# spring

# 1、spring 使用配置文件加载注解的Bean

## 1、spring 添加可以扫描的配置

|  |
| --- |
| <context:component-scan base-package=*"org.component"* />  <context:component-scan base-package=*"gbu/bs/foundation"* />  <context:component-scan base-package=*"nc/bs"* />  <context:component-scan base-package=*"org.dsp.ea.pay"* />  <context:component-scan base-package=*"org.dsp.ea.pout"* />  <!-- 启用Spring对AspectJ的支持，- proxy-target-class默认"false",更改为"ture"使用CGLib动态代理 - -->  <aop:aspectj-autoproxy proxy-target-class=*"true"* /> |

## 2、建立service类 （使用spring自带的读取，一定要再service中添加名字，否则不能够成功，除非使用之前的Applicaionhoder才可以读取类名）

|  |
| --- |
| /\*\*  \* **@author** 作者 HealerJean  \* **@version** 创建时间：2017年9月8日 上午9:43:06  \* 类说明  \*/  @Service("AspectStyleService")  **public** **class** AspectStyleService {    **public** **void** save(String name) {  System.*out*.println("save方法");  }    } |

## 3、取得配置文件获取bean，AspectStyleServiceName为上面的名字

|  |
| --- |
| /\*\*  \* **@author** 作者 HealerJean  \* **@version** 创建时间：2017年9月8日 上午9:33:38  \* 类说明  \*/  **public** **class** AspectStypeTest {  **public** **static** **void** main(String[] args) {    ClassPathXmlApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext-dsp.xml" );  AspectStyleService aspectStyleService = (AspectStyleService)context.getBean("AspectStyleServiceName");  aspectStyleService.save("HealerJean");  }  } |

## 4、右键运行成功（直接右键运行，不需要启动tomcat）



# 2、@Aspect（指定相关路径下的java，拦截方法，并且捕获处理异常可以用到）例如处理Controller下的异常，等通过多个catch处理

# 1、在捕获方法的同时处理异常（控制台不会报错，除非再抛出）

## 1、Service方法（有异常），service名字有用

|  |
| --- |
| @Service("AspectStyleServiceName")  **public** **class** AspectStyleService {    **public** AspectEntry save(String name,**int** age) **throws** AsperctMyException {  System.*out*.println("save方法");  AspectEntry person = **new** AspectEntry(name, age);  System.*out*.println("接下来会有1/0");  **int** i = 1/0;  //当使用@Aspect 中捕获了异常之后，around中必须给一个返回值，有了返回值代表已经结束这个方法了，那还执行这个干什么，就不会执行这个了  System.*out*.println("1/0在我上面");    **return** person;  }  } |

## 2、添加一个@Aspect注解的类，注意一定要@Component，用来专门识别上面的service

### 1、@Pointcut("execution(\*\*\*\*spectStyleService.\*(..))")用来拦截AspectStyleService中的方法

### 2、这个注解的方法anyMethod，名字下面都要用到，请严重注意

|  |
| --- |
| @Component  @Aspect //当前类标识为一个切面供容器读取 ,、@Aspect放在类头上，把这个类作为一个切面。  @Order(1) // 控制多个Aspect的执行顺序，越小越先执行  **public** **class** AspectStyleMethod {  // @Pointcut放在方法头上，定义一个可被别的方法引用的切入点表达式。  @Pointcut("execution(\* org.dsp.oa.xinjiangzzqpast.AspectStyleService.\*(..))")  **private** **void** anyMethod(){}  } |

### 3、@before前置加强,在捕获后运行方法前执行

|  |
| --- |
| @Before("anyMethod()") //标识一个前置增强方法  **public** **void** doAccessCheck() {  System.*out*.println("前置通知 @Before");  } |

### 4、 @After，final增强，不管是抛出异常或者正常退出都会执行，永远都会执行。（在处理异常之前执行）

|  |
| --- |
| @After("anyMethod()") //final增强，不管是抛出异常或者正常退出都会执行  **public** **void** after() {  System.*out*.println("最终结果 @After");  } |

### 4、@AfterReturning("anyMethod()") //后置增强，方法正常退出时执行（有了异常就不会执行和5相反）

|  |
| --- |
| @AfterReturning("anyMethod()") //后置增强，方法正常退出时执行  **public** **void** doAfter() {  System.*out*.println("后置通知 @AfterReturning");  } |

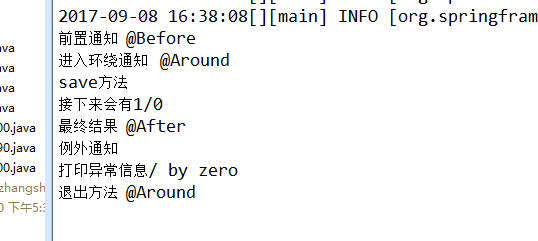
### 5、@AfterThrowing("anyMethod()") //异常抛出增强，有了异常才会执行，否则不能够执行（在异常处理之前执行）

|  |
| --- |
| @AfterThrowing("anyMethod()") //异常抛出增强  **public** **void** doAfterThrow() {  System.*out*.println("例外通知");  } |

