

AOP ---- spring AOP是一个高仿本的AOP 仿造的不错

相对于AspectJ的 AOP Spring的AOP是一个简化版 但是这个简化版已经够用了



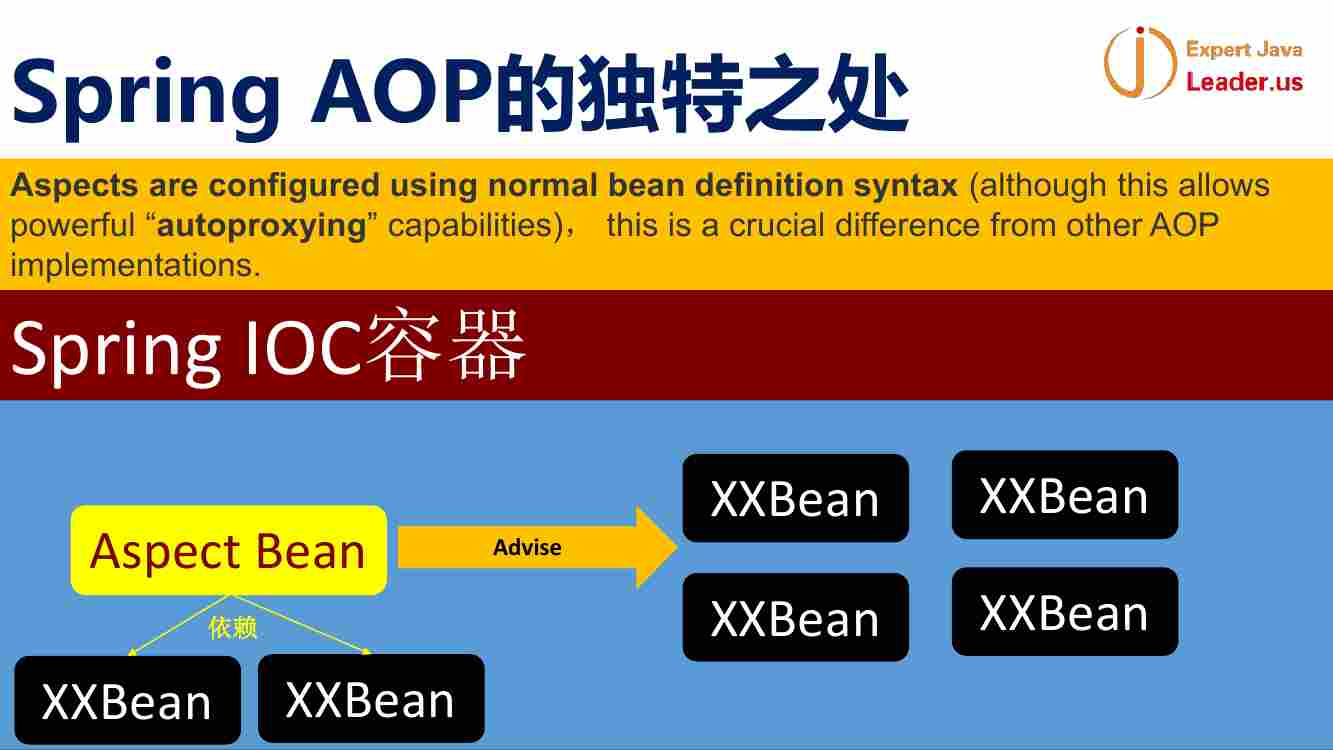
Spring一开始做了一套AOP 后来2.0的时候 转了 ---- 用了AspectJ ---- ASPECTJ是工业级的标准

