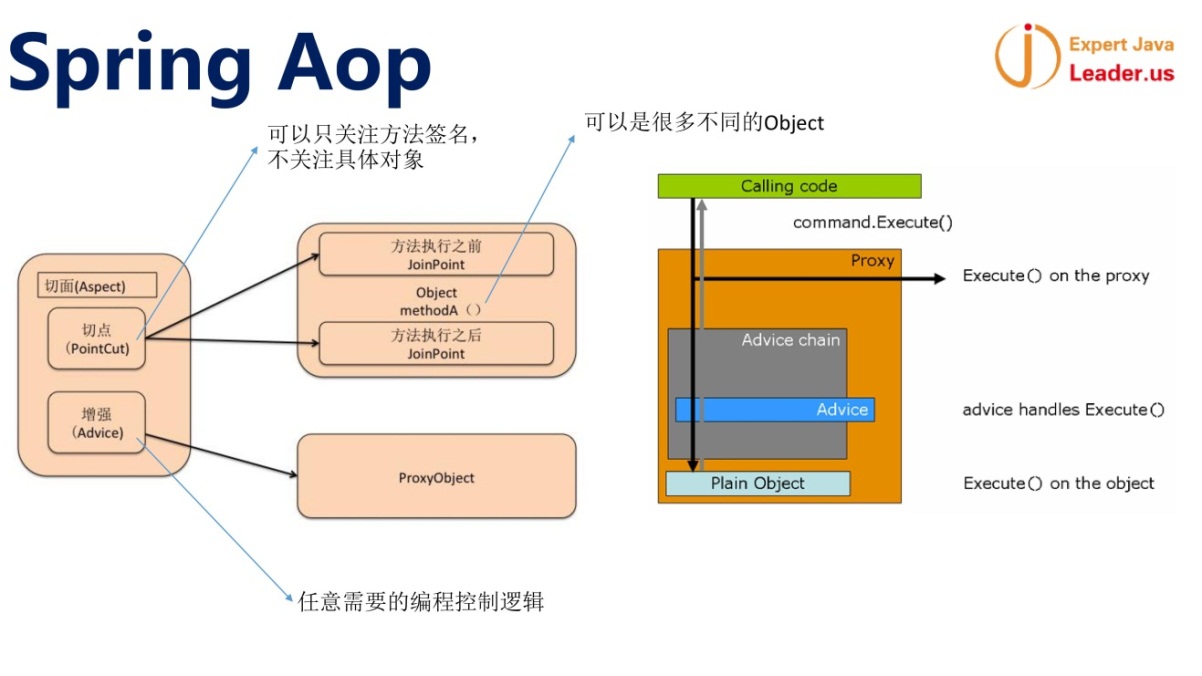


Spring AOP和标准的AspectJ Aop有区别联系 做的比较复杂 **同时为了兼容 做了很多包装**

基本的概念 Advice Advisor TargetSource Advised

**【这个图和struts2的很类似 ---- HW 和Struts2的图进行对比】**

就是在你关注的点上pointCut 做一些增强 ----- 切点在Spring中只能用于方法 方法前 方法后 环绕增强

增强的内容在Proxy对象上面的 ---- 增强的逻辑可以编程控制 ----- 另外在哪些pointcut增强可以灵活控制

**右边给出执行AOP过程 先进入proxy 然后 构建一个Advice chain ----因为可以构建多个Advice 组成一个链条 执行完成以后 才调用具体对象的方法 ---- 这个方法是JointPoint的一个方法点** (继承关系是 JointPoint<- Invocation <- MethodInvocation )---- 最后执行一个target的具体的方法 然后再返回 难就难在这 【非常类似于struts2中的拦截器】



**Advice在AOP联盟中是定义了一个空接口 --- 也就是 标记接口 真的实现有几种**

--- **认为Advice就是拦截器** ---- 之前的代码的确是拦截器

----- 所以 中间的类图 Advice下面的第一个子类类名就是Interceptor 就是拦截器 ----- 本来 这个Interceptor在AOP联盟中 有两个实现类 一个是MethodInterceptor 另一个ConstructorInterceptor 但是 构造器的Interceptor在Spring中用不到

所以 在spring中类图就简化 为 Advice <- Interceptor <- MethodInterceptor

**也就是一个Advice对应一个方法发拦截器MethodInterceptor ---对public方法的拦截**

**继承关系中 对一个Advice 就是对某个方法的拦截 MethodInterceptor --- 类似于Servlet中的filter**

------- 这个MethodInterceptor子类中 有一个方法 就是invoke 接口----

**Object invoke(MethodInvocation invocation) 这个你先执行你的拦截器的内容 然后 在调用目标invocation** --- 这个就是目标对象的方法 【非常类似于struts2中的过程】(继承关系是 JointPoint<- Invocation <- MethodInvocation )

----- **MethodInvocation** 也是**MethodInterceptor**依赖的类 就是这个invoke方法中 MethodInterceptor依赖了MethodInvocation这个接口

----- **拦截器本身可以是链条 ---- 所以 这里面MethodInvocation不一定是target的目标方法** 可能是责任链中传递的下一个Interceptor 未必是最终的目标方法 ----- 所以 MethodInvocation中的getMethod() 未必是最终的目标方法 而很有可能和上一个拦截器相关的 ----- 所以 这个invoke方法的参数不是Method 而是MethodInvocation

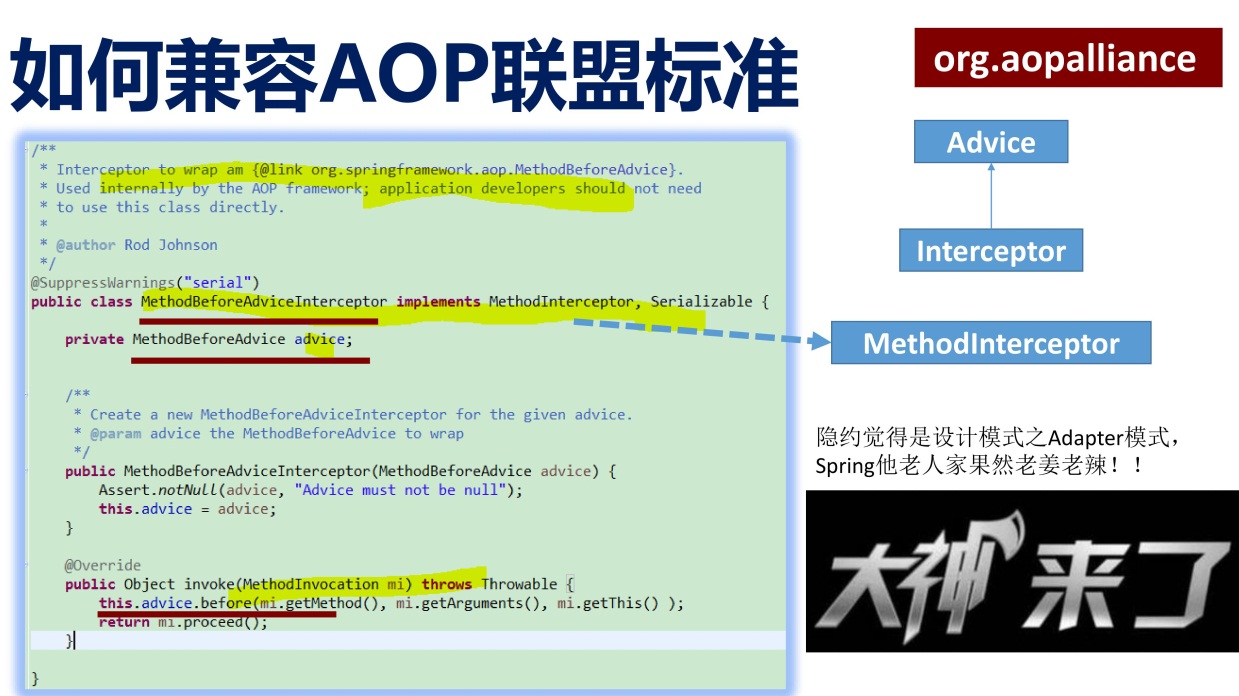
上面是正宗的AOP联盟的过程 【struts2的也是这个过程 ppt上可以对比一下】

但是 在spring aop中 直接从Advice这个顶级接口实现了一个**MethodBeforeAdvice这个类 ---- 从这个方法 void before(Method method, Object[] args, Object target) ----方法非常清晰化 参数非常清晰化【类似于JDK Proxy的说法】 ---- 接口设计的非常精确 简洁 ----- 比AOP联盟更清晰**

所以 在Spring中Advice就是MethodInterceptor ----- 然后Advice构成一个链条 一个一个传下去 ----- 那么Spring就必须朝着这个靠拢

**【从图中的箭头看出来 Advice Inteceptor和MethodInterceptor这三个是继承关系 Advice是接口 那么剩下的两个也是接口 ----- Spring的MethodBeforeAdvice是虚线箭头 ----画错了 应该是实线箭头 说明MethodBeforeAdvice是Advice接口的子接口 ----- 有一个自定义的before方法 ---- 层次关系上 MethodBeforeAdvice和Interceptor是一个层次关系 比MethodInterceptor高一个级别】**

Spring如何靠拢？？



如何靠拢的AOP联盟的？

Spring aop实现了一个MethodBeforeAdviceInterceptors实现了AOP联盟的MethodInterceptor接口 -----> 然后内部持有了一个MethodBeforeAdvice

**【一旦持有了一个对象 就可以对这个对象进行任意的操作】**

----- 整个MethodBeforeAdvice的构造过程就是传递一个MethodBeforeAdvice就可以了 然后 就可以构造出整个的MethodBeforeAdviceInterceptor 【实际上 挂钩到aop联盟的MethodInterceptor之后 就意味着和对应的aop联盟靠拢了 ---- 靠拢之后 需要把已经靠拢的MethodBeforeAdviceInterceptor和自己的MethodBeforeAdvice进行关联处理 ----- 因为spring aop编程的时候 就是直接使用MethodBeforeAdvice这个直接父类】

*void before(Method method, Object[] args, Object target) throws Throwable;*

(继承关系是 **JointPoint<|- Invocation <|- MethodInvocatio**n )

