

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



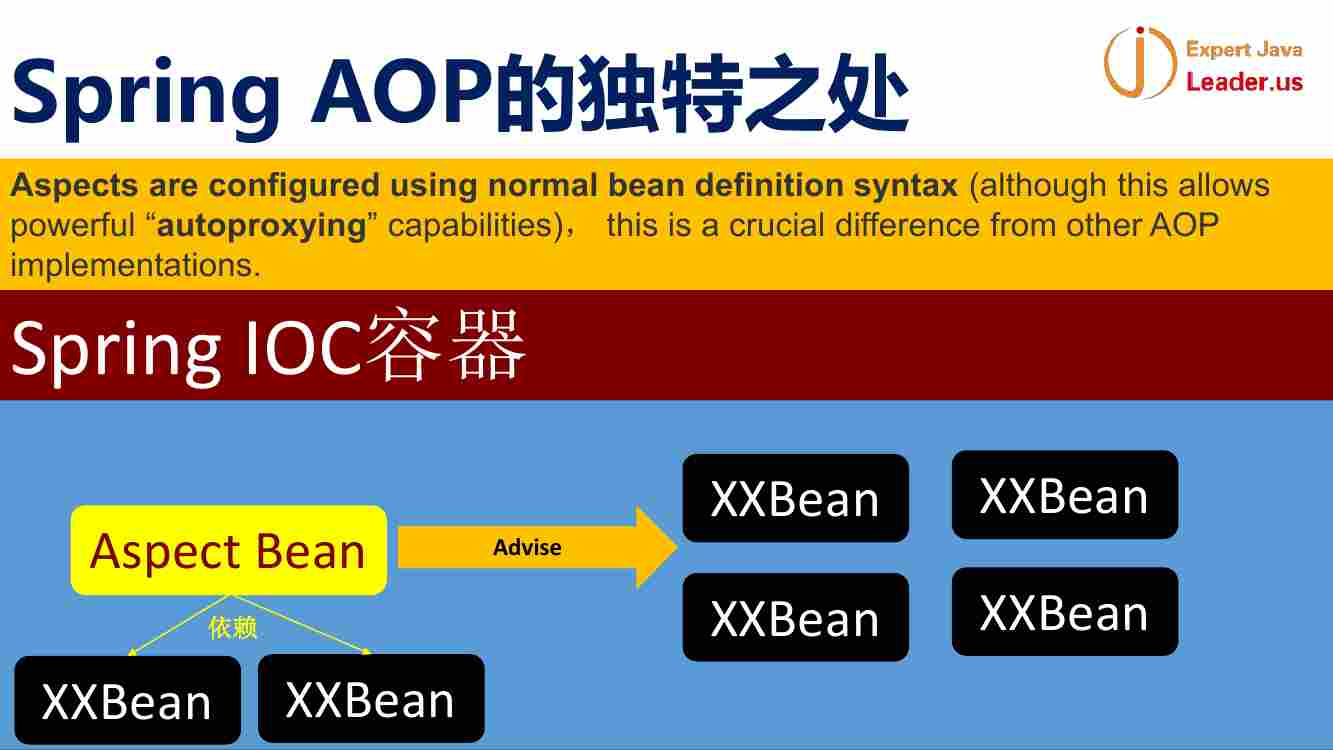
Spring一开始自己做了一套AOP 后来2.0的时候 转了 ---- 拥抱AspectJ ---- ASPECTJ在AOP是工业级的标准

2.0全面拥抱AspectJ **后面还是有问题的**

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** 如果是普通的bean 就可以进行装配引入数据库等等 ===== 这样就可以有其他依赖关系 **可以进行装配** 装箱 ----- 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（在图中就是黄颜色的圆角矩形 里面写的是Aspect Bean） ---- 类似图中画的一样 依赖其他的bean】 ----- ***图上面的意思 就是 我这个aspect在spring容器中就是一个普通的bean 这样可以依赖其他的bean --- 这样spring容器可以给他进行装配 然后这个spring装配好的aspect bean可以给其他Bean进行代理 ----- 这是优点 本身有容器 最大的特点***



上一页ppt已经说了spring aop的优点 就是官方支持



AspectJ融合到Spring中并不难 只是官方不愿意这样做

 autoproxy 【<aop:aspectj-autoproxy /> 对应的注解就是@EnableAspectJAutoProxy】

【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@AspaectJ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】

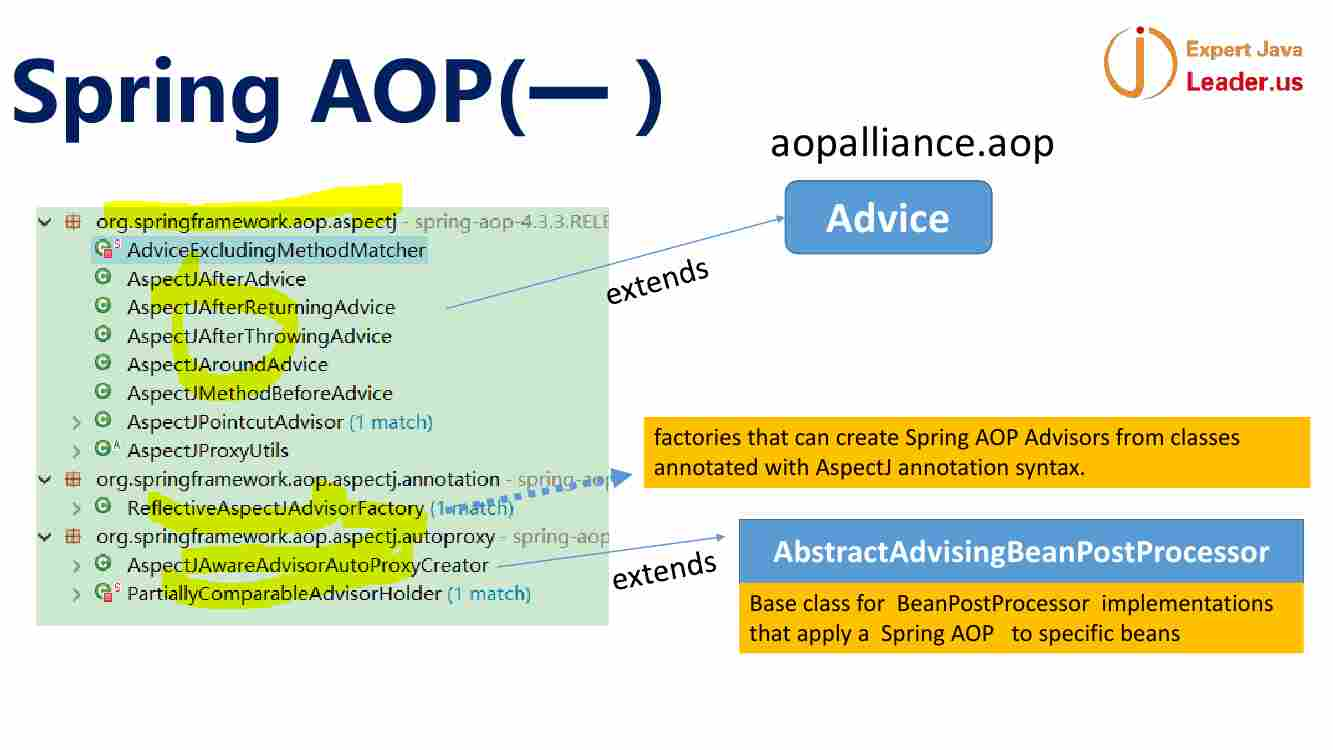
这里面 autoproxy是**容易引起误会的**==== autoAproxying的意思 --- 很简单 **spring中的AOP都是代理模式**，也就是发现这个是aop编程的bean对象 **就会自动设置成代理bean** 这个就是自动代理 autoproxy

这个里面可以使用Aspecj本身的注解来定义一个**切面（注解就是@AspectJ）** 使用方式和AspectJ用法是一样的---- 但是 **不依赖AspectJ的编译器**和**织入器**(PPT上写的就是 The AOP runtime is still pure Spring AOP though, and there is no dependency on the AspectJ compiler or weaver)【**所以 应该知道 AspectJ如果单独使用 该怎么用 才能知道这块的意思** *编译器 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 用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*** 【所以 这个PPT写的是容易误解的地方 使用的是AspectJ的注解模式 不是开启AspectJ的支持】