2.0全面拥抱AspectJ

AspectJ定义了一套注解方式 ---- spring支持这个注解方式

Spring 把aspectJ集成进来 也是可以的 但是spring默认没这样做 ---- 默认仅仅是吧AspectJ的注解拿过来了

【PPT中说的是 Spring2.0引入了一种更加简单的并且更加强大的使用要么是schema的方式或者AspectJ的注解的方式的自定义切面。这两种方式提供了彻底的typed的通知并且使用了AspectJ的切点表达式语言---- 同时 仍然使用了spring aop来进行织入。Spring2.0 aop彻底向后兼容了Spring1.2的AOP】【也就是 aspectJ和2.0之前自己定义的aop的都进行了保留】



【注意到 PPT中写的是 Aspects are configured using normal bean definition syntax ---- 这句话正好印证了《Struts2技术内幕》里面说的 spring使用的是IOC来实现的AOP 但是 Struts2使用的是AOP来实现的IOC】

【PPT上面的意思是尽管这需要强大的自动代理autoproxying的能力，但是切面可以使用正常的bean的定义语法来进行配置 ----- 这是区别于其他AOP实现的最重要的不同的地方】

面向切面编程 就是 把切面代码写到Aspect中 ----- Aspect是要编织到目标应用的代码中

Spring 容器有一个非常特殊的地方 我要编织到某一个应用的代码 ---- 这个类在Spring中可以是一个普通的bean --- 这样就可以有其他依赖关系 ----- 这个是spring aop的优点 【因为ppt上面说了 spring中的Aspect 可以像正常的bean一样 ---- 类似图中画的一样 依赖其他的bean】



上一页ppt已经说了spring aop的有点



AspectJ融合到Spring中并不难 只是官方不乐意



【PPT中意思是 @AspectJ指的都是一种类似于使用Java注解标注的Java类的一样的声明切面的方式。 [这个@AspectJ的方式在AspectJ5的发行版中引入引来。Spring使用了AspectJ提供的类库来解释和AspectJ 5](mailto:这个@AspectJ的方式在AspectJ5的发行版中引入引来。Spring使用了AspectJ提供的类库来解释和AspectJ%205)相同的注解 ---- 对切点的解析和匹配。 尽管AOP的运行环境依然是单纯的Spring AOP 但是 这个和AspectJ的编译器和织入器没有任何关联

-----

为了在Spring配置中使用@Aspect的切面，你需要能够启用Spring对基于@Aspects的切面配置的Spring AOP支持 并且 自动代理基于是否被那些切面锁通知的的bean上面----（这块翻译的不好）======= 通过自动在理 意味着如果spring决定乐一个bean是被一个或者多个切面advised了，那么autoproxying就会自动为这个bean（被一个或者多个切面通知的bean）生成一个代理--- 这样就能够很插入(目标）方法的调用 并且 确保刚才的那个通知能够按照需要来执行 ------ 所以 恍然大悟 就是 <aop:aspect-autoproxy /> 对应的注解就是 @EnableAspectJAutoProxy】

这里面 autoproxy是容易引起误会的==== autoAproxying的意思 --- 很简单 spring中的AOP都是代理模式，也就是发现这个是aop编程的bena对象 就会自动设置成代理bean 这个就是自动代理

