055734d51

User-Agent: PostmanRuntime/7.1.5

Accept: \*/\*

Host: 127.0.0.1:8080

accept-encoding: gzip, deflate

content-length: 162

Connection: keep-alive

----------------------------950323200802818776636126

Content-Disposition: form-data; name="name"

111

----------------------------950323200802818776636126--

3. POST请求参数为json格式

我们先看下http请求报。类型为Content-Type: application/json，参数在正文，是一个整体的参数。

POST http://127.0.0.1:8080/post2 HTTP/1.1

Content-Type: application/json

cache-control: no-cache

Postman-Token: 297b7c53-277a-4813-9029-f7cefc7c14b2

User-Agent: PostmanRuntime/7.1.5

Accept: \*/\*

Host: 127.0.0.1:8080

accept-encoding: gzip, deflate

content-length: 8

Connection: keep-alive

{name:1}

这个时候我们使用上面的方法就无法获得参数了，因为请求的参数汇集到一个json对象里面了。

我们想要获取json的请求值，必须使用@RequestBody，将请求值获取到注解修饰的对象中。一个方法中只能出现一次@RequestBody注解。当然我们想要使用Content-Type 是 txt, xml, html 等格式都是这样的格式。

@PostMapping("/post2")

public String post2(@RequestBody String name){

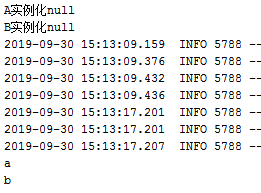
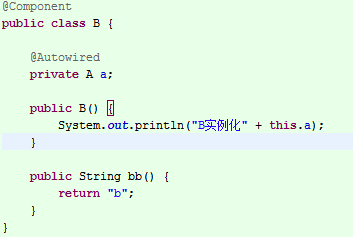
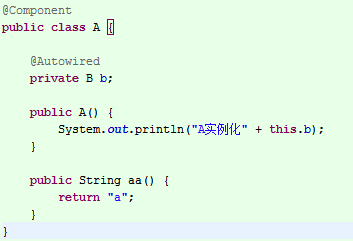
return name;

}

# 循环依赖

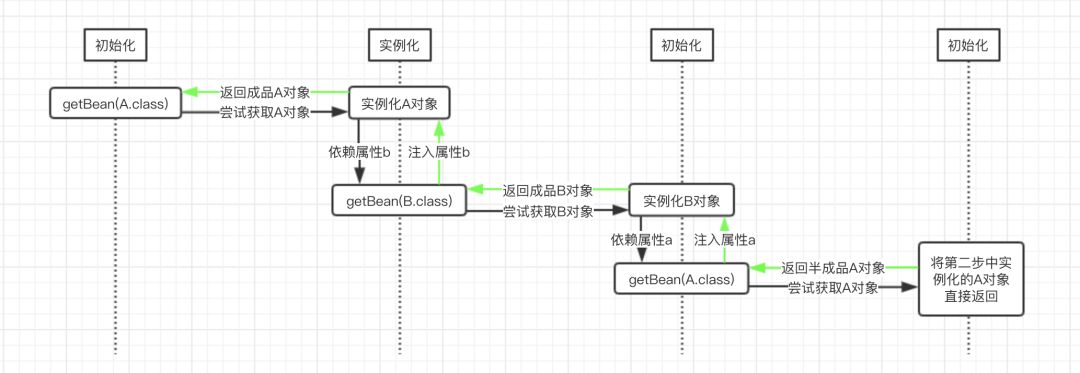
<https://www.imooc.com/article/34150>

## 循环案例



这里A和B中各自都以对方为自己的全局属性。这里首先需要说明的一点，Spring实例化bean是通过ApplicationContext.getBean()方法来进行的。如果要获取的对象依赖了另一个对象，那么其首先会创建当前对象，然后通过递归的调用ApplicationContext.getBean()方法来获取所依赖的对象，最后将获取到的对象注入到当前对象中。

我们在启动项目的时候，会发现spring在实例化对象的时候，属性的依赖是没有注入进去的。初始化后，在调用各自的方法时，是注入成功的。Spring实例化对象的过程如下



## Spring 实例化过程

对于Spring处理循环依赖问题的方式，我们这里通过上面的流程图其实很容易就可以理解

需要注意的一个点，Spring是如何标记开始生成的A对象是一个半成品，并且是如何保存A对象的。

这里的标记工作Spring是使用ApplicationContext的属性SetsingletonsCurrentlyInCreation来保存的，而半成品的A对象则是通过MapsingletonFactories来保存的