### 6、@Around("anyMethod()") 环绕增强，出现了异常， service中则不会再往下执行，因为有了return了。

|  |
| --- |
| @Around("anyMethod()") //环绕增强，  **public** Object doBasicProfiling(ProceedingJoinPoint pjp) **throws** Throwable {  System.*out*.println("进入环绕通知 @Around");  **try** {  Object object = pjp.proceed();  //其实它并没有执行，因为AspectStyleService中有了异常，所以不会到这一步  System.*out*.println("pjp.proceed() 后执行");  **return** object;    }  //如果说捕获到异常，会当上面的最终结果 @After 和@AfterThrowing 执行完毕，再执行下面的  **catch** (Exception e) {  //取得方法签名  MethodSignature methodsignature = (MethodSignature) pjp.getSignature();  //方法签名得到返回的类型 这里返回的是一个我们的实体  **if** (methodsignature.getReturnType().equals(AspectEntry.**class**)) {  AspectEntry personFinal = **new** AspectEntry("重新授予Name", 20);    System.*out*.println("打印异常信息"+e.getMessage());  **return** personFinal; //后来加的给定返回值，也不会再执行service中异常之后的信息了，因为既然有了返回值，还执行个球  }  }    System.*out*.println("退出方法 @Around");  **return** **null**;  } |

### 6、控制台打印结果

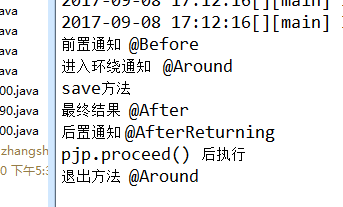


# 2、没有异常的 @Apect（将1/0和异常捕获全部注释掉了）

## 1、@Around注释的捕获方法如下

|  |
| --- |
| /\*\*  \* \* 这里必须返回一个object  \*  \*/  @Around("anyMethod()") //环绕增强，  **public** Object doBasicProfiling(ProceedingJoinPoint pjp) **throws** Throwable {  System.*out*.println("进入环绕通知 @Around");  Object object = pjp.proceed();  System.*out*.println("pjp.proceed() 后执行");//再after和afterreturn后执行  System.*out*.println("退出方法 @Around");  **return** object;  } |

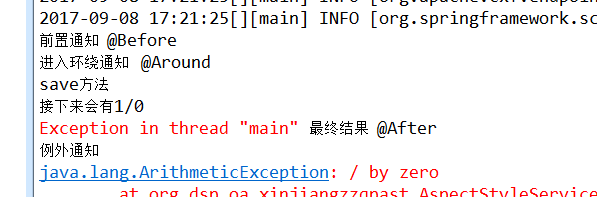
## 2、运行结果



# 3、有异常，没有捕获，会发现出现了异常就不会继续在mian中执行下去了，除非遇到了catch，

### 1、service 中遇到了异常，它下面的程序就不会执行了，直接被捕获了

## 2、around执行后，遇到了异常，没有捕获，就不会往下执行了



# 4、@Pointcut

## 1、通配符

### \*  匹配任意字符，但只能匹配一个元素

### .. 匹配任意字符，可以匹配任意多个元素，表示类时，必须和\*联合使用

### +  必须跟在类名后面，如Horseman+，表示类本身和继承或扩展指定类的所有类

## 2.、举例

### 1、execution(public \* \* (. .)) 任意公共方法被执行时，执行切入点函数。

### 2、execution( \* set\* (. .)) 任何以一个“set”开始的方法被执行时，执行切入点函数。  3、execution( \* com.demo.service.AccountService.\* (. .)) 当接口AccountService 中的任意方法被执行时，执行切入点函数。

### 4、execution( \* com.demo.service.. (. .)) 当service 包中的任意方法被执行时，执行切入点函数。

## 我的如下相当于3

|  |
| --- |
| @Pointcut("execution(\* org.dsp.oa.xinjiangzzqpast.AspectStyleService.\*(..))") |

## 下面的到时候看看

within(com.demo.service.) 在service 包里的任意连接点。 within(com.demo.service. .) 在service 包或子包的任意连接点。   
this(com.demo.service.AccountService) 实现了AccountService 接口的代理对象的任意连接点。   
target(com.demo.service.AccountService) 实现了AccountService 接口的目标对象的任意连接点。   
args(java.io.Serializable) 任何一个只接受一个参数，且在运行时传入参数实现了 Serializable 接口的连接