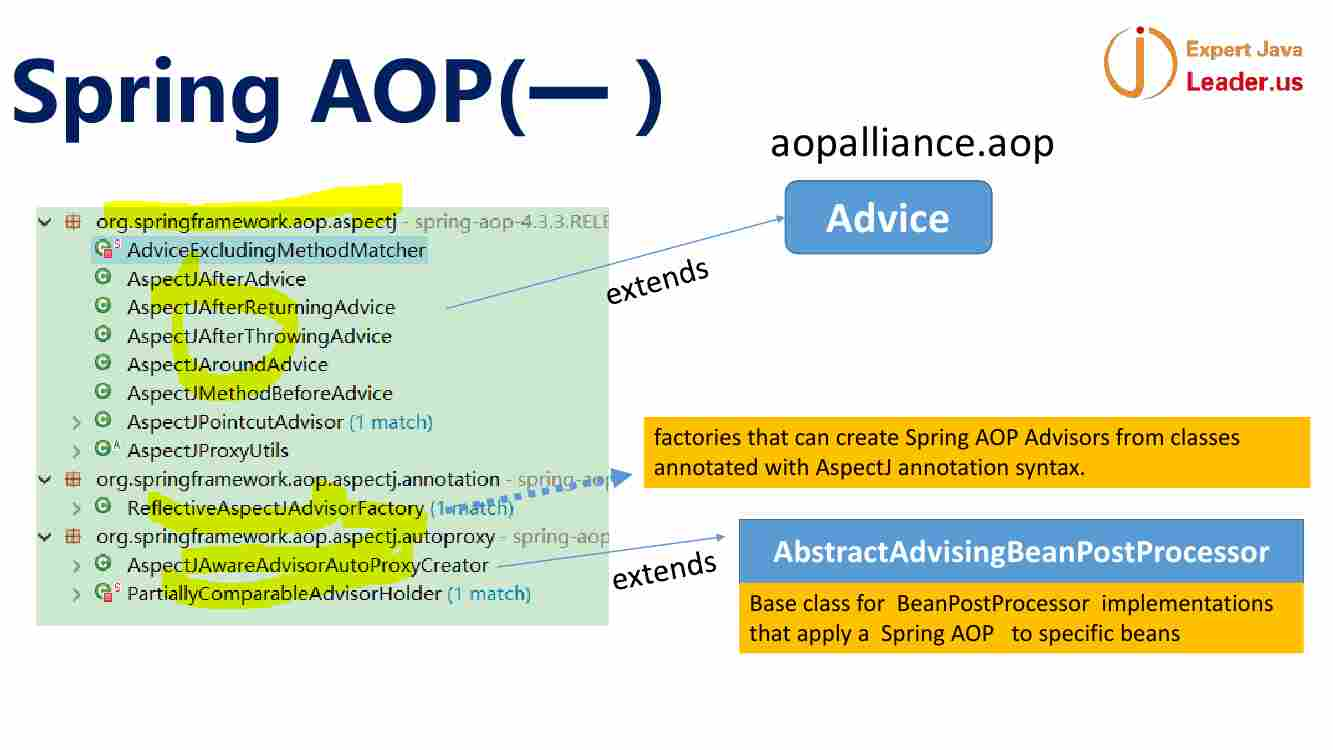
这个里面可以使用@Aspecj本身的注解来定义一个切面 使用方式都是一样的 ---- 但是 不依赖AspectJ的编译器和织入器 ---- 不依赖人家 该怎么实现呢？ 那就是使用代理 ------