----- 在实现父类MethodInterceptor的invoke方法的时候 就是使用this.advice.before(mi.getMethod(), mi.getArguments(), mi.getThis() ); 【相当于执行了前置通知】 ----- return是返回值 mi.proceed(); ------这个就是使用mi直接调用返回结果【源码里面 点击mi 直接跳到jointPoint这个类里面去了】

------ 这样一做 就符合Aop联盟的标准了

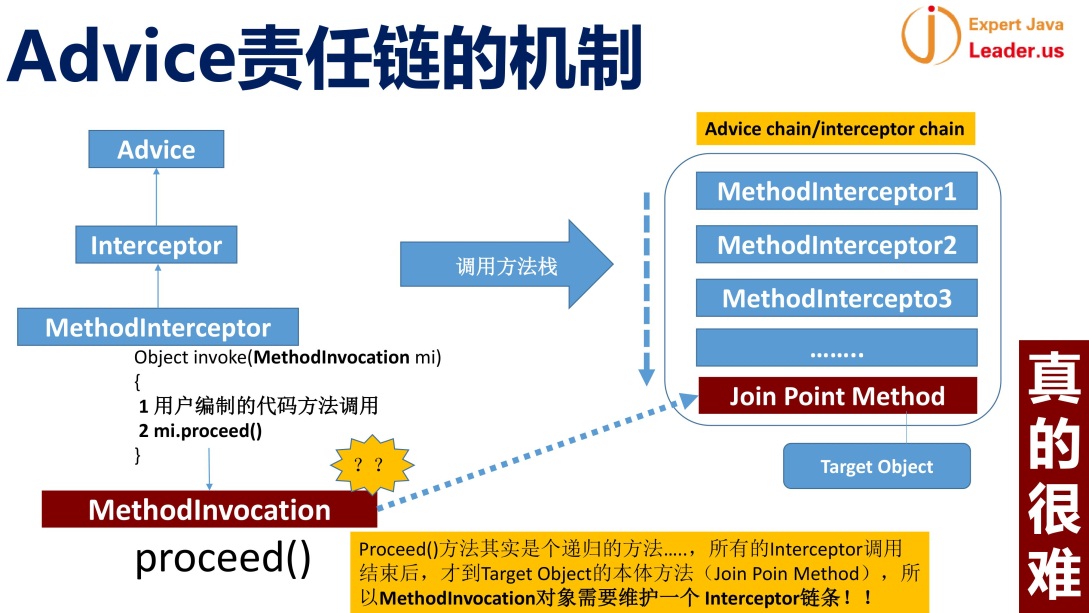
======= 实际上 这个是一个Adaptor模式 希望让自己定义的一套 和AOP联盟的标准进行兼容 所以 这个是Adaptor模式

【***对应到适配器模式中 要执行的是责任链这套规则 那么 现在 我们有的是MethodBeforeAdvice这个接口 用户来重写*** 但是 在整个流程中 MethodBeforeAdvice需要融入到整个责任链过程中 ---- aop联盟的责任链的节点实际上是MethodInterceptor类型 实际上 标准的责任链模式 有一个抽象的父类的HandlerRequest节点 ------ 这样的话 我就要给一个实际的实现类 来实现这个标准的抽象类 就是接口MethodInterceptor ------ 实现他这里面的方法 ------ 所以  **到Spring aop这里 MethodBeforeAdviceInterceptor就是职责连的真实的具体执行节点** ----- 为什么会有MethodBeforeAdvice呢？ 是设计者Rod为了让使用Spring的人有更好的用户体验(历史遗留 开始这么设计的) 所以 就如leader前面所说的 更容易让用户理解使用 ----- 所以把这个advice真正的核心逻辑 放到了MethodBeforeAdvice中 ------ 既要满足用户良好的用户体验 又要满足已有的执行流程 如果这两个接口有矛盾 ----- 适配器模式解决 --------- 适配器模式的要诀就是 谁被持有 谁就是Adaptee ---持有的类就是Adaptor ---- Adaptor满足已有的流程 并且 **适配用户体验良好的MethodBeforeAdvice ------ 所以 MethodBeforeAdviceInterceptor就是Adaptor ------ MethodBeforeAdvice就是Adaptee =------ 用户体验 和 适配到AOP标准两不误**】

**【因为是功能链 所以 一定要真正执行每一个HandlerRequest的具体方法 然后 放行】【这块要具体研究一下功能链的写法】【已经研究完】**

我们的Advice 实际上回到流程里面 要么是MethodBeforeAdviceInterceptor 要么是AfterReturningAdviceInterceptor 要么是AfterAdviceInterceptor

【我们这里面 还有一个就是MethodBeforeAdvice实现了AOP标记接口Advice这个接口 ---- 现在没看到用处 我估计是语义需要 都是Advice 繁盛Advice就是一个标记接口 实现了 也不用真的实现什么具体的方法 ----- 既符合了AOP联盟的要求 **又保持了自己的既定事先 ---- 因为Advice空接口 所以 直接实现不需要代码改动太多 ---- 还是一个两不误**】



Advice复杂 在于责任链 是一个链条 ----- methodInterceptor的时候 首先 1. 用户编制的代码的方法调用 2. Mi.proceed() **【实际上就是执行拦截的内容 然后放行】 ----- 这里面就是涉及到了MethodInvocation的proceed()的调用**

难点在 Advice是一个链条 对一个Target Method上面 上面是一个链条 ---触发的时候 就不是本体的 而是右边的图 就是从上向下执行 到Joint Point Method ----- **如果有around 还会再一层层返回 反向执行**

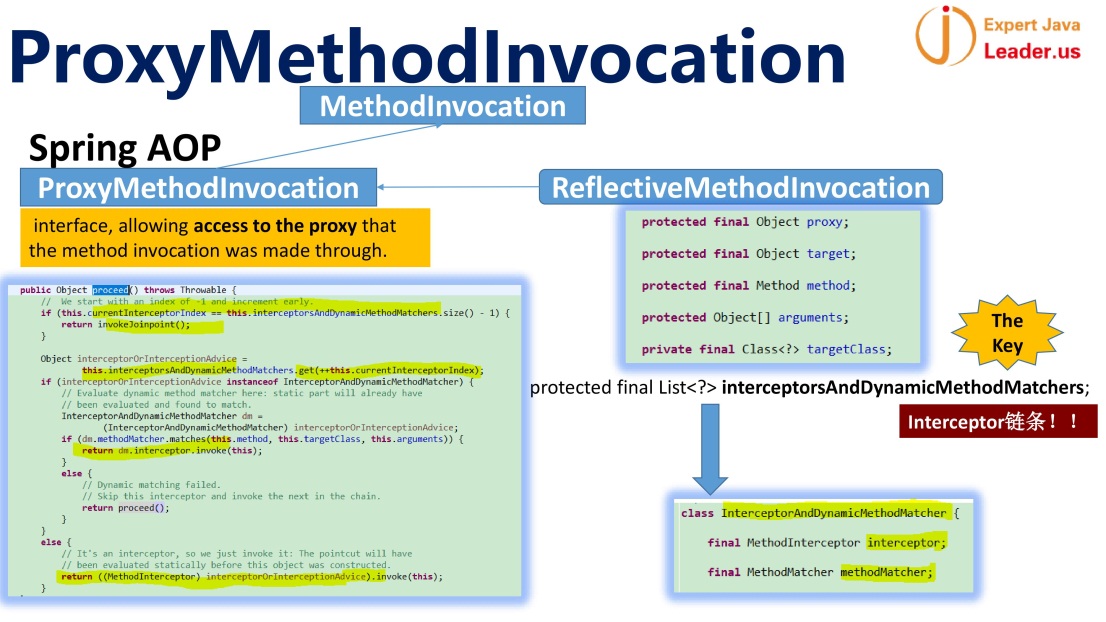
这个地方是一个是一个栈式调用

**----- 所以 这个MethodInvocation的调用 是一个递归的调用 有难度 ----- 所以MethodInvocation接口 比MethodInterceptor接口复杂很多 需要搞一个链条处理的功能**

【老师说 这个methodInvocation要维护责任链的链条 -------- **这个应该是cc老师说的内部链** 】

**----- 特点就是 MethodInterceptor触发MethodInvocation的mi.proceed()的调用，同时反过来 MethodInvocation里面又维护了methodInterceptor里面的链条**

整个过程是复杂的编程【看看能否给一个内部链的Demo 根据CC老师讲的责任链】



Spring AOP对这个MethodInvocation做了一个扩展 【\*\*\*\*\*\*\*\*\*\*\*前面是Spring AOP对methodInterceptor做了一个扩展之后 适配了用户体验有好的MethodBeforeAdvice (这样Spring自己的Advice就变成了AOP联盟的Advice ---- 这个绝对是看起来好看 Advice本身就是空接口)

----- 这里面又扩展了MethodInvocation （因为MethodInvocation里面的链条是AOP联盟的链条 想要工作 Spring AOP也要适配 ----- 因为数组形式的责任链有两部分 一个是RequestHandler 另一个是RequestHandlerInvocation ---- 这样Spring先适配自己的MethodBeforeAdvice为MethodBeforeAdviceInvocation 实际上就是Interceptor 然后 做了一个可有可无的 直接实现空接口Advice =🡺 RequestHandler部分适配完毕，并且自己的Advice别人一看也是实现了Aop联盟的接口的； 另一方面 就是 毕竟自己的Advice Advisor 工作原理都和AOP联盟的不一致 所以 为了责任链能够跑通，适配AOP的RequestHandlerInvocation ---- 所以 有 ReflectiveMethodInvocation-|>ProxyMethodInvocation-|>MethodInvocation-|>Invocation-|>JointPoint）----- 看了下面以后 知道 我们Spring的aop是基于代理的 但是AspectJ是直接在class级别修改原有的target的】

**第一步的接口 就是 ProxyMethodInvocation 为了得到代理**  也就是可能在***target执行过程中需要得到proxy对象*** ----- 这样 通过代理类里面得到这个proxy对象

【继承关系是 **JointPoint<- Invocation <- MethodInvocation <- ProxymethodInvocation <。。。ReflectviceMethodInvocation** 最后一个这个是实现类 前面的都是接口 】

真正实现ProxyMethodInvocation的是ReflectiveMethodInvocation **这个类里面 proxy target method argument targetClass属性**