这里的ObjectFactory是一个工厂对象，可通过调用其getObject()方法来获取目标对象。在AbstractBeanFactory.doGetBean()方法中获取对象的方法如下

protected T doGetBean(final String name, @Nullable final Class requiredType,

@Nullable final Object[] args, boolean typeCheckOnly) throws BeansException {

// 尝试通过bean名称获取目标bean对象，比如这里的A对象

Object sharedInstance = getSingleton(beanName);

// 我们这里的目标对象都是单例的

if (mbd.isSingleton()) {

// 这里就尝试创建目标对象，第二个参数传的就是一个ObjectFactory类型的对象，这里是使用Java8的lamada

// 表达式书写的，只要上面的getSingleton()方法返回值为空，则会调用这里的getSingleton()方法来创建

// 目标对象

sharedInstance = getSingleton(beanName, () -> {

try {

// 尝试创建目标对象

return createBean(beanName, mbd, args);

} catch (BeansException ex) {

throw ex;

}

});

}

return (T) bean;

}

这里的doGetBean()方法是非常关键的一个方法（中间省略了其他代码），上面也主要有两个步骤

* 第一个步骤的getSingleton()方法的作用是尝试从缓存中获取目标对象，如果没有获取到，则尝试获取半成品的目标对象；如果第一个步骤没有获取到目标对象的实例，那么就进入第二个步骤
* 第二个步骤的getSingleton()方法的作用是尝试创建目标对象，并且为该对象注入其所依赖的属性。

这里其实就是主干逻辑，我们前面图中已经标明，在整个过程中会调用三次doGetBean()方法

第一次调用的时候会尝试获取A对象实例，此时走的是第一个getSingleton()方法，由于没有已经创建的A对象的成品或半成品，因而这里得到的是null

然后就会调用第二个getSingleton()方法，创建A对象的实例，然后递归的调用doGetBean()方法，尝试获取B对象的实例以注入到A对象中

此时由于Spring容器中也没有B对象的成品或半成品，因而还是会走到第二个getSingleton()方法，在该方法中创建B对象的实例

创建完成之后，尝试获取其所依赖的A的实例作为其属性，因而还是会递归的调用doGetBean()方法

此时需要注意的是，在前面由于已经有了一个半成品的A对象的实例，因而这个时候，再尝试获取A对象的实例的时候，会走第一个getSingleton()方法

在该方法中会得到一个半成品的A对象的实例，然后将该实例返回，并且将其注入到B对象的属性a中，此时B对象实例化完成。

然后，将实例化完成的B对象递归的返回，此时就会将该实例注入到A对象中，这样就得到了一个成品的A对象。

我们这里可以阅读上面的第一个getSingleton()方法：

@Nullable

protected Object getSingleton(String beanName, boolean allowEarlyReference) {

// 尝试从缓存中获取成品的目标对象，如果存在，则直接返回

Object singletonObject = this.singletonObjects.get(beanName);

// 如果缓存中不存在目标对象，则判断当前对象是否已经处于创建过程中，在前面的讲解中，第一次尝试获取A对象

// 的实例之后，就会将A对象标记为正在创建中，因而最后再尝试获取A对象的时候，这里的if判断就会为true

if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {

synchronized (this.singletonObjects) {

singletonObject = this.earlySingletonObjects.get(beanName);

if (singletonObject == null && allowEarlyReference) {

// 这里的singletonFactories是一个Map，其key是bean的名称，而值是一个ObjectFactory类型的

// 对象，这里对于A和B而言，调用图其getObject()方法返回的就是A和B对象的实例，无论是否是半成品

ObjectFactory singletonFactory = this.singletonFactories.get(beanName);

if (singletonFactory != null) {

// 获取目标对象的实例

singletonObject = singletonFactory.getObject();

this.earlySingletonObjects.put(beanName, singletonObject);

this.singletonFactories.remove(beanName);

}

}

}

}

return singletonObject;

}

这里我们会存在一个问题就是A的半成品实例是如何实例化的，然后是如何将其封装为一个ObjectFactory类型的对象，并且将其放到上面的singletonFactories属性中的。

