throwing="ex"不写报错：error at ::0 formal unbound in pointcut

<aop:after-throwing method="handlerException" pointcut-ref="businessService" throwing="e"/>

问题：看了很多关于Spring AOP的文章，在讲各种切入方式（before、around、after-returnning、thrown等）时，被切入的业务主体Bean的方法，基本都是无参数的。   
  
  
也有提到有参数的，但都是一个String型的参数。   
  
以before为例，无参数方法的切点配置为

**Xml代码  [收藏代码](javascript:void())**

1. **<aop:before** method="before" pointcut="execution(\* cn.xxxx..\*.\*(..))"**/>**

如果方法有一个String型的参数param，则配置为

**Xml代码  [收藏代码](javascript:void())**

1. **<aop:before** method="before" arg-names="param" pointcut="execution(\* cn.xxxx..\*.\*(..)) and args(param)"**/>**

注：如果参数其实可为任何类型，即Object。如果非要强调是String型，参数为其他类型的方法，不想被切点切入，则可写成 method="before(java.lang.String)"   
（经实际测试 arg-names="param" 不写也可以）   
  
但如果 cn.xxxx..\*.\* 的方法有多个参数，且个数不定，要想让切点可以切入，这么个写法就不行了。   
  
我搜了N多的帖子，也没能找到方法，最终几经辗转，终于在网友的帮助下，点破了这一层窗户纸，其实也很简单，还是在配置的写法：

**Xml代码  [收藏代码](javascript:void())**

1. **<aop:before** method="before" pointcut="execution(\* cn.xxxx..\*.\*(..)) and args(..)"**/>**

与之配合的切点的写法是

**Java代码  [收藏代码](javascript:void())**

1. **public** **void** before(JoinPoint jp) **throws** Throwable {
2. ...
3. }

这样，不论业务Bean的方法有多少个参数，都可以被这个切点切入了。如果需要访问各个参数，只需

**Java代码  [收藏代码](javascript:void())**

1. Object[] args = jp.getArgs();

----------------------------------华丽的分割线-----------------------------------   
  
其实，能够满足如上需求的方法，至少还有一种，就是拦截器。 

**Mxl代码  [收藏代码](javascript:void())**

1. <bean id="myService" class="org.springframework.aop.framework.ProxyFactoryBean">
2. <property name="interceptorNames">
3. <list>
4. <value>beforeInterceptor</value>
5. </list>
6. </property>
7. <property name="target">
8. <ref bean="realServiceTarget" />
9. </property>
10. </bean>
12. <bean id="realServiceTarget" class="cn.xxxx.xxx.Xxxx"/>
13. <bean id="beforeInterceptor" class="cn.xxxx.xxx.MyChecker"/>

思路是在注入ServiceBean时，偷梁换柱一下，用myService代替，实际是指向Spring的拦截器，它可以在执行真正的ServiceBean之前，先执行beforeInterceptor所指向的拦截代码   
（这里是MyChecker，这个拦截器要实现org.aopalliance.intercept.MethodInterceptor接口，并完成public Object invoke(MethodInvocation invocation) throws Throwable方法）然后再交还给target属性指明的真正的ServiceBean。它不仅仅能够得到方法的参数，而且还有更强的功能——决定是否继续执行target。

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** MyChecker **implements** MethodInterceptor {
3. /\*\*
4. \* 用户访问认证方法。
5. \* 如果登录合法则开始执行服务，否则返回错误。
6. \*/
7. **public** Object invoke(MethodInvocation invocation) **throws** Throwable {
8. Method invokeMethod = invocation.getMethod();
9. Object[] args = invocation.getArguments();
10. ...
11. **if**(....){
12. **return** invocation.proceed(); // 检查OK，继续执行
13. }**else**{
14. **return** **null**;  // 检查NG，阻断执行
15. }
16. }
17. }

看到了吧，这里不仅仅可以得到目标方法的参数 Object[] args = invocation.getArguments();   
还可以，控制是否要继续执行目标方法，还是阻断。因此用来做认证是再合适不过了。   
  
----------------------------------华丽的分割线-----------------------------------   
  
为什么要加入两条分割线之间的之一段呢？除了也作为before切入的另一种实现方式，更主要的是想引出下面的问题：   
  
我发现，在我成功的用AOP切入之后，每次都执行了两遍切点，且我检查了配置文件，并没有重复定义<AOP>   
  
当我在切点代码中加入

**Java代码  [收藏代码](javascript:void())**

1. Object target = jp.getTarget();

我发现，这个target有一次是我的ServiceBean，而另一次是类似$Proxy22之类的东西。这才恍然大悟，它应该是那个拦截器！也就是说，拦截器被切点切入了一次，真正的target又切入了一次。   
  
显然，从配置文件的写法上似乎是无法避免拦截器被切入的（虽然org.springframework.aop.framework.\* 并不是切面），那只能在切点里想办法回避了

**Java代码  [收藏代码](javascript:void())**

1. **public** **void** before(JoinPoint jp) **throws** Throwable {
2. Object target = jp.getTarget();
3. **if**(AopUtils.isAopProxy(target)){
4. **return**;
5. }
6. ...
7. }

至此，可以让多参数方法被切入的before切点完成了。   
  
（after-returnning没有遇到什么障碍，多参数也很简单，甚至不用在pointcut配置中声明，还可取到返回值）

**Xml代码  [收藏代码](javascript:void())**

1. **<aop:after-returning** method="afterReturning" arg-names="retVal" returning="retVal" pointcut="execution(\* cn.xxxx..\*.\*(..))"**/>**

切点，可以同时得到JoinPoint和返回值

**Java代码  [收藏代码](javascript:void())**

1. **public** **void** afterReturning(JoinPoint jp, Object retVal) **throws** Throwable {
2. Object target = jp.getTarget();
3. **if**(AopUtils.isAopProxy(target)){
4. **return**;
5. }
6. ...
7. }

最后，不知道哪位晓得，如果通过AOP而不是中间段落中提到的【拦截器】，来实现现有拦截器的功能，即取得入口参数，并可以控制继续执行被切的业务ServiceBean，还是阻止它执行。