**并且ReflectiveMethodInvocation中 维护了一个protected final List<?> inteceptorsAndDynamicMethodMatchers; ------ 这个就是前面说的维护的职责链 ---这个例子 就是链条**

我们看一下这个InterceptorAndDynamicMethodMatcher

class **InterceptorAndDynamicMethodMatcher** {

final *MethodInterceptor* **interceptor**;

final *MethodMatcher* **methodMatcher**;

public InterceptorAndDynamicMethodMatcher(MethodInterceptor interceptor, MethodMatcher methodMatcher) {

this.interceptor = interceptor;

this.methodMatcher = methodMatcher;

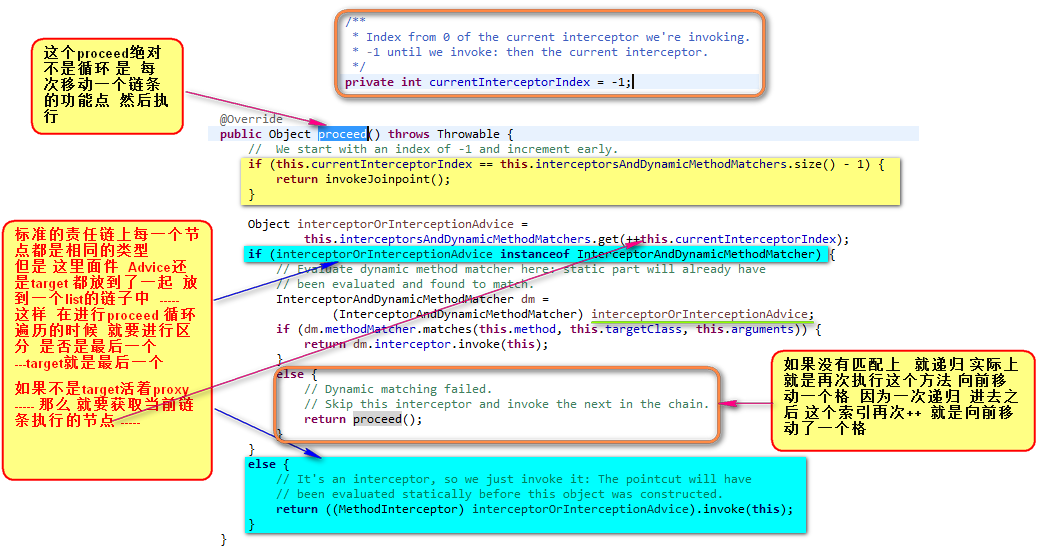
}

}

这是跟这个target相关的Advice的链条 ------- Advice -----在这里面 就是interceptorsAndDynamicMethodMatchers ---- 类型 实际上就是InterceptorAndDynamicMethodMatcher ---- 这个类里面就两个属性 一个是MethodInterceptor 【找到了MethodInterceptor 就是关键点了 前面说了 MethodInterceptor驱动了MethodInvocation的proceed() ----- 然后 MethodInvocation里面又持有了MethodInterceptor这样的一个链条 ----- 找到了interceptorAndDynamicMethodMatcher 就找到了MethodInterceptor这个类 就找到了对应的链条】 另一个是MethodMatcher ----- MethodMatcher就是是否匹配的这个方法本身 ----- 动态匹配的 不一定匹配到这个本身【我认为是责任链模式中 每一个节点处理这个request的时候 就要看是否满足对应的条件】 ----- 运行过程 会检查 **当前这个Interceptor是否符合object的调用 【也就是MethodInterceptor是责任链上的一个点 是否要执行呢 需要判定 责任链里面还是有这个判断的 就要看看功能链中是否有这个 现在spring把这个封装成一个MethodMather 再和MethodInterceptor合并 封装成了一个InterceptorAndDynamicMethodMatcher】**

这样扩展以后 这个Reflective就包含了 proxy和当前对象 以及整个的责任链条 【正常的责任链 是客户端构建的 那么 这个ReflectiveMethodInvocation就相当于标准责任链中的client 构造了链条 ------ **这样 目标对象 + 代理对象 + 责任链条 这样一个数据结构体** 就是ReflectiveMethodInvocation ---- **很完善** ***（估计在MVC中的责任链 就是 目标对象 + 责任链条 也就是没有代理对象了） ------ 所以 叫方法调用的含义 就是整个链条 的内容都在这 所以 整个调用就是非常清楚的 ---- 所以 一定有一个类 来调度这些内容***】

所以 看这个ReflectiveMethod

【**就是 要学习这个调度职责链的思想 这个职责链驱动方法就是一个proceed() 一定是外部给力**】【**要追踪责任链模式的return的值的用处 ------** 】

【**注意这句代码： dm.methodMatcher.matches(this.method, this.targetClass, this.arguments) --- this是ReflectiveMethodInvocation的属性 就是 有proxy 有class 有arguments 有target ------ 没有Advice ====> 我这个对象是否是链条中的对应的Interceptors? 下面的代码中 就是 给出的list<Object> 这个 里面 的链条是这个Advisor的链条 ---- Advisor可能对应的目标 代理 类是不一样的 ---- 但是 整个的Advisor链条都在一起 下面的代码就有 ---- 大杂烩 所以 后面才会带上MatchMethod!!!!!! ====== 非常重要 ===== 所以 你进行功能链迭代的时候 一定要判断 这个是不是你的！！！！ 重新理解Pointcut里面的表达式的每一部分 非常关键**】

如果已经到头了 当前所有的interceptor 就invokeJointPoint() ----- 就是调用整个proxy target

**protected Object invokeJoinpoint() throws Throwable {**

**return AopUtils.invokeJoinpointUsingReflection(this.target, this.method, this.arguments);**

**}**

------

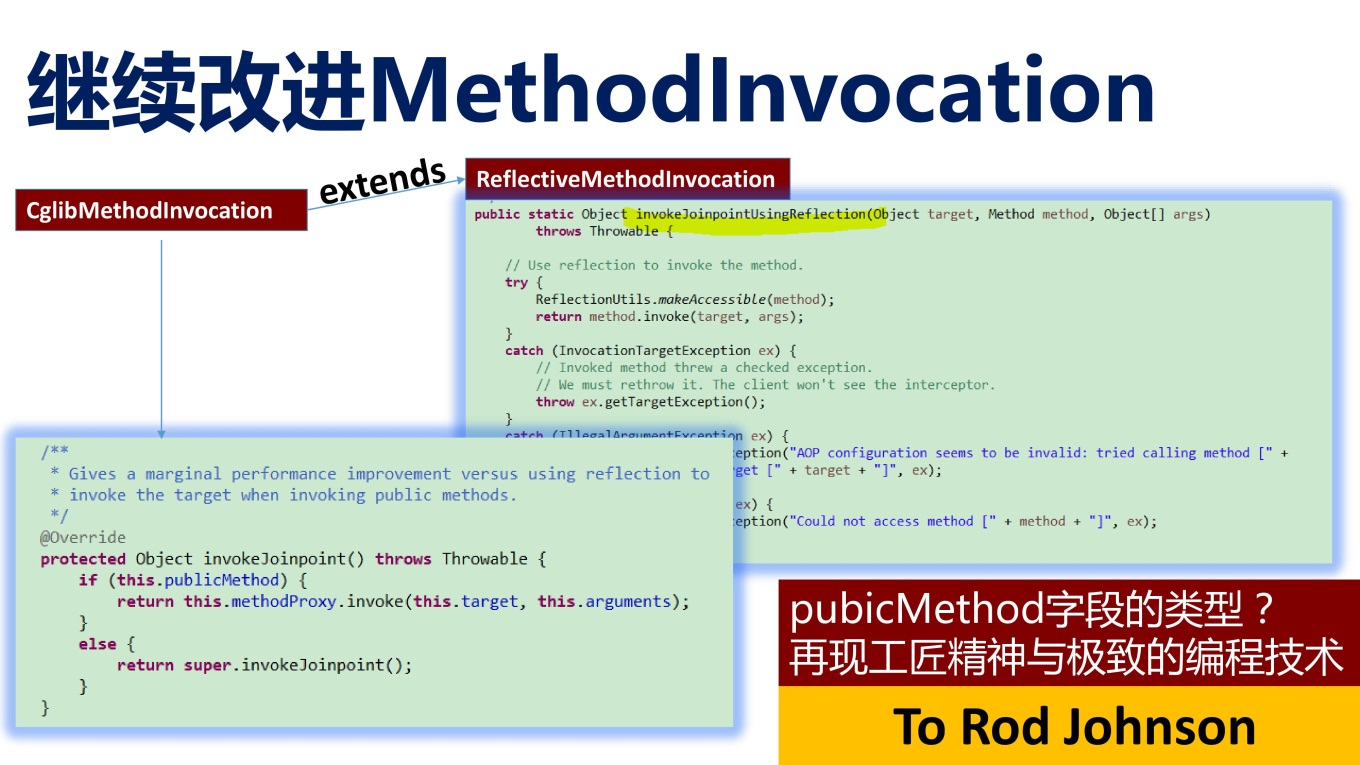
***如果不是最后一个 就从链条中获取一个 然后 发现是不是InterceptorAndMethodMatcher 如果是 就执行 不是 就是methodInterceptor ------ 说明链条中未必都是同一个元素 InterceptorAndMethodMatcher ---- 还可能直接是MethodInterceptor 【所以 这个责任链的组装是关键】 ------ 既可以满足动态匹配的 又是满足一个必须执行 所以 这个方法把所有的情况都考虑到了【老师并没有讲清楚 要自己思考】***

我们看一下这里面件的invokeJointPoint()



这里面是先makeAccesible() ----- 然后invoke() 反射调用 ----- 这样会有一个性能损耗 ----- 所以 下面的CglibMethodInvocation这个类里面就对此做了优化