这主要是在前面的第二个getSingleton()方法中，其最终会通过其传入的第二个参数，从而调用createBean()方法，该方法的最终调用是委托给了另一个doCreateBean()方法进行的

这里面有如下一段代码：

protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final @Nullable Object[] args)

throws BeanCreationException {

// 实例化当前尝试获取的bean对象，比如A对象和B对象都是在这里实例化的

BeanWrapper instanceWrapper = null;

if (mbd.isSingleton()) {

instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);

}

if (instanceWrapper == null) {

instanceWrapper = createBeanInstance(beanName, mbd, args);

}

// 判断Spring是否配置了支持提前暴露目标bean，也就是是否支持提前暴露半成品的bean

boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences

&& isSingletonCurrentlyInCreation(beanName));

if (earlySingletonExposure) {

// 如果支持，这里就会将当前生成的半成品的bean放到singletonFactories中，这个singletonFactories

// 就是前面第一个getSingleton()方法中所使用到的singletonFactories属性，也就是说，这里就是

// 封装半成品的bean的地方。而这里的getEarlyBeanReference()本质上是直接将放入的第三个参数，也就是

// 目标bean直接返回

addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));

}

try {

// 在初始化实例之后，这里就是判断当前bean是否依赖了其他的bean，如果依赖了，

// 就会递归的调用getBean()方法尝试获取目标bean

populateBean(beanName, mbd, instanceWrapper);

} catch (Throwable ex) {

// 省略...

}

return exposedObject;

}

到这里，Spring整个解决循环依赖问题的实现思路已经比较清楚了。对于整体过程，读者朋友只要理解两点：

Spring是通过递归的方式获取目标bean及其所依赖的bean的；

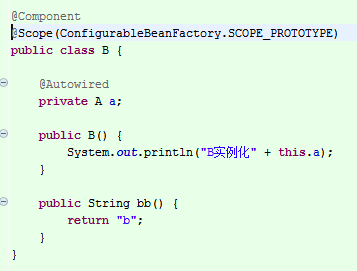
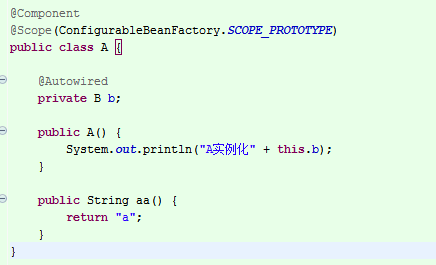
Spring实例化一个bean的时候，是分两步进行的，首先实例化目标bean，然后为其注入属性。

结合这两点，也就是说，Spring在实例化一个bean的时候，是首先递归的实例化其所依赖的所有bean，直到某个bean没有依赖其他bean，此时就会将该实例返回，然后反递归的将获取到的bean设置为各个上层bean的属性的。

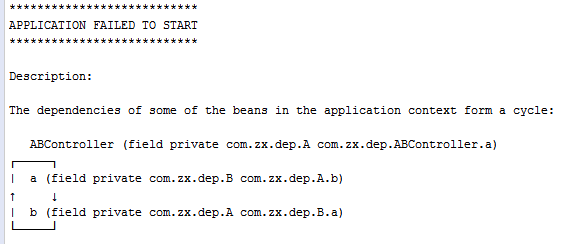
## 注意

### 只针对单例对象

上面只是对单例Bean解决循环依赖吧，靠内部3个Map对象解决的，原型Bean直接抛异常不处理



启动的时候直接报错，提示如下



### 单例对象中使用了@Async

符合下面条件的时候，项目启动会失败

* 已开启@EnableAsync的支持
* @Async注解所在的Bean被循环依赖了

# Spring bean生命周期

<https://blog.csdn.net/qq_36520235/article/details/82780301>

# Spring Validation

Spring Validation最佳实践及其实现原理： <https://mp.weixin.qq.com/s/gL7qAKd_Rvu-N4ENZOtUNQ>

Java api规范（JSR303）定义的bean校验标准接口。但是没有具体的实现。

目前最好的实现方案是hibernate的实现。

* 支持简单的字段检验
* 支持分组校验，同一个对象在不同的场景，可以进行不同字段的校验
* 嵌套校验，校验字段为对象的字段。
* 校验集合，对集合中的对象字段进行校验
* 自定义校验。自定义注解，进行校验
* 可以设置校验失败模式