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 ---- 这就是proxy本身的意思



**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（规范- spring的作者也参与了这个联盟

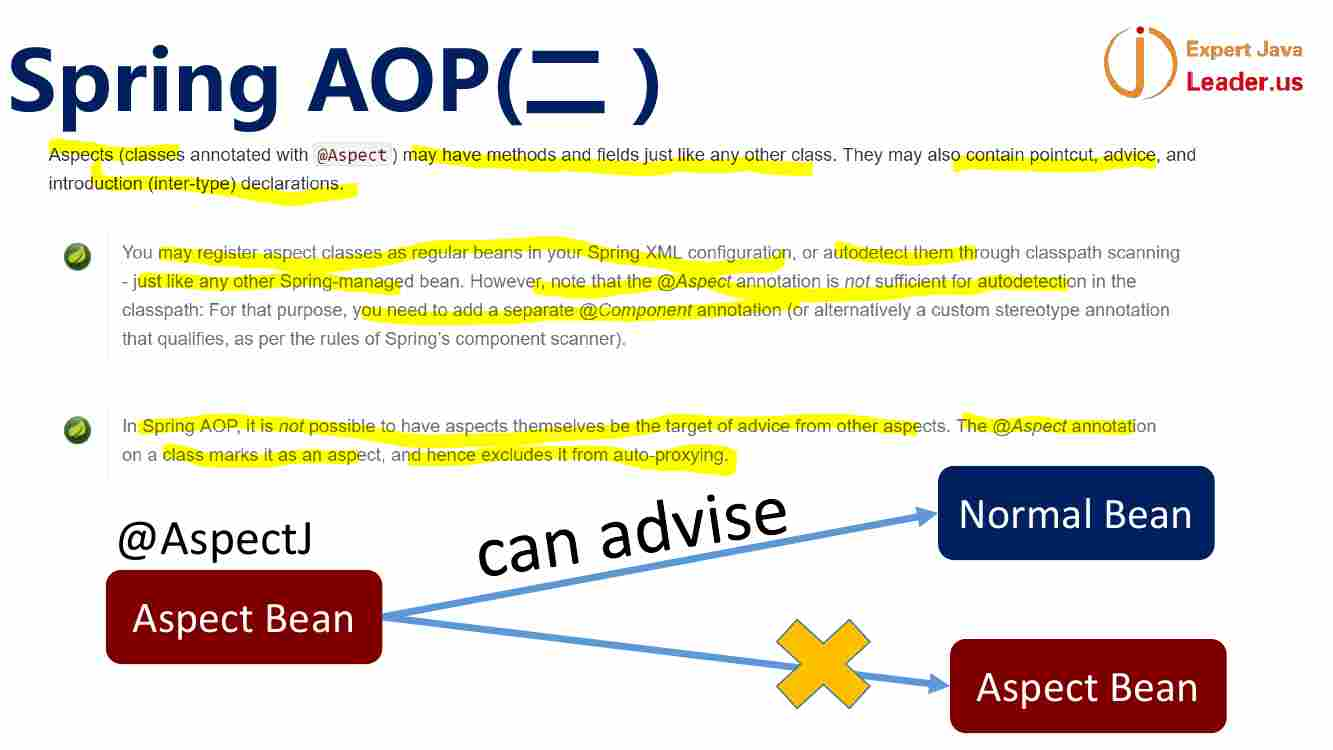
**为了兼容1.2的AOP的能力 做了一些封装** ------- 【**这部分类就是图中的org.springframework.aop.aspectj下面的这几个类 这个包之前学的是spring对aspectj的支持**】这里面就是AspectJAfterAdvice 等等一系列类 **继承了 aopalliance.aop.Advice这个类**

【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**（where） -----这个里面是org.sf.aop.aspectj.autoproxy.AspectJAwareAutoProxyCreator ------ 这个是继承了AbstractAdvisingBeanPostProcessor ---- 去扫描到被@AspectJ标记的类 然后 为这些类产生相关的代理*【这块可以写到PPT HW中】* 然后返回 完成proxy代理的创建【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表示这个是一个切面 （图中标记的左边的红颜色的框 使用了@Aspect ）***【试试看看 好像没有@AspectJ这个注解 应该是@Aspect注解】***

---- @pointcut是表示要匹配的特定的方法 就是切入点

----- Advice表示定制的切面代码

------ AspectJ 这个**本身可以加一个@Component或者 @Configuration 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普通bean ==== 所以 spring为了识别这个bean是否是Aspect 特意做了一套接口 ---- 如果是AOP本身的bean 就不会autoproxy 【可以写入作业】 这块重要**

【

用法就是

*@AspectJ ---- 这个是切面*

*public class MyAspect{*

*@Before(${pointcut-name}) ---- 这个就是Advice*

*Public void xxxx(){*

*}*

*@Pointcut(“execution()”) --- 这个就是pointcut*

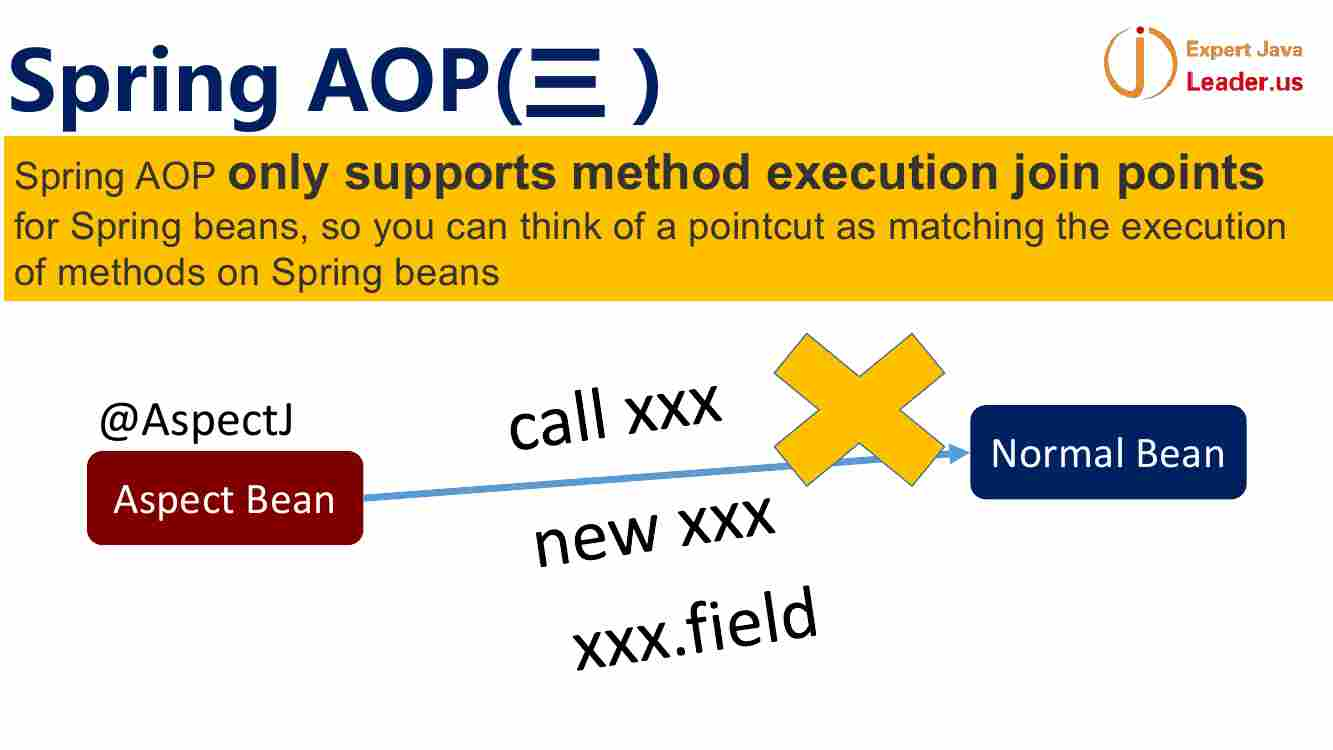
*public $pointcut-name{*

*}*

*}*

】

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 new出来本身 可以被发现和拦截的 【可以学aj 然后展示出来】