**如果构造RefllectiveMethodInvocation ----- 调用proceed() 就完成了AOP的拦截 ---- 这句话非常关键**



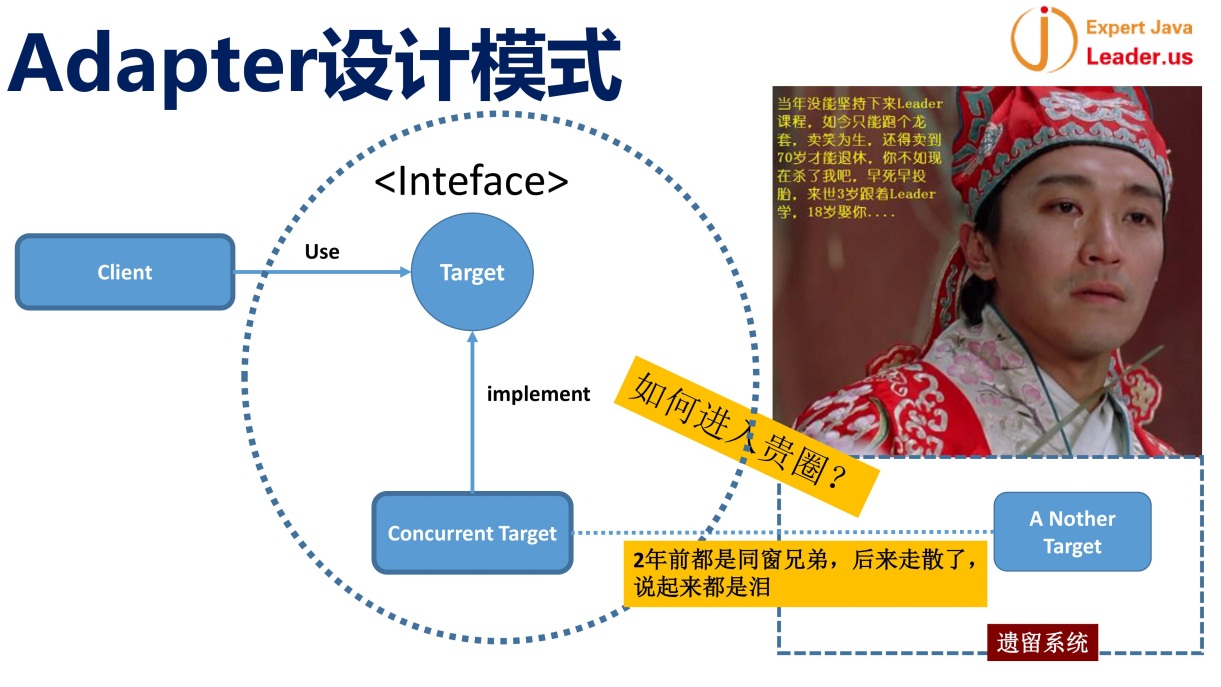
**Spring中又做了改进**

Cglib的mi又扩展了这个ReflectiveMethodInvocation ------ 扩展了一个点 invokeJointPoint() ----- 执行到目标对象自身的方法的时候 做了扩展 就是 publicMethod的话 直接invoke ------ 这里面 如果是publicMethod的话 那么直接invoke 不在makeAccessible 这样有性能损耗 ----- 同时 这个this.publicMethod是一个final的 JVM会对final进行优化 一旦确定就不变 ------- 因为 如果是public方法 如果是第二次在执行的时候 这个if就没有了 else也没有了 直接是**return this.methodProxy.invoke(this.target, this.arguments); 加速性能 --- Spring作者很强 很极致的编程**

所以 只要构建除了ReflectiveMethodInvocation ---- 主体的工程就完成了 这个就是SpringAOP的核心 【需要根据一个例子 去追踪这个链条的构建过程 看看链条中有哪些元素 从执行的角度 来看这个链条的执行过程】

**【重点理解methodInvocation的结构 ----- 其实我理解是 本应该所有的Advice链条都应该放到代理类中 ----如果是静态代理 这个还有可能做到 但是 因为是动态代理 没有办法把这个链条写到动态代理类中 ------- 所以 出现了一个第三者的类MethodInvocation 来代替这种不足】**

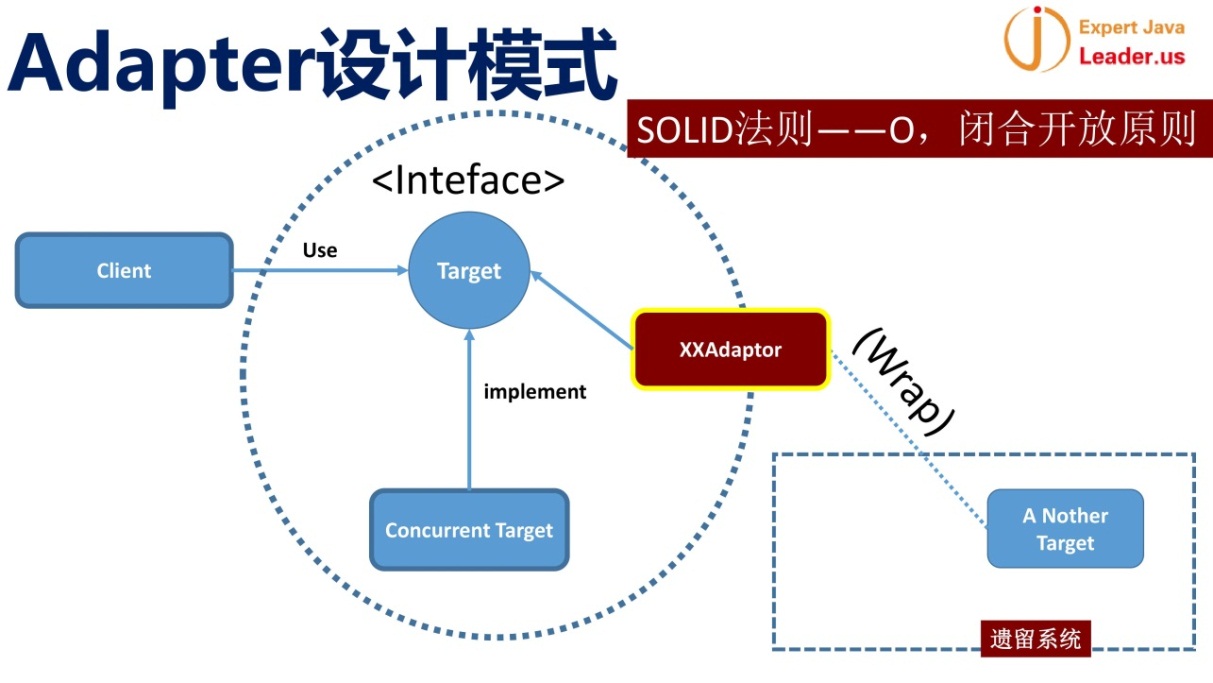




Adaptor模式 客户端用Target ---- 永远看到不变的接口 但是 具体用哪一个实现 客户端是不知道的 ----- 这个是典型的面向接口编程 此时 还不是某一种模式

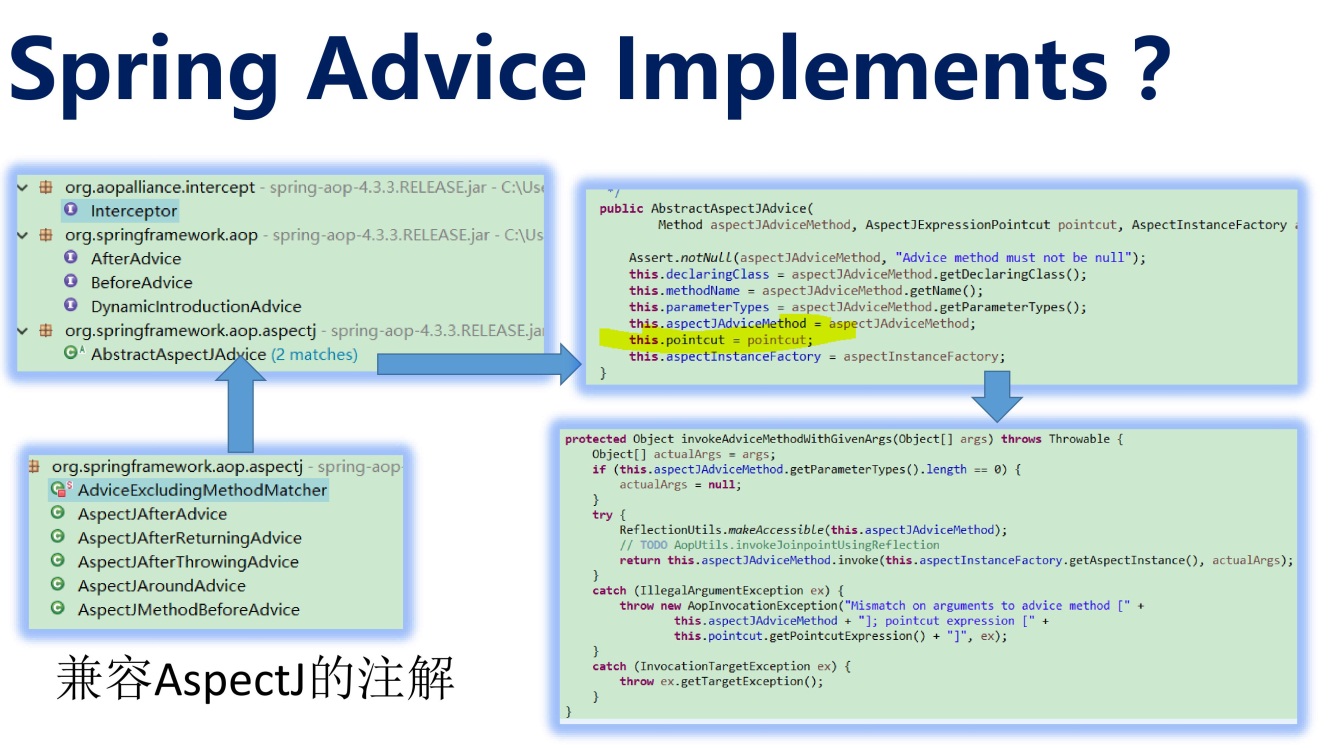
此时 有一个Target 并没有遵循左边的接口 那么 对应的客户端就无法真正的使用上这个Target 这样的话 需要这个不同类的Target进行改造才能被使用 所以 就有了适配器模式 ---- 把这个异类纳入进来 ---- 这种方式 希望不变代码 使用一个适配器 client --- 适配器实现这个目标接口 --- 这样这个Adaptor就会被client真正使用到 ----- 同时 这个Adaptor本身是没有内容的 **需要持有这个异类 + 把需要实现的接口通过异类的方法进行改造** -----