做法@EnableAspectJAutoProxy ----含义是 开启AspectJ的注解 但是 使用spring本地的方法来实现【就是代理】

这个语句@EnableAspectAutoProxy绝不是开启AspectJ的支持 --- 而是使用了AspectJ的注解模式

这样做之后 就会发现某一个bean是否是使用了AspectJ的注解进行了修饰 ----- 如果是 那么就返回一个带有这个切面的代理对象

自动生成一个proxy



AOP联盟里面定义了基础的jar包 接口 所有的实现都要实现它 --- spring的作者也参与了这个联盟

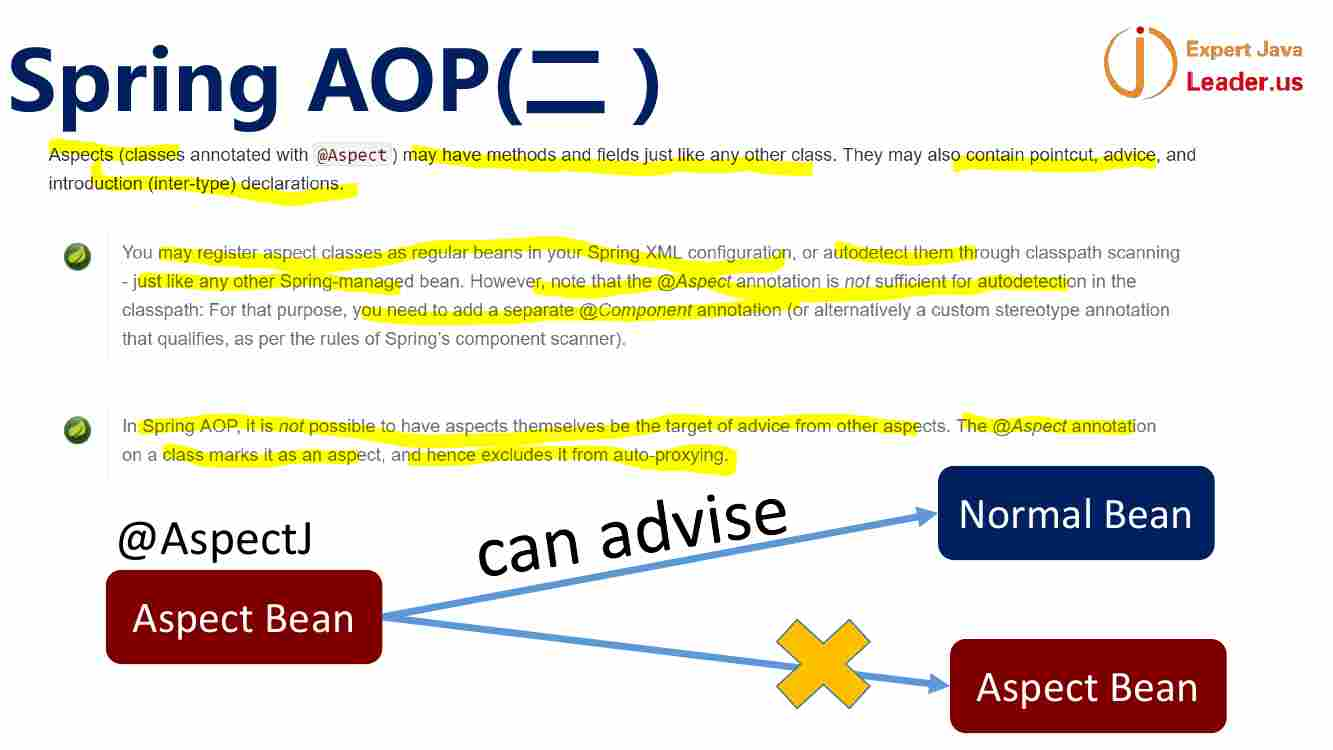
为了兼容1.2的AOP的能力 做了一些封装 ------- 这里面就是AspectJAfterAdvice 等等一系列类

【所以 ppt上写的这 这些类 **extends** aopaliance.aop.Advice】

Advisor是spring中特殊的词 相当于切面 --- -然后这里面有 【里面解释了 ReflectiveAspectJAdvisorFactory ---- 这个是工厂 可以从标记了AspectJ注解语法的类上面创建Spring AOP的Advisor ----- 有时间 回去复习一下Spring的AOP的基础知识 ---- 这个在包 aspect.annotation】

【AspectJAwareAutoProxyCreator和PartiallyComparableAdvisorHoldder 都是继承了AbstractAdvisingBeanPostProcessor ------- 对于BeanPostProcessor的基类 实现了能够把一个Spring的AOP应用到目标bean上面】