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 【需要看aspectJ的使用】

还有调用某个类的field

【图中的xxx都是指的同一个类的对象】

Spring aop还有 不能对field进行拦截

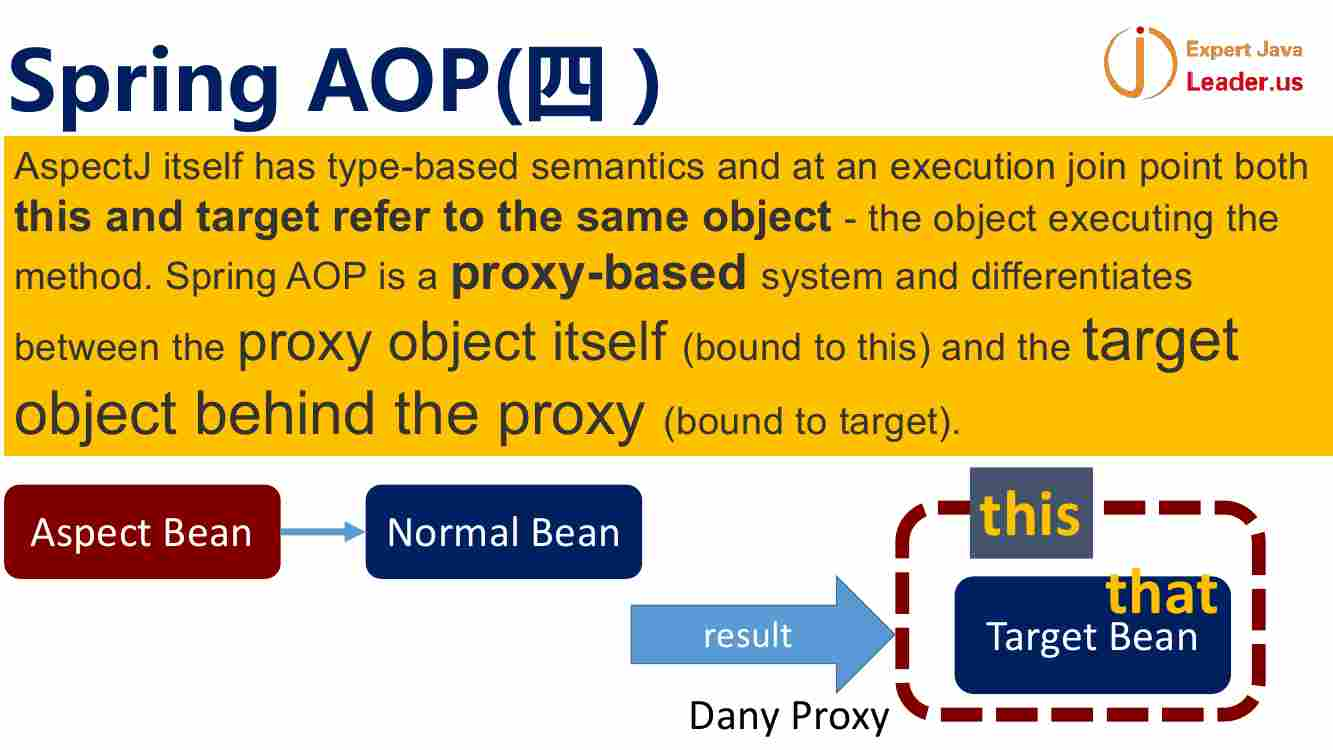
这几种情况很少见

**比如调用这个Normal bean在外面拦截很少见**

还有 new xxxx

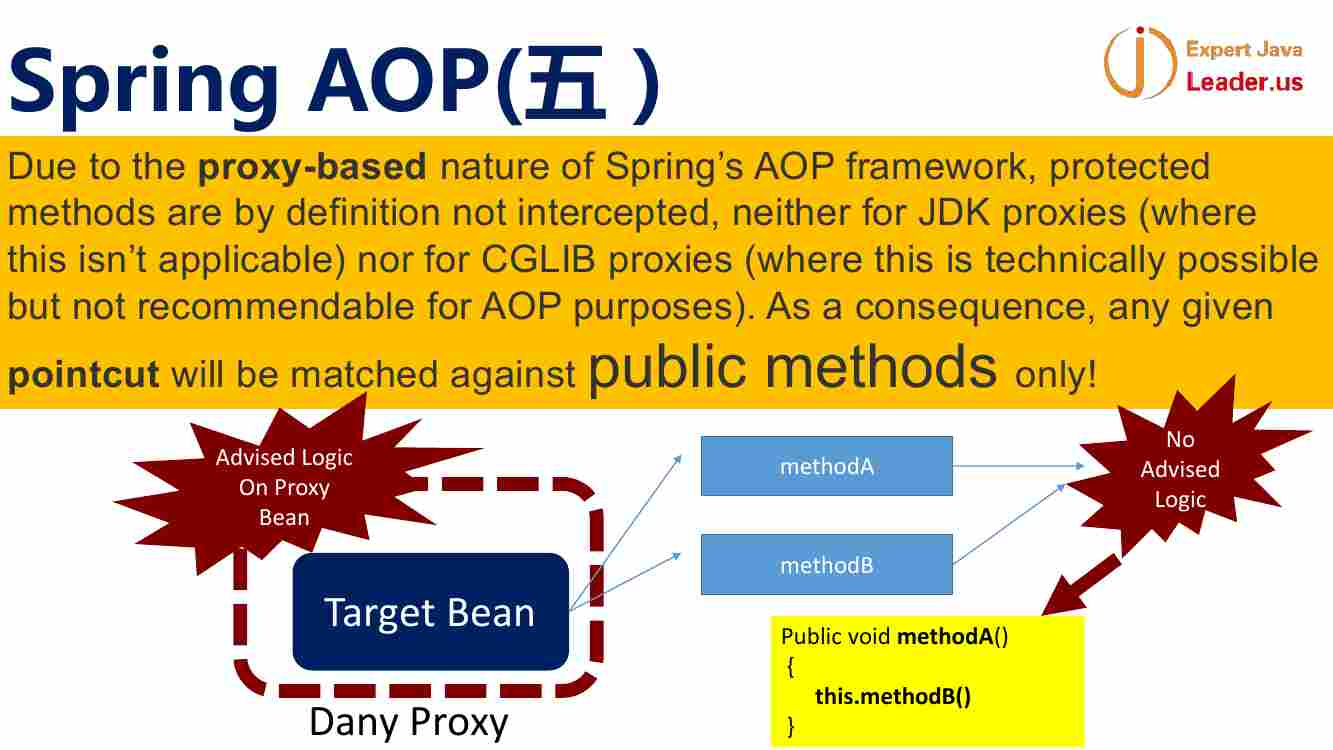
Spring中的对象是container进行的 **所以 没办法拦截** ------ 上面的两种很少用 ------ 所以 就execute某个方法去拦截 别的都不用 -------**spring就是对执行某一个方法进行拦截 足够了**【Struts2是对某一个类Action进行拦截】

对属性的拦截很少见 所以 也很少用 所以 上面是**打叉**



由于AspectJ在**调用这个方法之外做拦截 就是刚才的call** --- 所以有两个概念 一个是target +另一个就是this ---也就是**proxy本身 【ppt上也这么说】** --- spring执行某一个bean中的方法 原有的proxy和this是相等的 ----- 但是在Spring里面 proxy是外面的新对象 但是 targetbean就是被装配的对象 里面对象得不到外面的 外面得不到里面的 特殊编程可以 ---- spring aop中target bean就是被代理的目标对象 真正访问的时候 是外面的代理对象【PPT上也说 Spring AOP是一个基于代理的系统 并且区分了代理对象本身（绑定到this）和代理对象背后的目标对象（绑定到target） --- 这个绑定到this的意思就是 从spring容器中获取这个对象 调用方法 --- 对象中的this】

【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】



在spring aop是代理方式 是有些约束 和 默认方式

默认方式 如果发现你的代理的对象 如果是有接口 那么就生成一个JDK的实例来代理 --- 如果没有实现接口 使用CGLIB进行动态代理

限制就是 protected private方法 不能被拦截 ----- 这个指的是没有接口的情况 接口中的方法都是public的 ------ CGLIB到是可以 但是 意义不大 --- aop拦截内部方法没有意义

所以 spring中定义的pointcut 也就是pointcut里面的切入点表达式 对应的方法 仅仅是public方法【所以 学习aspectj的时候 对于访问修饰符可以写\* 但是从这里面来讲 貌似不可以 仅仅可以是给public的方法 --- 这个地方可以测测】

**只能对于一个对象上的public方法进行拦截 ----- 包装 私有的 受保护的方法都不能做这样的一个操作【检验】**

Spring aop在目标对象之外做的封装 那么这个target bean的**任何内部操作**是不能被aop干扰的 ----- 图中所示

就是 我的spring aop 对这个target bean的methodA methodB 进行了拦截 ----- 但是 当我的methodA内部调用了this.methodB()的时候 实际上 这个内部调用 aop是无法拦截的 methodB周围是无法拦截的 ----- 【老师让这一块自己验证一下】

----- 比如 这个拦截的业务是开启一个事务 transaction ------ 这时候 调用methodA的时候 开启了一个tx 但是 内部调用的this.methodB()是没有事务的 因为 这个事务实际上是在外面的代理类上面

【

举一个例子

对于 aspectj的前置通知 后置通知 都是@Before @AfterReturing 都是 直接给切面的逻辑 就好了 然后配一个@Pointcut 这样其实看不出来 实际上 假设以JDK 动态代理来做这件事 ---- 前置通知 后置通知 实际上 是被做成这样 就能理解是

ISomeService target = new SomeServiceImpl();

ISomeService proxy = (ISomeService) Proxy.newProxyInstance(

target.getClass().getClassLoader(),

target.getClass().getInterfaces(),

new InvocationHandler(){

@Override

Public Object invoke(Object proxy, Method method, Object[] args)

throws Throwable{

xxxxxxxxxxxxxxxxxx –> 前置通知

Object result = method.invoke(target, args);

Yyyyyyyyyyyyyyyyyy-> 后置通知

Return result;

}

}

);

这样看来 使用了jdk的动态代理 如果这个时候 invocationHandler里面的的target就是 PPT中的targetBean 然后 method就是methodA 那么这种前置通知 后置通知 执行都和method.invoke没有关系 ----- 这样 method.invoke(target, args) 传递的仍然是目标对象本身 --- 那么这个方法内部调用的是ppt上说的methodB 但是 执行对象就是target 所以 就没有被aop增强的功能

对于AspectJ中的环绕通知 形式非常类似于JDK本身的动态代理的写法

@Around(“execution(\* \*..ISomeService.doSome(..))”)