就是下页

 这样 Concrete Target【老师的PPT可能有错误 是Concrete Target 不是Concurrent Target --- concurrent是并发的意思】和XxxAdaptor都是相同的Target的实现类 ---- 如果使用了XxxAdaptor的话 这样 来自Client的请求 就都会通过XxAdaptor转发到这个Another Target里面

================ 这个Adaptor可以看成代理  **Anther Target就可以看成真实的目标类 ---- 适配器模式主要用来解决遗留问题**

Spring AOP 也是存在遗留问题的 跟AOP规范不兼容【因为说了 从Spring2.0开始 才拥抱AOP规范的】 ------ 所以 这里面就是用了适配器模式进行适配 满足AOP规范



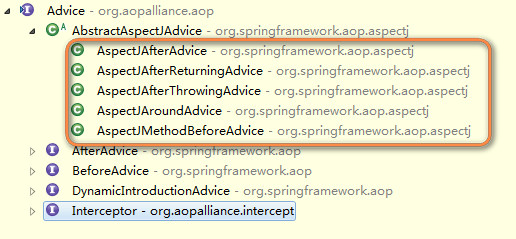
下面看一下这个类图

前面说过了 这个Advice是aop联盟的接口 是 标记接口 ------ 直接子接口是Interceptor

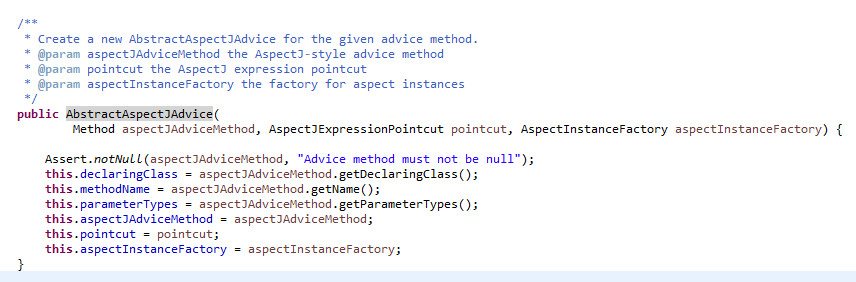
在Spring中 有**BeforeAdvice AfterAdvice和AbstractAspectJAdvice 和Interceptor**都是同一个级别的 只是 前面三个是Spring aop扩展出来的

这页PPT里面 是为了介绍Spring AOP 为了兼容业界标准AspectJ做的努力 ----- 左下角是所有AbstractAspectJAdvice的子类 --- 右上角是这个AbstractAspectJAdvice的构造方法 ------ 右下角是这个AbstractAspectJAdvice中的invokeAdviceWithGivenArgs 【实际上就是调用这个advice的方法的介绍】

我把这几个图进行了展开

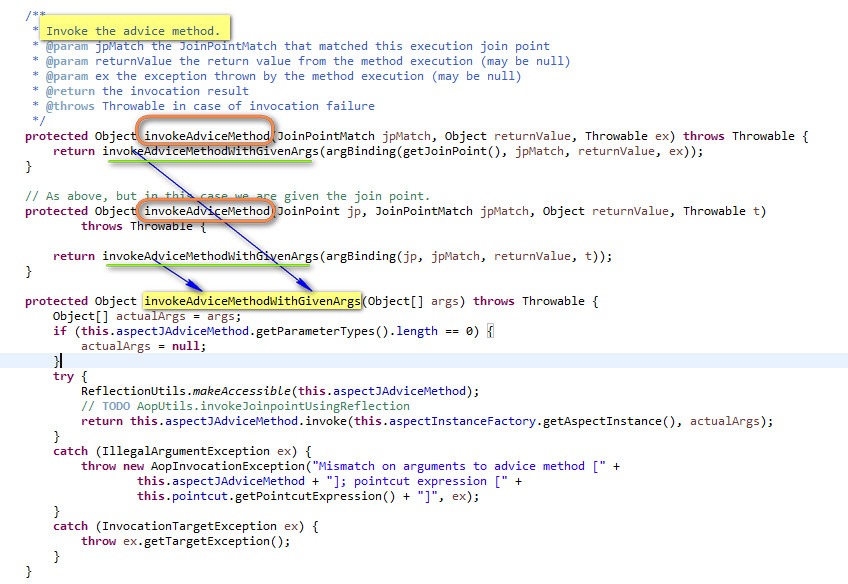


AbstractAspectJAdvice的子类扩展情况



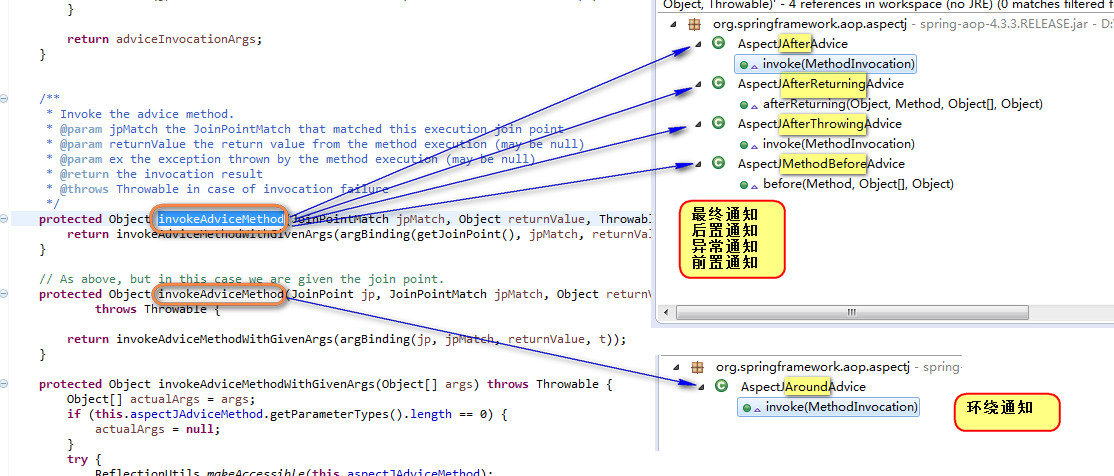
AbstractAspectJAdvice的构造方法

Spring往后 鼓励使用AspectJ的兼容模式 有pointcut aspect的方法

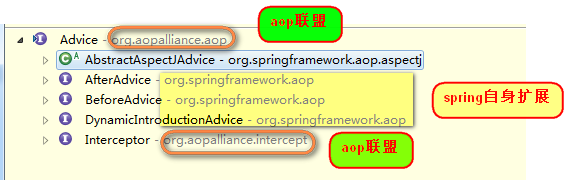


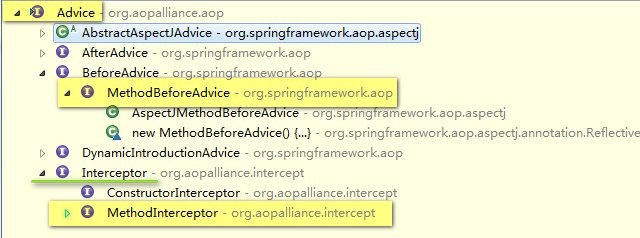
invokeMethod方法的时候 --------- 这里面最终都是调用反射方式 **进行调用最终还是回到了beforeAdvice和AfterAdvice** 这两种Spring里面的内容【真的是是这样么？？】【注意 持有Factory 就是获取instance 否则 就需要newInstance了 这样 就可以通过Factory获取instace了】

AbstractAspectJAdvice的invokeAdviceMethod的详细调用情况



这两个invokeAdviceMethod的最终调用分布情况



 再扩展一下 这个图里面就是之前学到的几个类的继承关系

DynamicIntroductionAdvice这个很少用 ----- 因为最后又要和AspectJ进行兼容 所以 有出现了一个AbstractAspectJAdvice 把AspectJ的注解全部兼容进来

**这块没有用到Aspect J的编程功能 而是引入注解 ---- 所以 Spring往后 是鼓励使用AspectJ的注解的兼容模式**

【前面是讲 因为历史遗留问题 还是 用户体验问题 向AOP联盟的规范靠拢 现在是讲 如何和业界的AOP规范的AspectJ进行兼容】

**【注意 要兼容 就很有可能使用到适配器模式 ---- 非常经典的话 ---- java中是对象适配器 + 变形的类适配器 这里面是就是变形的类适配器（类-接口适配器）】**

**【最后 整个Aop联盟的Advice在Spring aop做了兼容之后 分成了几个派别 一个是本身的Interceptor 另一个spring自身的BeforeAdvice AfterAdvice 和 为了兼容AspectJ的AbstractAspectJAdvice 这样三个大的派别 ----- AspectJ最后又挂到了Spring aop的接口上 ==== 再次印证了Spring 只是兼容AspectJ的注解 并没有真正把AspectJ注解的实现拿过来直接用】 -----经典**



AOP本身是没有Advisor这个概念的 –spring aop才有 无论使用哪种 都是左上图的做法 注解里面的切入点表达式就是pointcut ---- 很多时候pointcut和advice 确定where & when -----经常混合到一起 就是Advisor ---- **spring一开始就是Advisor** **【所以 开始为了兼容 出现了后面的事情 ---- 刚说了 有兼容 就有适配器模式】**