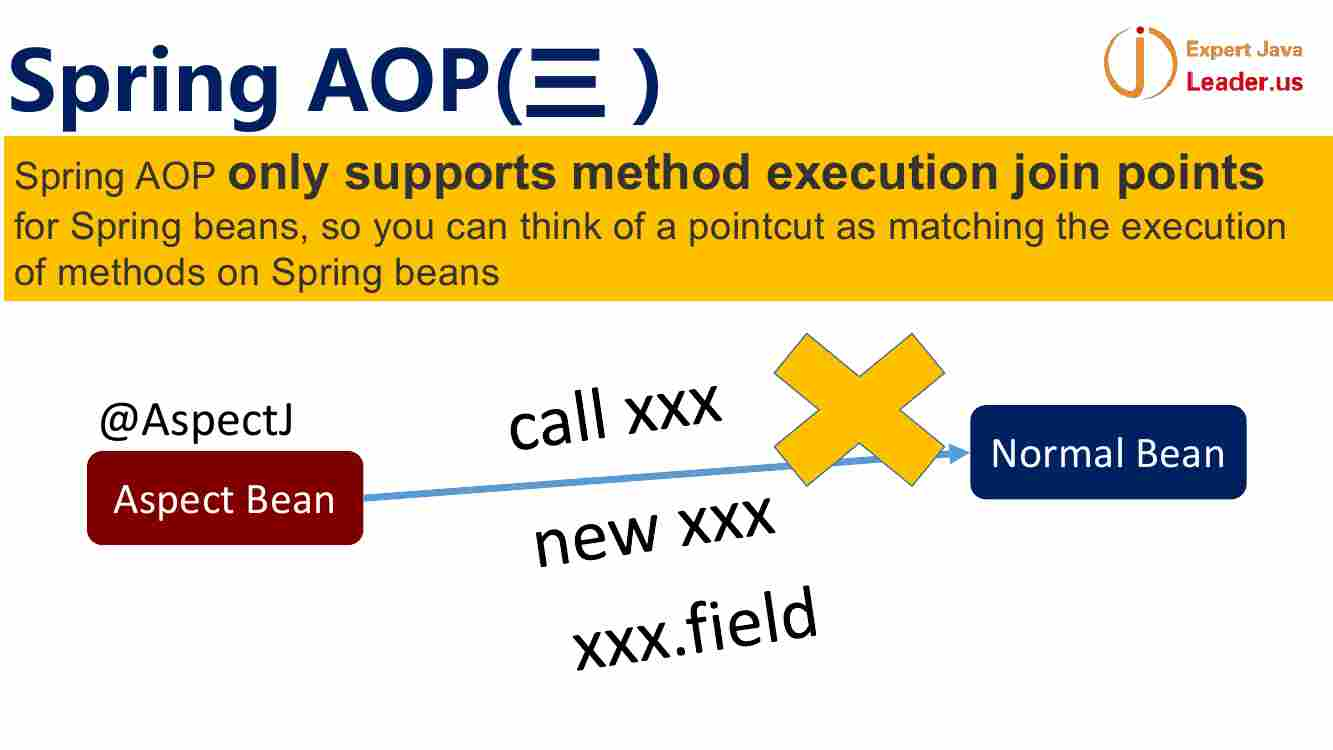
------- 这样 这两个类 扫描被AspectJ影响的类 然后 产生一个代理并返回 ---- 这个完成proxy的代理的创建。



@AspectJ注解的bean表示这个是一个bean ---- @pointcut是切面 ----- Advice表示定制的切面代码 ------ AspectJ 这个本身可以加一个@component或者 @Configurable 来进行自动扫描

还有切面本身不能装配切面 但是 切面本身可以装配任意一个bean

识别一个bean 是否是autoproxy bean --- 不能装配其他的bean ==== 这个是spring aop的一个局限性 还有一个局限性 在下页ppt



【Spring AOP仅仅支持 ---- 这个地方不要看了 看看spring的基础把 execution join points是什么意思】

标准的AspectJ能力很强 ---- 可以做bean对象的创建可以被发现和拦截的 另外 调用某个bean的方法 ---- 这个调用是在**调用者中加的代码** --🡪 去拦截 和 execution不一样的 ---- execution是执行里面的方法 但是call是在外面