Public Object myAround(ProceedingJoinPoint pjp) throws Throwable{

xxxxxxxxxxxxxxxxxx –> 前置通知

Object result = pjp.proceed();

yyyyyyyyyyyyyyyyyy-> 后置通知

return result;

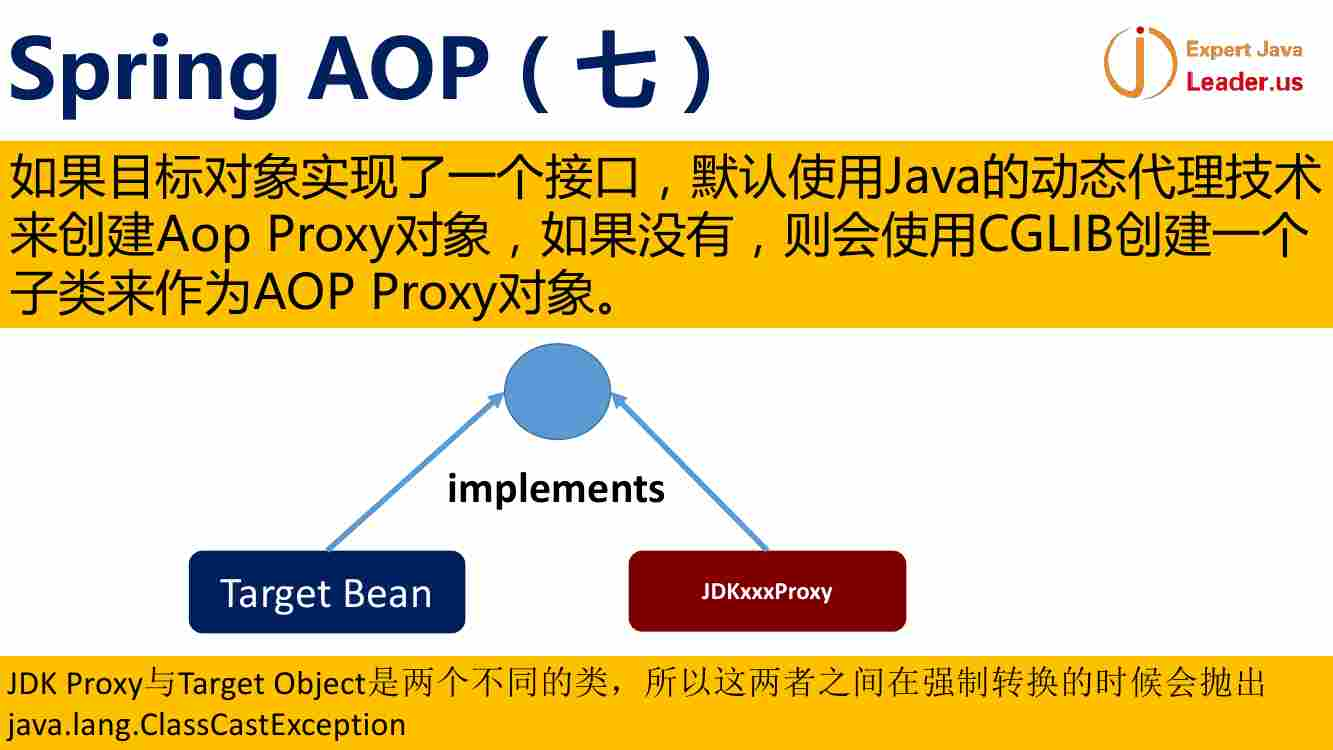
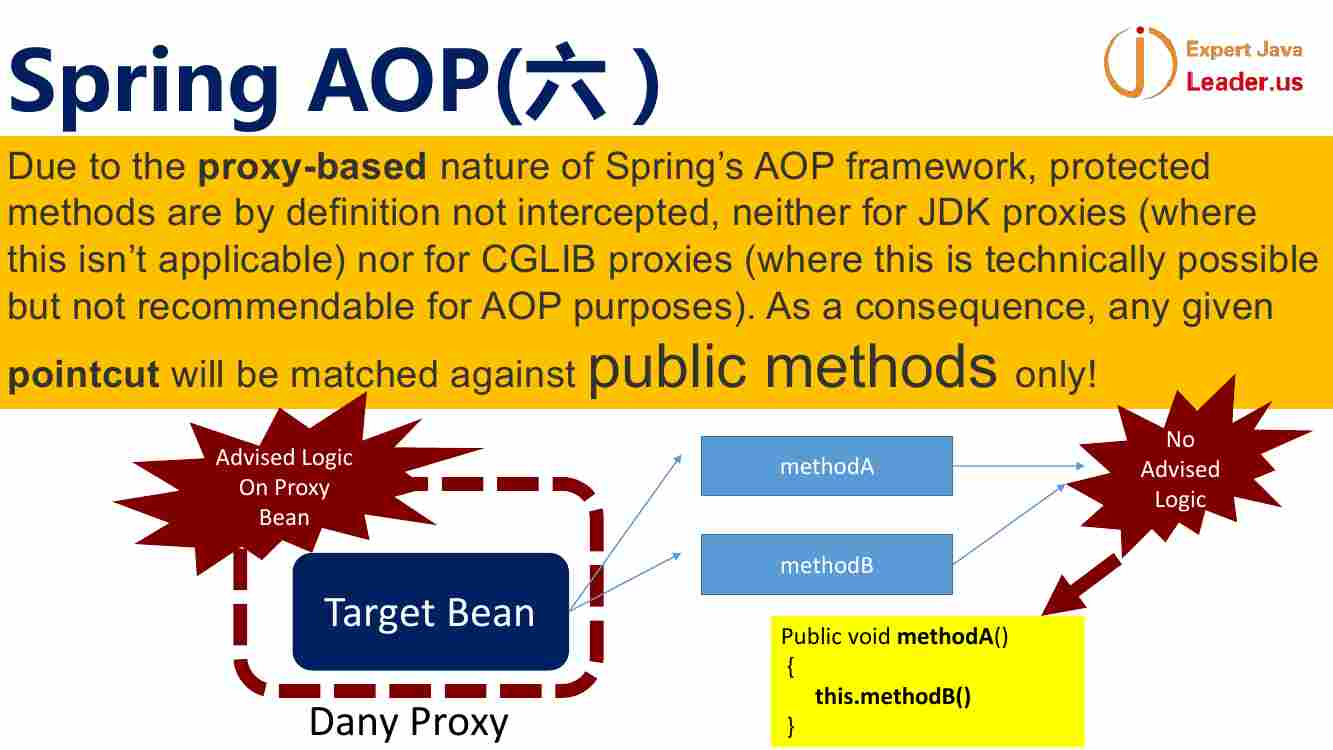
}

如果这个methodA被环绕通知增强了 直接看代码 就知道 我这个pjp和你的增强没有关系 被调用的时候 还是jdk动态代理里面的method.invoke(target, args) 所以 调用到mehodB的时候 还是target本身 所以 。。。