做上面图是Advisor这个接口的定义 --- 可以获取Advice 但是 下面那个方法没有用到 ---- **Advisor这个接口就是获取到Advice**

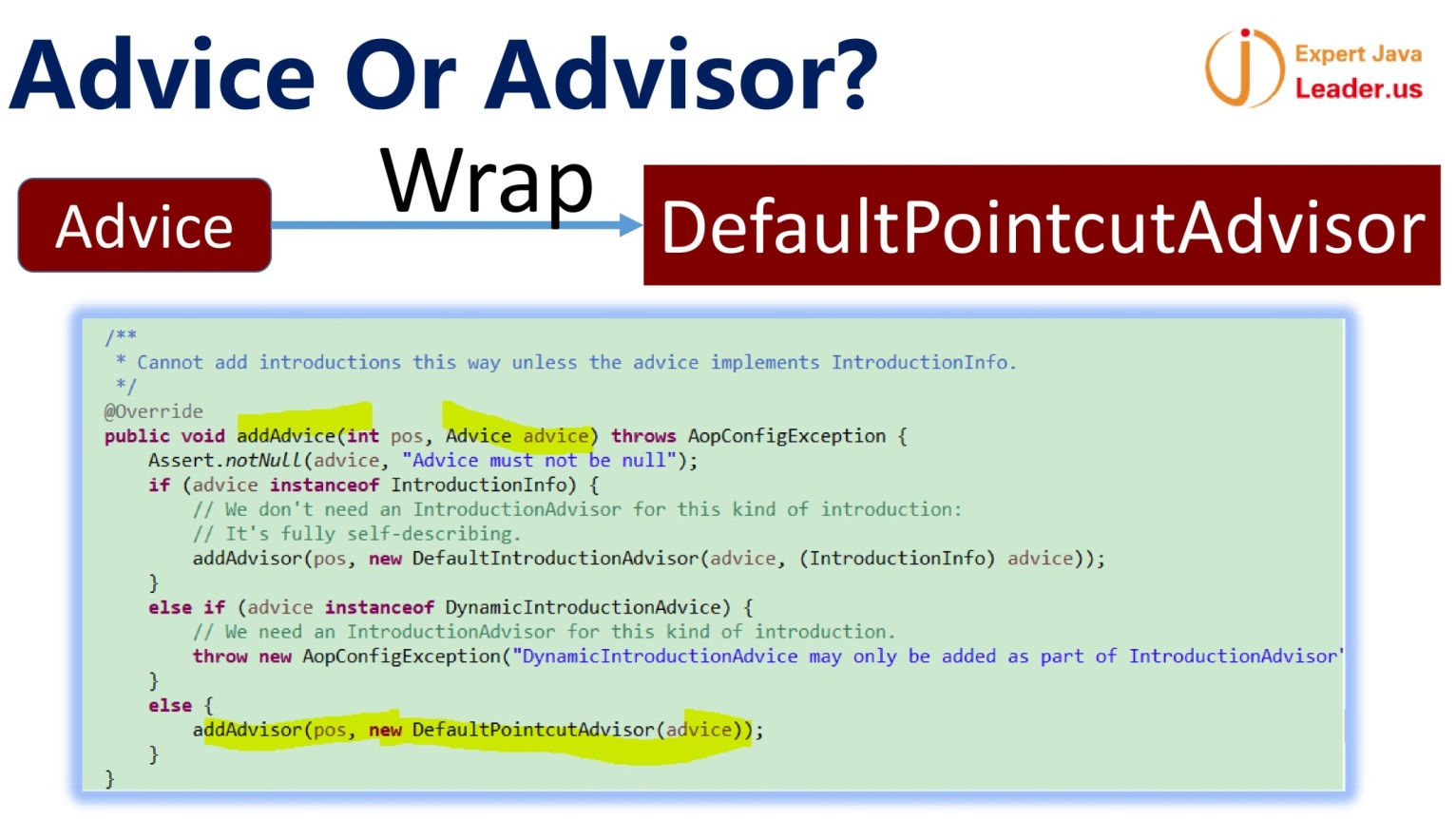
*IntroductionAdvisor 就是目标类级别拦截 ---- 就是 不改变目标类和属性的情况 加一些新的方法和属性 ----特殊情况用 不是对某一个方法的增强*

**最常用的就是PointCutAdvisor ----- 扩展了Advisor本身 增加了PointCut ===🡺这个可以获取到PointCut【Advisor可以获取到Advice 这样 PointcutAdvisor就可以获取到Advice+Pointcut ---- 就是Advisor的定义】**

**大量用的就是pointCutAdvisor就是AbstractPointcutAdvisor**

比较重要的是 DefaultPointAdvisor --- 包含了PointCut （where） ---- DefaultPointcutAdvisor可以进行where + when

Advisor的层级结构**(总结的纸上)**

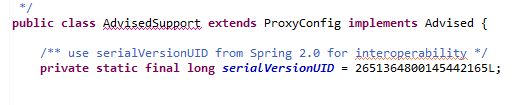


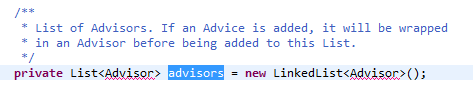
到底用Advice还是Advisor呢？ **Spring中 定义的Advice最终都会被包装成Advisor 【结论 非常重要】**

上面这段代码  **给某个目标Advice 加入到Advisor列表中**

【====== **这段代码来自 AdvicedSupport 【Adviced是构建责任链的核心】**

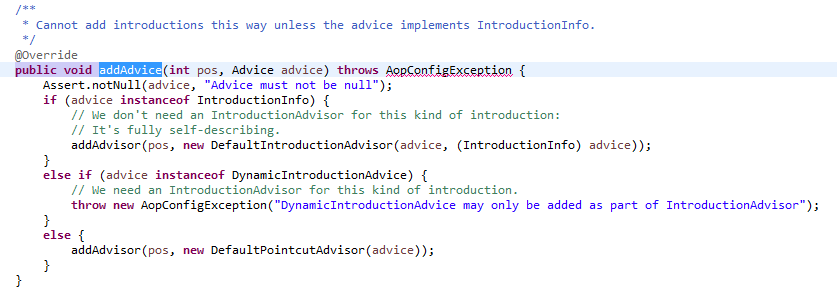
**看一下AdvisedSupport的定义**



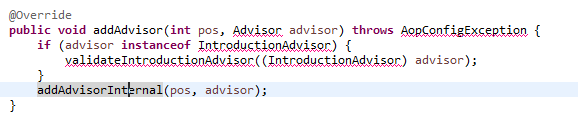


**也就是说AdvisedSupport是实现了Advised这个接口 --------**

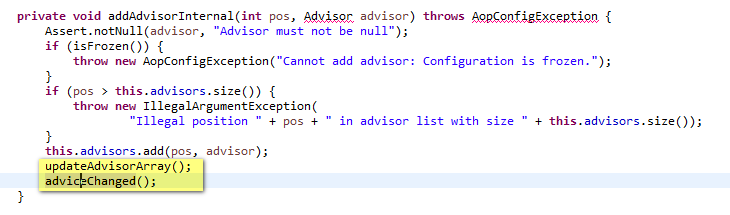
**====== 就是 Advice通过Adviced的实现类转换成了Advisor**



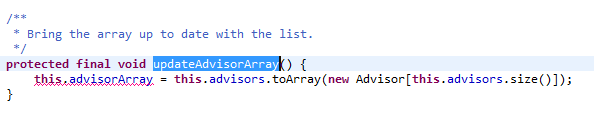
这个addAdvisor的片段



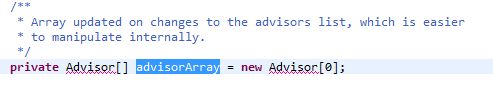
下面是addAdvisorInternal



\*\*\*\*\*\*\*\*看一下updateAdvisorArray();

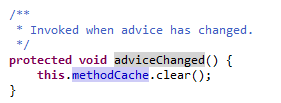


说明AdvicedSupport不仅有一个List<Advisor> 还有一个Advisor[] Advisor数组 **----真正操作的时候 需要转换成数组**

 开始数组容量为0 数组不可以扩容

所以 每次都是List#toArray 生成一个新的数组 ---- 注释说数组内部操作更方便【？？？？什么原因】

\*\*\*\*\*\*\*\*看一下advicedChanged();

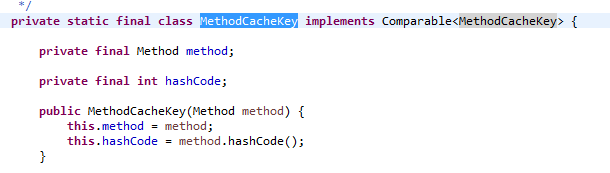
【因为拦截策略新加入了 所以 clear 给出最新的结果】

\*\*\*\*\*\*\*\*看一下mechodCache是什么

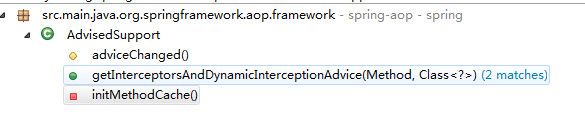


这个Map<MethodCachekey, List<Object>> methodCache 就是Method本身是key ---- advisor链条是value ===== **本质出来了 我们配置的Advisor是如何解析成最终的目标方法 和 拦截链的映射关系 ----- 就是依靠AdvicedSupport这个类**

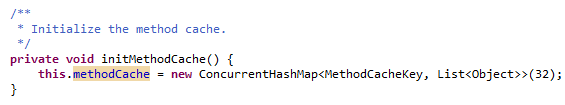
\*\*\*\*\*\*\*\*看一下MethodCachekey 是AdvicedSupport的今天内部类

 是方法的hashCode和Method本身构成的

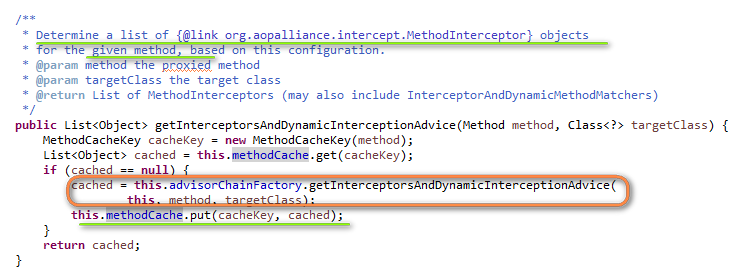
======== 都什么地方用到了这个methodCache呢？ 【**MethodCache引发出来 研究一下午的血案**】