Spring aop还有 不能对field进行拦截

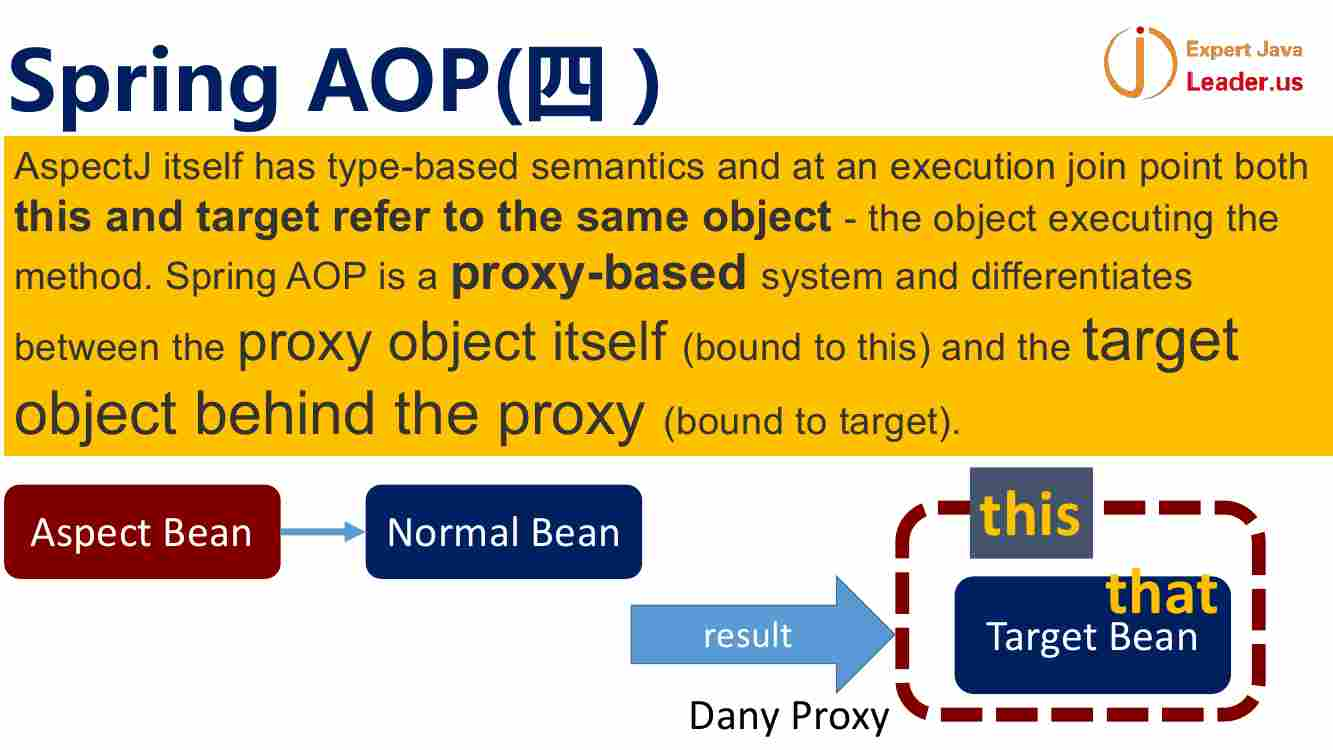
这几种情况很少见

比如调用这个Normal bean在外面拦截很少见

还有 new xxxx

Spring中的对象是container进行的 所以 没办法拦截 ------ spring就是对执行某一个方法进行拦截 足够了【Struts2是对某一个类Action进行拦截】

对属性的拦截很少见



由于AspectJ在调用这个方法之外做拦截 --- 有两个概念 一个是target 另一个就是this ---也就是proxy本身 --- spring仅仅限制【有点生搬硬套了。。。。 这块 需要了解AspectJ的用法才行】

【这个地方12：00 讲的不太好 这块】

【AspectJ本身有基于类型的语义并且在一个execution join point的地方 AspectJ本身(this)和目标对象都是同一个对象----- 也就是执行方法的对象。但是Spring AOP是一个基于代理的系统并且并且能够对代理对象本身（绑定到了this）和代理背后的目标对象（绑定到target上）区分开】