如果想要methodB被增强产生作用的话 必须直接调用外面的代理对象.methodB() 这样 转换成JDK内部的动态代理的时候 才有 被增强的。。。

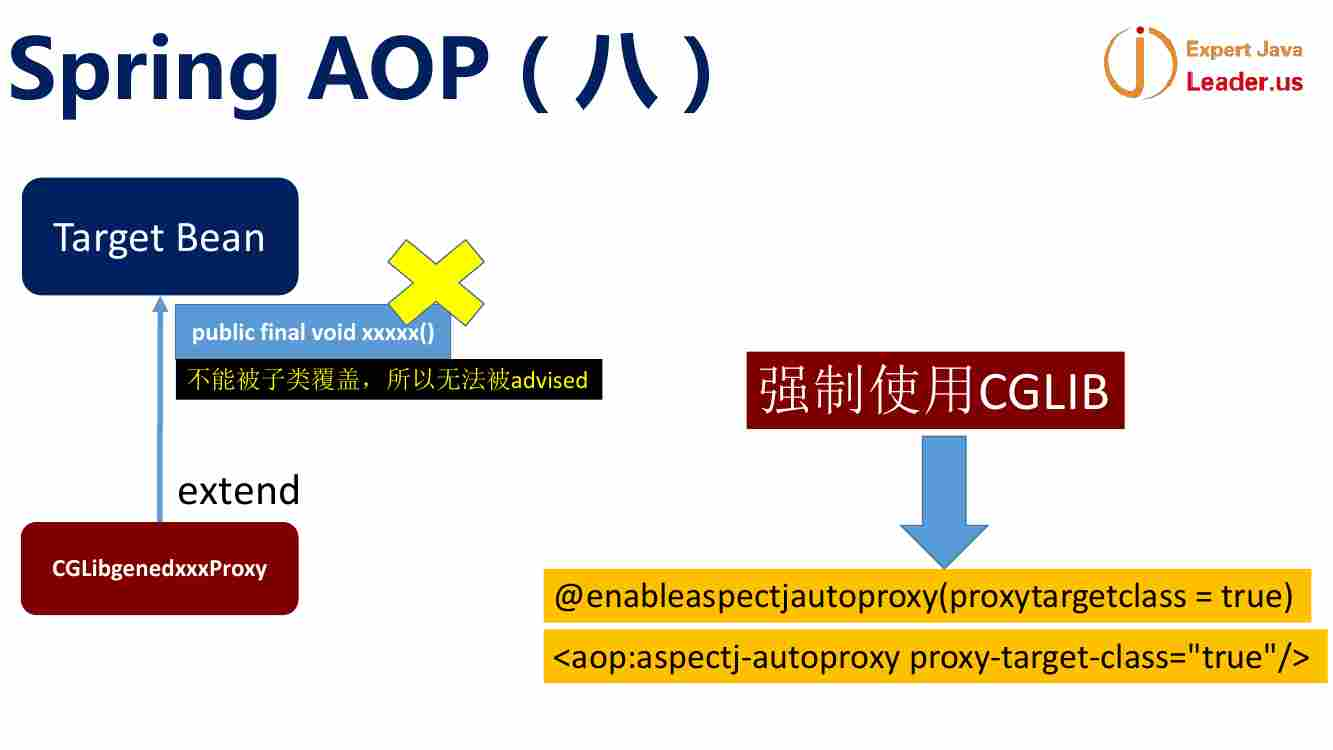
**如果要想源码验证 必须了解java字节码框架 javasisst和asm的用法 然后 再去结合spring + JDK源码**

】



假设发现TB实现了接口 就使用JDK动态代理 生成新类 ---- 返回这个新类 这个JDKxxxProxy和TB是两个不同的类 将至转换会报错 但是 可以把这个JDKxxxProxy转换成向上的接口类【这种接口的uml是另一种 学会】

**【这种 jdk的动态代理的特点 就有一个 父接口 就是原有的接口 ---- 回头看一下原来的jdk动态代理类的特点 里面有源码】**



如果是cglib 没有接口 那么cglib生成一个类 子类扩展使用这个bean 上面的uml就是 CGlibgenedxxxProxy是继承TB的 【这样 和JDK动态代理不太一样的地方 就是 这里面CGlibgenedxxxProxy类把原有的类作为自己的父类 那么 就可以强转成TB 但是JDK proxy是不行的】----- final类不可以被继承 也就是不可以被扩展 所以 final的类不能被aop代理 同样 ---- 也就是final的方法 就不能被重写 所以 也不能被代理

不管有没有接口实现 cglib 都可以 代理的是类 不是接口 -----

强制使用CGLIB的方式

注解：@EnableAspectJAutoProxy(proxyTargetClass=true)

Xml : <aop:aspectj-autoproxy proxy-target-class=true>

【如果对于spring aop自身的做法 使用ProxyFactoryBean的话

配置是这样的

<bean id=”serviceProxy” class=”org.springframework.aop.framework.ProxyFactoryBean”>

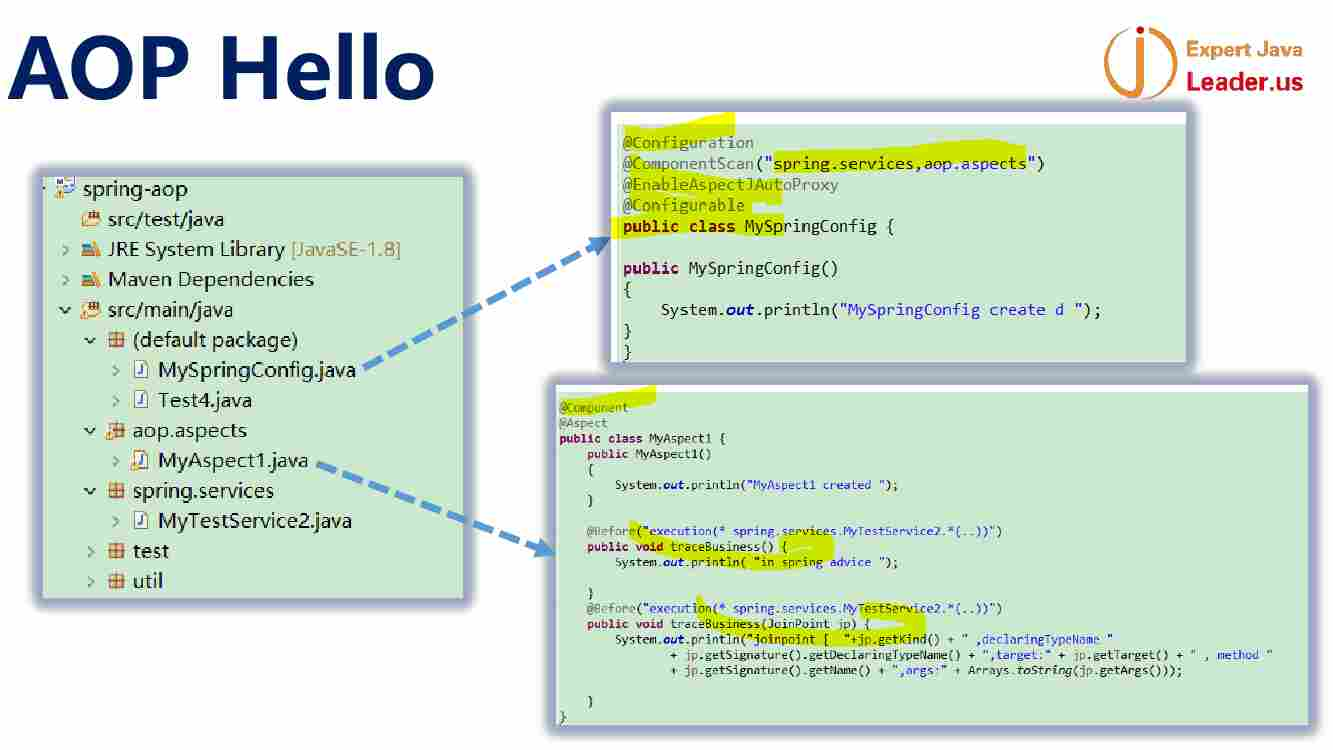
<property name=”target” ref=”someService” />

<property name=”interceptorNames” value=”afterAdvice” />

<property name=”proxyTargetClass” value=true />

</bean>

】



这个是一个AOP的工程

先看一下 这个MySpringConfig 这个里面@Configuration + @ComponentScan(“spring.services, aop.aspects”)【扫描了aspects ---- 在xml中 aspect实际上 是来配置到xml中的 所以 下面的MyAspect是需要@Component进行标记的】【这就是前面老师讲的 我的Aspect可以被装配 特殊的原因 就是 我的Aspect需要被创建出来 ***如果是xml 需要被配置到<bean>中才能生效***】

还有 重点就是 MySpringConfig上面使用了@EnableAspectAutoProxy这个注解

在MyAspect1中

这里面 两个切入点是一样的

execution(\* spring.services.MyTestService2.\*(..)) ---- 这个是对spring.service下面的MyTestService2这个类的所有方法

有两个前置通知需要织入

一个是打印一句话 另一个打印里面的JointPoint的信息 jp实际上就是切入点表达式对象

其实用AOP的时候 可以知道方法调用参数的

===== 注意到 这个MyTestService2这个类 是没有实现接口的 --- 直接是一个类【那么可想 虽然没有配置@EnableAspectJAutoProxy的属性targetProxyClass为true但是 因为没有实现接口 使用的代理还是cglib的】