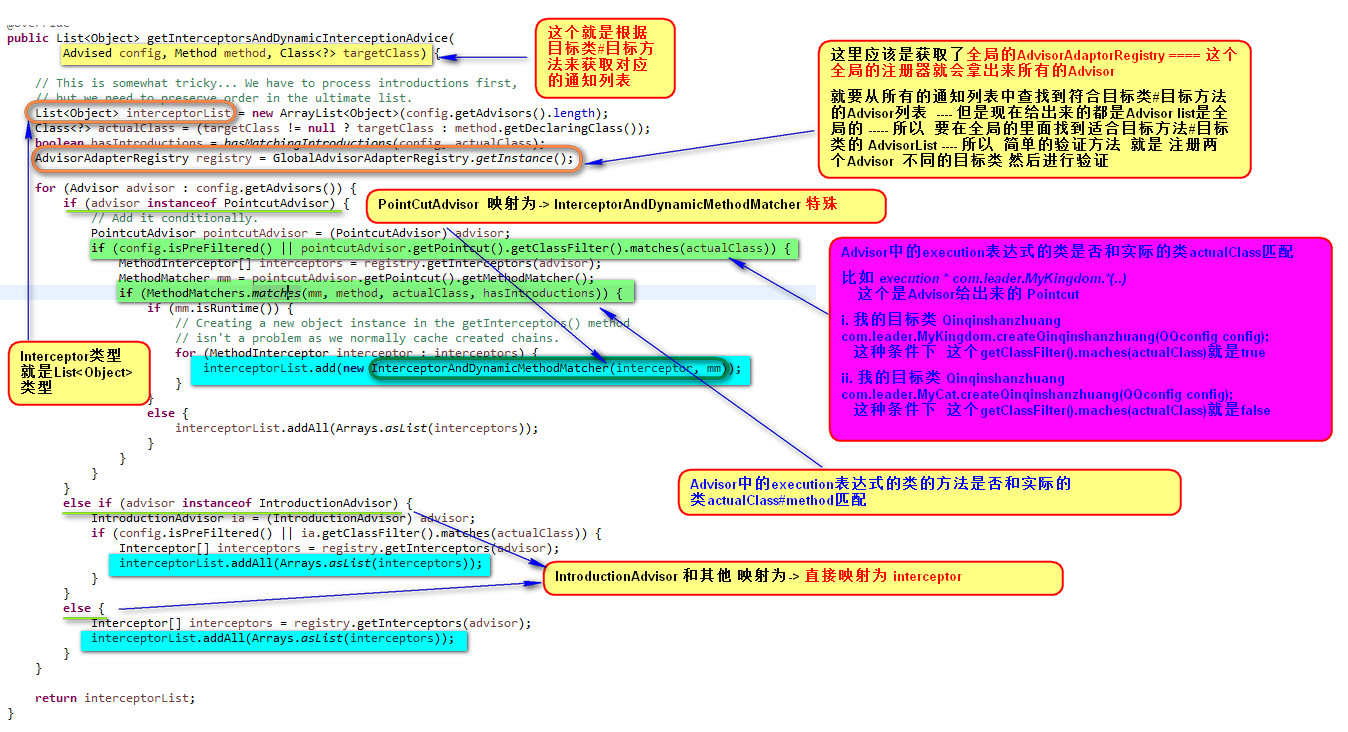
看initMethodCache()

 用的是ConcurrentHashMap

**getInterceptorsAndDynamicInterceptionAdvice(Method, Class<?>) 【注意这个命名 Interceptor & DynamicInterceptionAdvice 所以 下面的方法返回值类型就是 List<Object>】**

【基本很少见到new 全都是工厂模式 这个工厂直接在成员变量定义的时候 就赋值了  **----- 有时候直接传递工厂是非常可以理解的**】

这个就是查询一个方法对应的List<Object> 实际上就是List<Advisor>!!!!! ====🡺 找不到的时候 中间通过AdvisorChainFactory来创建这个List<Advisor>！！！！！====== **也就是 一个目标方法target method 会有哪些Advisor过来进行增强？** ---- method -> List<Advisor>



代码的重心思想就是：我现在有registry (估计是 advisor->interceptors的Map) **【需要弄清楚一个问题 就是interceptor 和 这个Advisor的关系？？】**

还有advisor (advice + pointcut =====> pointcut就是一个切入点表达式String)

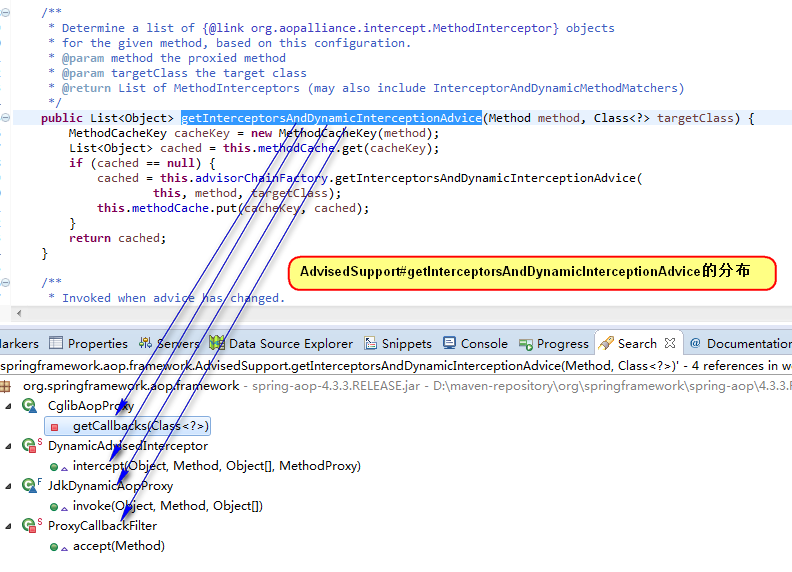
====> 依据Advised config 为目标类targetClass#的目标方法method 找到拦截器链 List<Object>

**注意 为什么叫InterceptorAndDynamicMethodMatcher呢？**

------- 中心词还是落在了Interceptor & methodMatcher

也就是 pointcutAdvisor ----- 实际上 是解析切入点表达式 --🡪 **结果就是 找到匹配的类 + 匹配的方法（这个就是最终目标） ---- 这时候 因为MethodMatcher#matches(Class, method) -> 保留这个mm就可以日后作为判断Advice是否是这个依据 ---- mm包含了method和Class的信息**

* Advice部分也要进行对应 =>通过Registry兑换成interceptors
* 这样 就把interceptors和MethodMatcher组合 日后使用 ---- 否则 都不知道的interceptor是谁的！ 所以 体现出了methodMatcher的重要性！ 而绝对不是可有可无 ==🡺 所以 如果我们用AspectJ来进行编程 那么 基本都是PointcutAdvisor ==== 结果就是 我们的List里面都是InterceptorAndDynamicMethodMatcher ---- Dynamic是动态的 就是 我在遍历这个动态过程中 找到的methodMatcher
* 这里面 Spring的作者 又一次使用了极致编程 就是 直接传递进来targetClass 和 method ---- 判断class的时候 直接用 没有使用method.getDeclaredClass 这样 反射可能会降低性能
* 至于后面的IntroductionAdvisor和其他 估计是普遍都适配 如果有 就都找出来 所以 每一个targetClass#method
* 这就看出来 InterceptorAndDynamicMethodMatcher是最重要的！！！！

 可以看到 这个获取链条（同时也是生成链条 因为没获取到就Create）的方法 用到了AopProxy中 ---- **典型的就是CglibAopProxy 和JdkDynamicAopProxy [进一步体现了struts2的ActionProxy含有ActionInvocation！！！！的思想]**

】

如果是IntroductionInfo类型 就把这个Advice包装成 DefaultIntroductionAdvice 然后addAdvisor

如果是DynamicIntroductinAdvice类型 不支持

剩下的都包装成DefaultPointcutAdvisor类型 **【刚才说了 Advice有三大派别 一个是spring自身的BeforeAdvice AfterAdvice 还有就是 Advice的嫡系 Interceptor 还有就是和AspectJ适配的AbstractAspectJAdvice 还有一个就是不常用的DynamicIntroductionAdvice 那么 这里面可以看到 除了IntroductionInfo 和 DynamicIntroductionAdvice 这两种 主要的三大派别的Advice全部都被包装成DefaultPointcutAdvisor】**

也就是大部分advice编程DefaultpointcutAdivsor加入进来

**------ Advisor如何适配到Aop联盟的规范呢？**

【所以 要明白 就是 AdvisorAdaptor的作用 一个是判定是否是对应的Advice 另一种 就是 通过这个Advisor就可以获取到对应的Interceptor 完成二级适配】

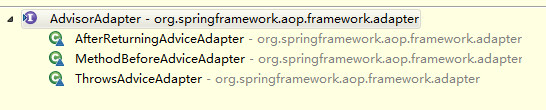
因为Advisor是兼容到AOP联盟 所以 这个时候 必须进行适配

做了AdvisorAdaptor这样一个接口 过来一个Advice 变成Advisor ----- Advisor包含了所有的信息

这个接口有一个boolean supportsAdvice这个方法 如果支持advice 就进行转换 ------ 另外 Advisor上面就可以得到Advice ---- 有了Advice就可以得到methodInterceptor ---- 这样 就是通过AdvisorAdaptor中 如果supportsAdvice成立 那么 就可以得到Advice 那么 就得到了Interceptor【spring aop的Advice是来自于BeforeMethodAdviceInterceptor 等等】 有了Interceptor 就可以回归到AOP规范里面

**【所以 无论是Spring自己的Advice 还是 自身的新概念Advisor 都是适配到Interceptor ------ 这样 Advisor通过AdvisorAdaptor适配到Advice ---- 如果这个Advice是Spring aop自己不符合规范的Advice 还要借助BeforeMethodAdviceInterceptor或者AfterMethodAdviceInterceptor回归到AOP联盟的规范----- 下面就举一个具体的例子 来表明这一点 MethodbeforeAdviceAdaptor ------ 】**

【继承关系



有三个AdvisorAdaptor的实现子类 ----- 分别是 AfterReturningAdviceAdaptor MethodBeforeAdviceAdaptor和ThrowsAdviceAdaptor】