我们运行这个例子 发现创建出来这个apsect 并且 this是创建出来的代理类 不是MyTestService类

Target是本体 真正的本体



现在看使用由接口的 ----- 运行的时候 从容器中取出来这个bean之后 直接强转成这个bean类型本身 所以 就像前面说的 报错了

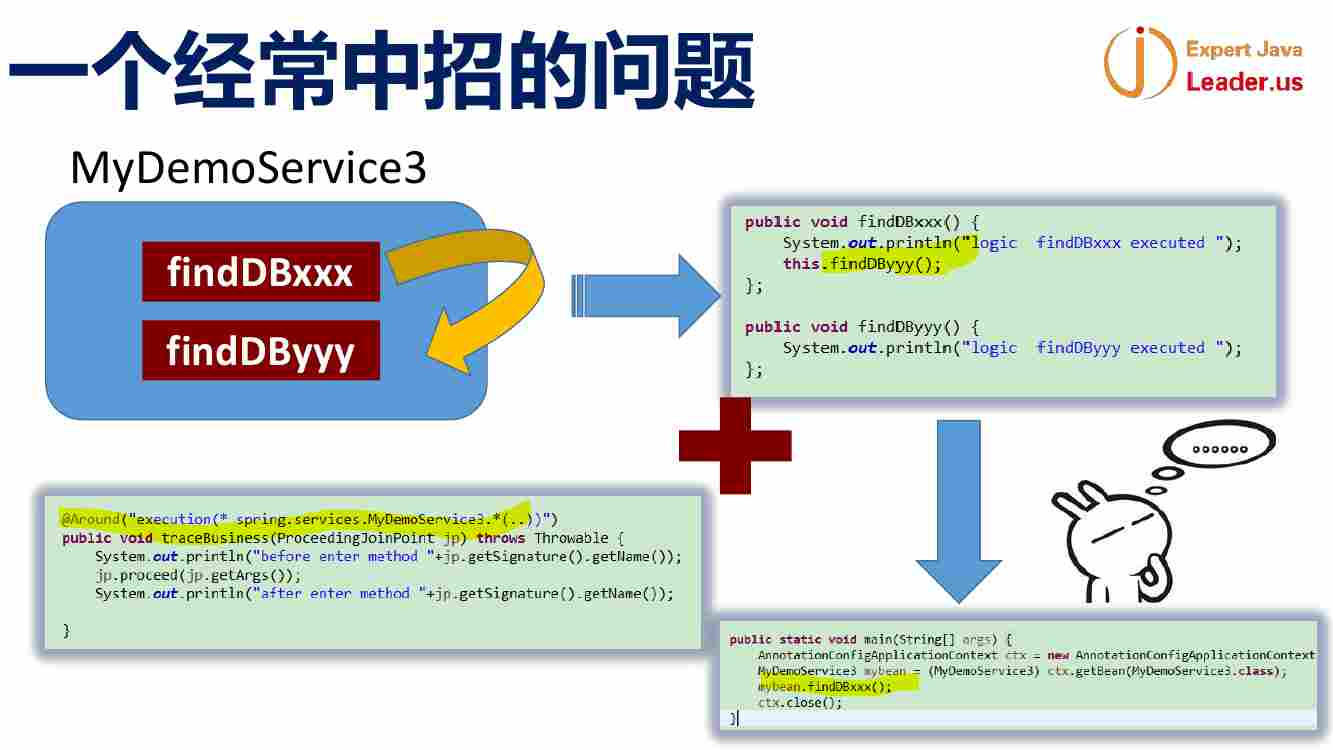
这个是一个新的类 ---- 可以cast成MyDemoServiceIntf



面向接口编程 和 CGLIB都由用武之地

面向接口编程 强调的不多 因为业务的实现类只有接口 这时候 接口是多余的 但是 正规

CGLIB适用性比接口更广

这个实际上 是前面那个图的例子 就是 我有一个环绕通知 环绕了spring.services.MyDemoServices3下面所有的方法 ---- 这些方法里面有findDBxxx和findDByyy 然后 findDBxxx内部调用了findDByyy 问最后外面的输出会怎么样？

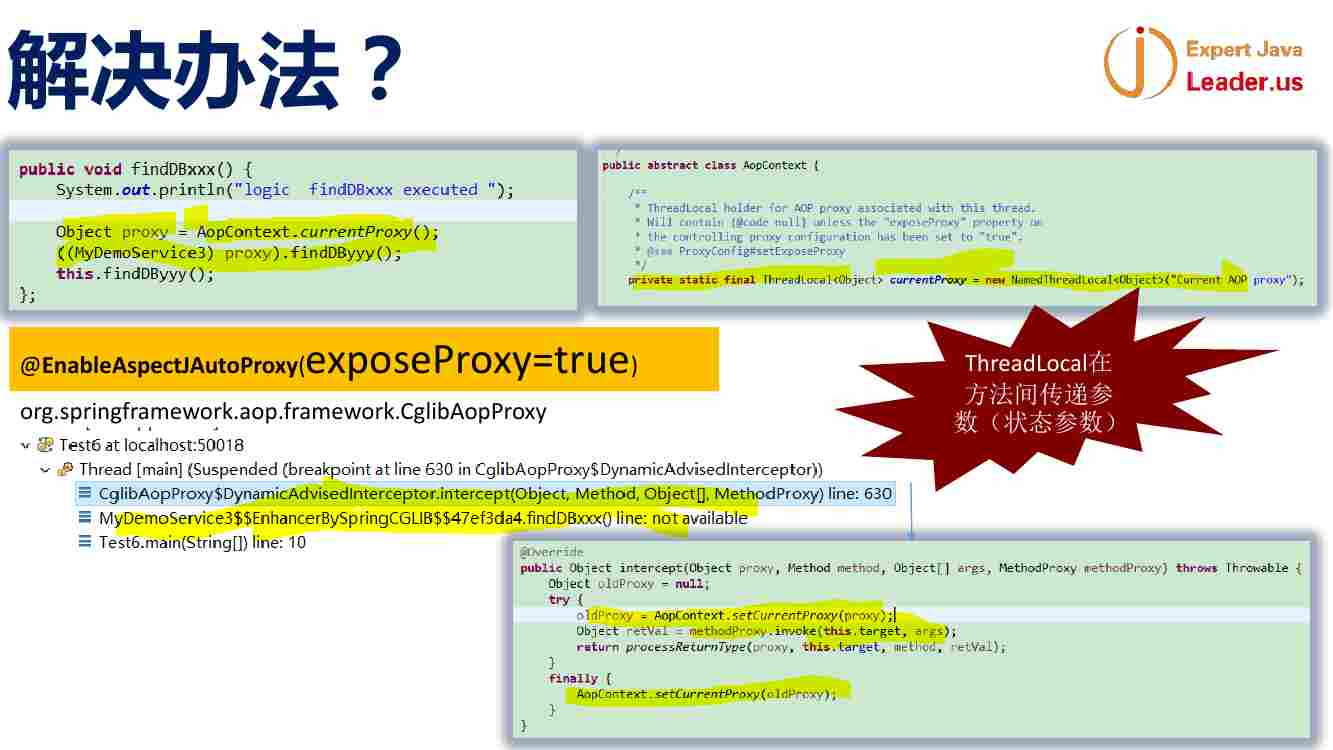


左边是错误的 右上角的答案是对的 原因还是 前面我自己分析的 也就是老师上面画的图

外面调用不影响里面

如果methodA没有事务定义 mb有事务定义 这个时候 逻辑会出错

如果希望我们内部调用的方法也要被代理 有aop的话 做法如下

 【**老师写的红色背景的字 就是ThreadLocal在方法之间传递参数 或者状态参数 ---- 这个就很好的体现了这种用法 ----- 也就是事先把需要传递的参数 放到ThreadLocal中 然后 方法需要这个参数的时候 直接通过ThreadLocal获取**】

可以使用AopContext这个类 完整的类名是 org.springframework.aop.framework.AopContext

因为这个里面有一个方法是静态的 所以 直接使用Object proxy = AopContext.currentProxy();

这样获取到了当前线程相关的AOP proxy **【怎么把这个代理对象放置进去 源码可以分析】**

然后 对这个代理proxy 直接强转成我这个类 MyDemoService3 然后 调用findDByyy() ---- 如果这个是JDK的动态代理 就不能这样强转 要转换成接口 要是CGLIB的动态代理 就没有问题

这时候 发现代码就正确了

要想正确通过AopContext.currentProxy()获取到当前的代理 必须进行配置

@EnableAspectJAutoProxy(exposeProxy=true) ----- 追踪这个注解 里面的代码 实现就是

图中intercept这个方法做的事情【**有必要做源码分析**】

【

解决思路：

因为 我们在methodA中调用methodB的时候 实际上

是thid.methodB();这个this就是target本身 是不能进行aop的根本原因

***那么 如果我能在这个methodA中获取到外面的proxy的话 不就可以直接执行这个methodB了么（前面老师说了 tb和proxy之间正常情况是没有办法彼此相互获取到对方 但是说了 可以用特殊的编程技巧 这里面就是使用了特殊的编程技巧 AopContext 和 ThreadLocal）***

这个类是一个抽象类 -----

public abstract class AopContext {

private static final ThreadLocal<Object> currentProxy = new NamedThreadLocal<Object>("Current AOP proxy");//NameThreadLocal<Local>

public static Object currentProxy() throws IllegalStateException {

Object proxy = currentProxy.get();

if (proxy == null) {

throw new IllegalStateException(

"Cannot find current proxy: Set 'exposeProxy' property on Advised to 'true' to make it available."); //这个异常提示你 如何让这个类中的方法生效

}

return proxy;

}

static Object setCurrentProxy(Object proxy) {

Object old = currentProxy.get();

if (proxy != null) {

currentProxy.set(proxy);

}

else {

currentProxy.remove(); //为空 表示 我传入了一个空的代理 就是不想设置代理了 这时候 就是吧代理删掉

}

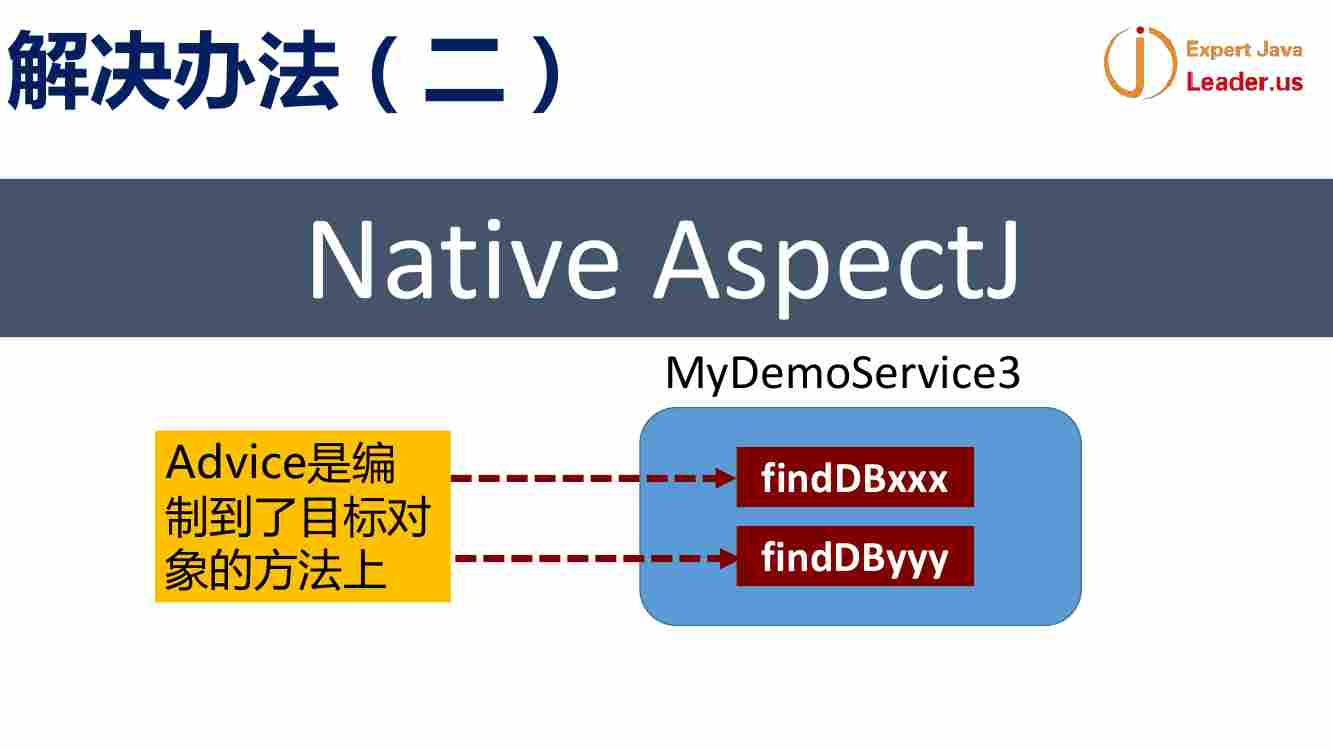
return old;

}

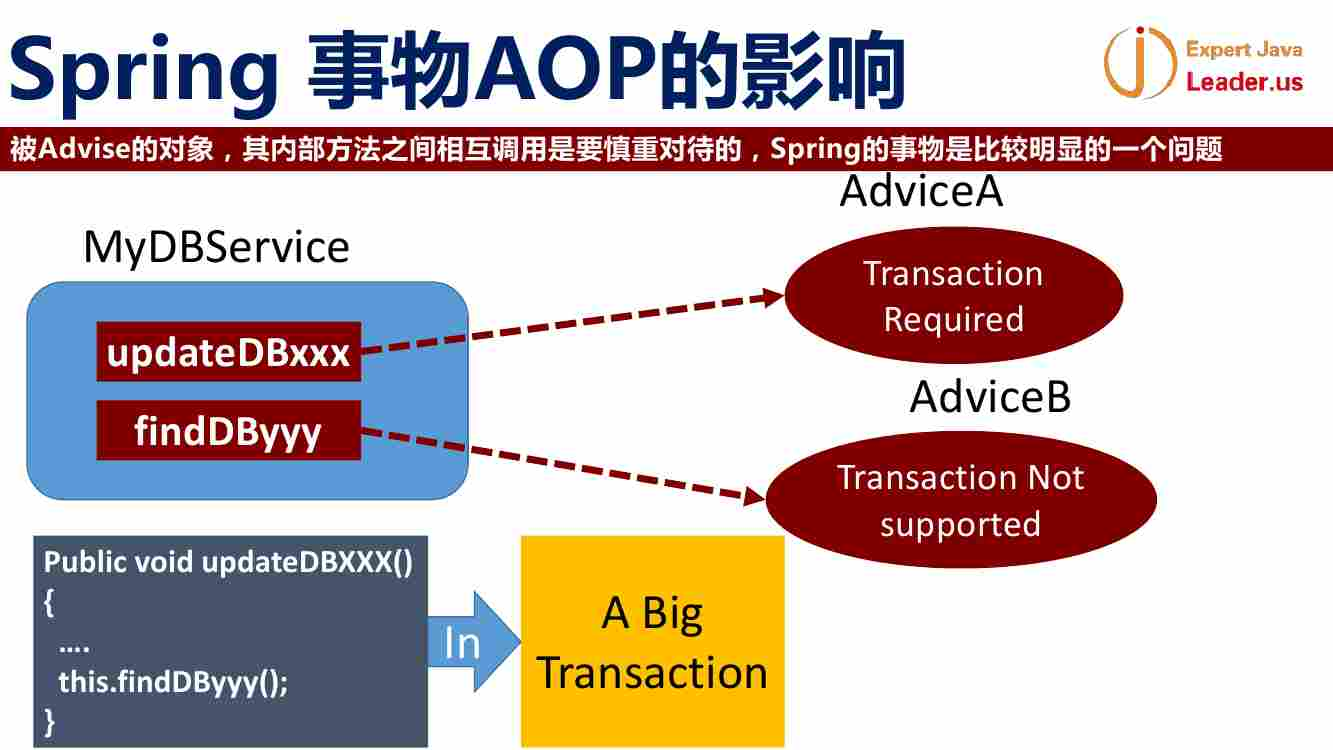
}】

第二种解决方法 就是 直接使用Native AspectJ

【***再次追踪一下上面的情况发生的根本原因 就是 用了代理 导致 tb和proxy不是一个 但是 前面说了 aspectj会把代理的增强内容 也就是advice直接织入到tb的方法本身 ----- 也就是tb和proxy是一样的 这样 我直接使用aspectj本身 不就对了么 而不是使用spring aop这种动态代理的机制***】



因为aspectj直接在tb的方法周围进行advice的织入 相当于把运行期的代码进行了修改 所以 是比较复杂的 也不符合solid大法 改的是class本身 --- 相当于装饰 -----不建议这么做



总结 的就是 被Advise的对象 在内部方法之间调用是要慎重对待的 Spring的事务是比较明显的一个问题 【事务 不是事物 ppt又写错了】

AdviceA 的时候 是update 需要事务 但是 find方法 findDByyy是不需要事务的 ----

这样 两个方法被AdviceA AdviceB 分别进行织入

但是 如果你的update的时候 需要调用findDByyy() 这样 被AdviceB织入的Transaction Not Support就会失效 此时 this.findDByyy就会纳入到updateDBxxx()的整个事务中 ----- 就会和想象的逻辑不符合

这个时候 不要使用this.findDByyy() 像刚才那样 使用外部的proxy 使用AopContext中的ThreadLocal进行aop本身的传递就可以了



【PointCut表示的就是切入点 所以 这里面指的切面表达式】

现在有一个接口 泛型接口Sample<T>

这个泛型接口里面有两个方法 一个方法 是 泛型方法 sampleGenericMethod(T param);

另一个是 集合类型的泛型方法 sampleGenericCollectionMethod(Collection<T> param);

【+放在接口后面 表示当前接口机器实现类】

PointCut的切入点表达式 可以命中到对应的接口 （where问题） 【注意 @Before可以把args这个参数写入value属性中 也可以单列出来】 ----- 这样 写成 execution(\* ..Sample+.sampleGenericMethod(\*)) &&args(param)  后面的args中指定的param就要把参数传递进来 这个原来本身是泛型的T param 这个时候在Aspect类里面就没有定义了 务必要进行转换 可以使用Object 也可以直接使用具体类型进行替代 这里面就是实际的MyType

但是 下面的集合方式是不行的

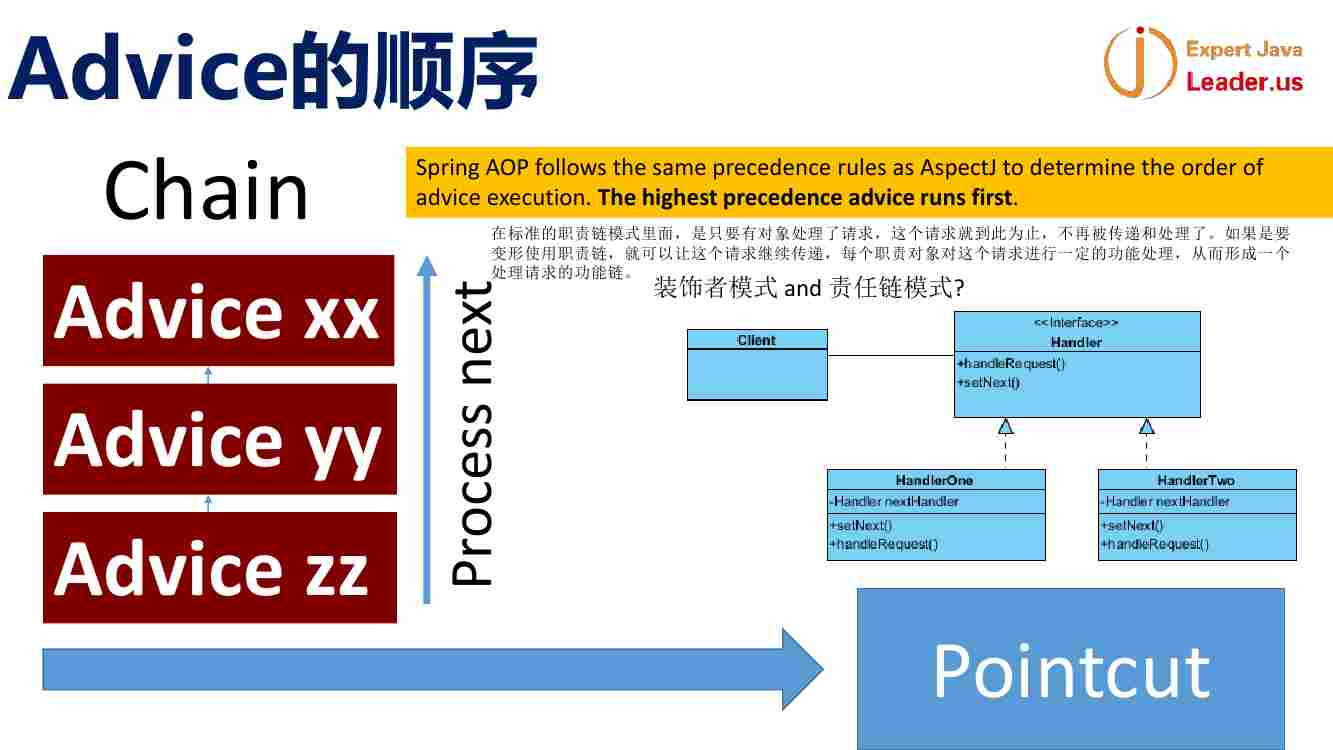


这个地方是要对point cut进行共享 做一个最佳实践

在一个Aspect中 定义一个Web Layer层 和一个 service layer层的PointCut 两个切入点 然后在其他地方进行使用

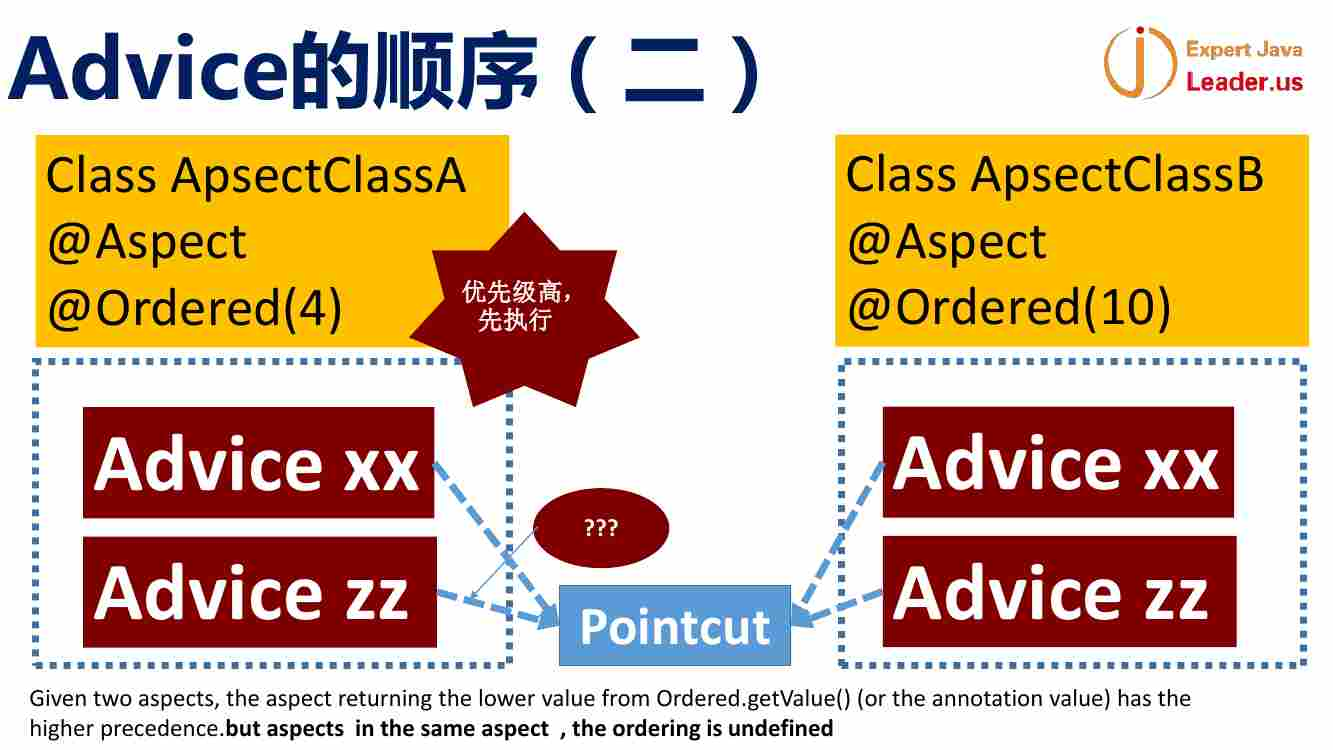
后面那个是xml中的方式

Xml用的少了

【看看源码 是不是责任链模式】

一个target会被多个advice进行织入 多个advice就组成了链条模式 --- 标准责任链过程中 某一个请求就结束 但是 这里面使用了变形的责任链【cc讲的设计模式有这个 可以在ppt中说出来】 ----- 变形的责任链就是 所有的都处理 – spring中的责任链条很多都是用这个变形的责任链 如果是查找 就会使用标准的责任链 找到就结束

Advice就是所有的都要执行 执行顺序根据Advice定义的order --- 优先级越高 越先执行



定义成两个不同的Aspect 不同的优先级 ---- 数值越小 优先级越高 越先执行

【图中 就是 Aspect ppt单词写错了 Apset AspectClassA B中分别有两个advice要织入到Pointcut 也就是目标类的目标方法中 根据@Ordered中的数值决定哪个先执行 内部的Advice按照书写的顺序执行】

【做例子说明】

但是 在一个Aspect中定义了多个 这个顺序就不确定 ---- 因为方法先后顺序不确定 ----- **一般是按照字母顺序**

如果要存在不同的切面 装同一个pointcut 建议放到不同的Aspect中 否则 就会顺序不确定【可以查看源码是按照什么顺序来的】