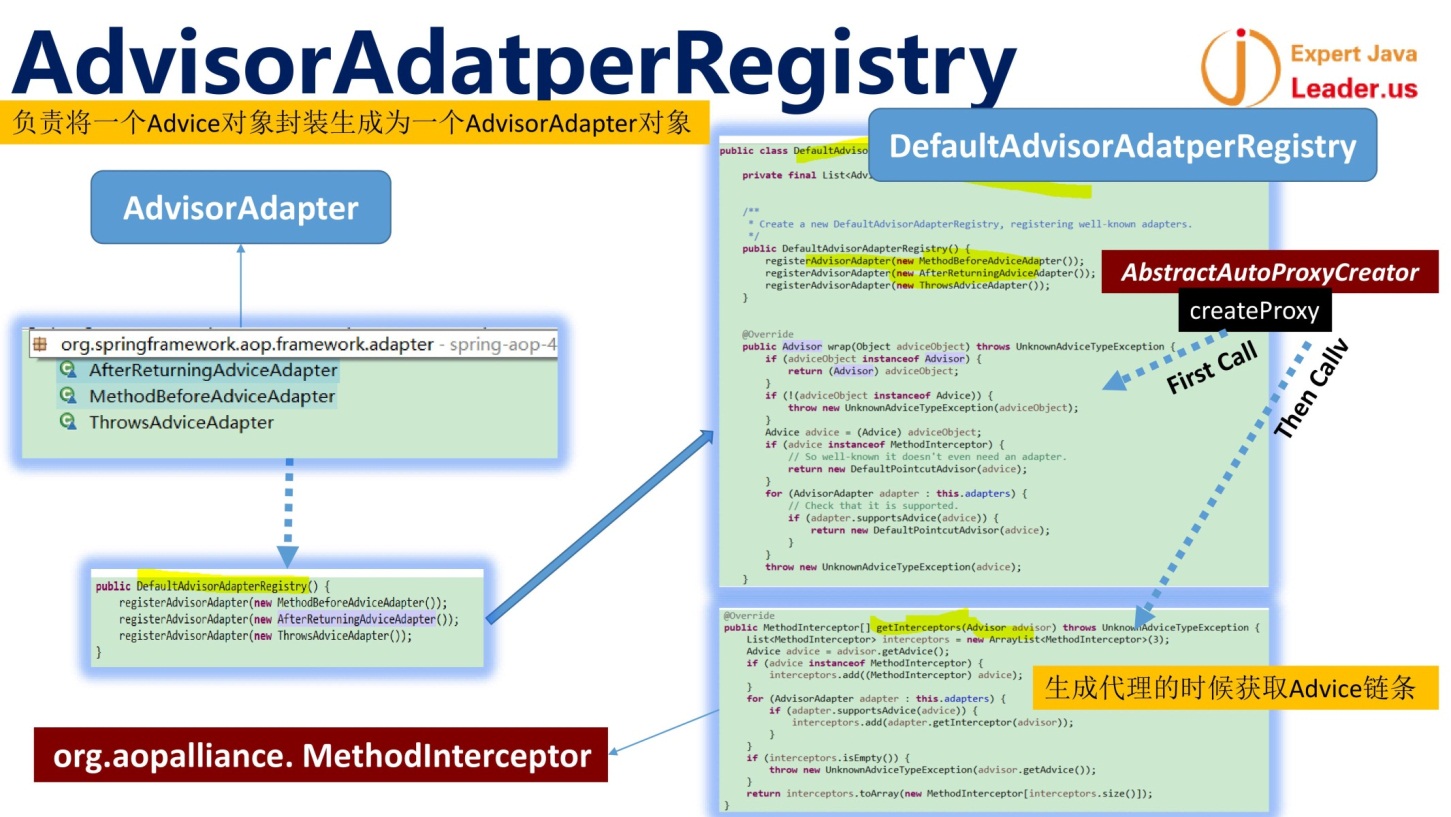
来看一下 MethodBeforeAdviceAdaptor 实现了 AdvisorAdaptor



---- 这个命名很纠结的 命名这个地方是AdvisorAdaptor 但是 真的实现类确是XxxAdviceAdaptor

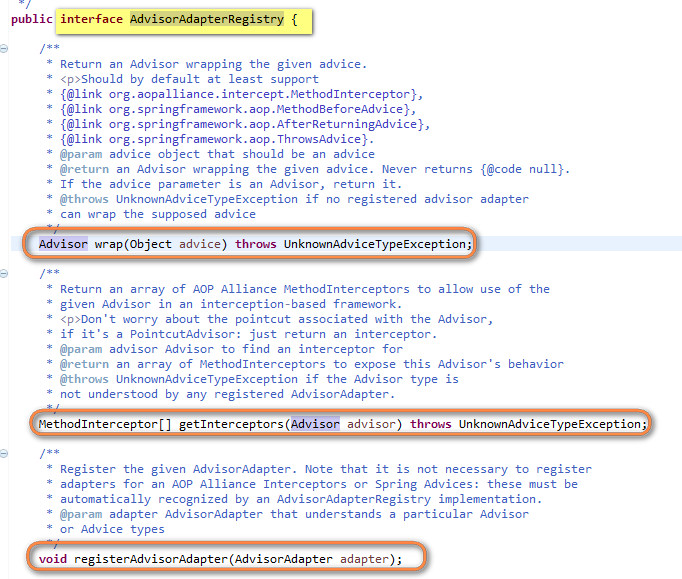
这个里面 如果支持Advice 那么 就可以new获取到这个Advice的Interceptor -----【这**样Advisor通过二级适配** 兼容到了AOP联盟的规范 ---- 可以换画一个图 advice 和 Advisor适配的过程】

做的就是 把 MethodBeforeAdvice变成一个MethodBeforeAdviceInterceptor 这样一来 如果是methodBeforeAdvisor 就会产生一个MethodBeforeInterceptor 这个就是AOP标准的

 AdvisorAdaptorRegistry

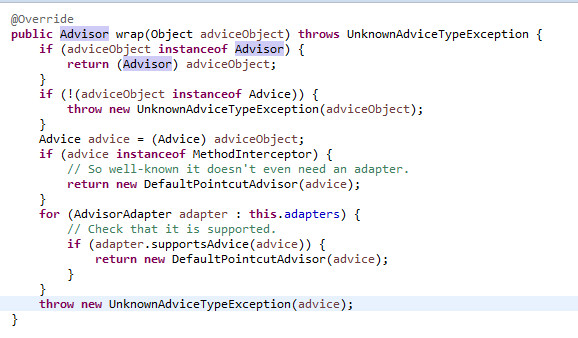
AdvisorAdaptor并不多 ---- 实现有这三种 然后 放在Registry中了 -------

构造方法中 就把这三种AdvisorAdaptor进行了注册 --- -registerAdvisorAdaptor

 【这个AdvisorAdaptorRegistry 有三个方法 可以翻译一下】

构造的时候 register 每一种Advisor负责每一种Advice处理

有了这个AdvisorAdaptorRegistry之后 就可以把一个对象包装成Advisor 并且把这个Advisor上面的Interceptors都获取到

如果仅仅是Advisor本身 就返回

如果是Advice 那么 看这个Advice是不是MethodInterceptor ----- 如果是 那么就转换成DefaultPointCutAdvisor 【实现了Advice到Advisor的映射】

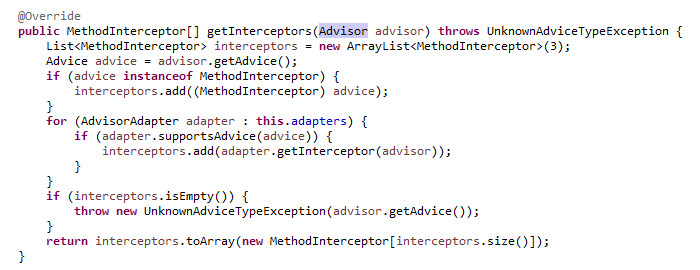
如果是Advice 但是 不是MethodInteceptor这种 那么就从前面注册好的adaptor里面 找 看看是哪一种Advice ------- 如果是对应的Advice 就转换成DefaultPointcutAdvisor【还是Advice到Advisor的映射 只是这块的映射 通过AdvisorAdaptor的子类来辅助完成的】

【这个里面 三个Adaptor的supports方法 分别对应了BeforeMethodAdvice AfterReturningAdvice和xxxx 自己看一下去 看看MethodInterceptor 和 其他的Advice的继承关系 】

【这里面用的Adaptor得作用是Adaptor里面有一个supports的方法 判定是不是这个Advice】

Wrap就在后面产生AopProxy的时候 首先根据这个Advice产生这个Advisor --- 调用的就是这个wrap的方法 得到这个上面的Advisor对象

---- 得到Advisor对象之后 就获取这个Advisor对象上面所有的拦截器 MethodInterceptor[]

【这个方法 用了AdvisorAdaptor的getInterceptor这个方法 获取了一个Advisor对应的所有inteceptor】

这样就返回了一个链条 ---- 也可以能是一个

通过这里面的方法 可以把一个对象上面Advice变成Advisor 然后从Advisor里面得到所有的MethodInterceptor[] 链条

【注意图中的First Call SecondCall】



正常创建一个Target Object 对应一个Proxy ---- 现在希望 每个线程产生一个独立的TargetObject 和 对应的线程独立的Proxy ---- 就是每一个线程都有自己的target 和代理

或者 生成一个资源池 **类似于数据库中的连接池 ---- 这个里面 target比较重**  ----- 复用思想

其实 默认情况 Spring中 我们的bean默认是单例的 --- 所以 只有一个Target 和Proxy

所以 目标是

现在我的图中 有一个XxxxClass这个类

现在想产生***类似于Prototype这种情况的多实例*** ---- 产生多个target Object -----做法一样 因为AOP的原因 增加了复杂度

【就是 要是prototype 可以生成多个target 但是 如果要是让proxy也具有prototype这样的性质 该怎么办？】

如何实现非单实例的bean 每次调用都产生一个新的呢？ 做了下面的设计 ----



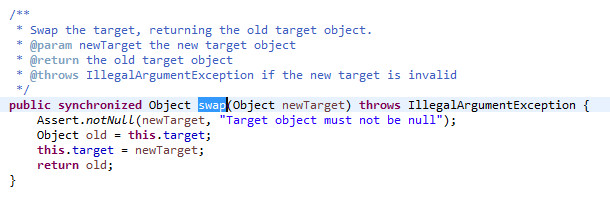
定义了一个TargetSource这个类 产生TargetObject ----- 根据class产生TargetObject ----- TargetSourceCreator产生一个TargetSource

【二级工厂 最终产品是TargetSource】

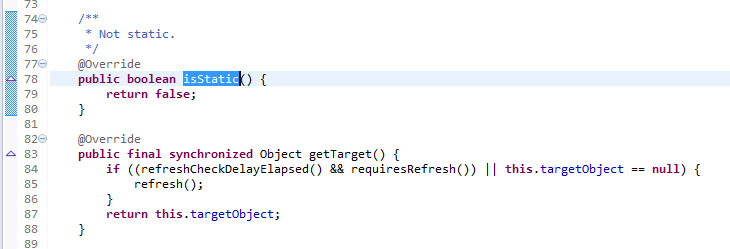
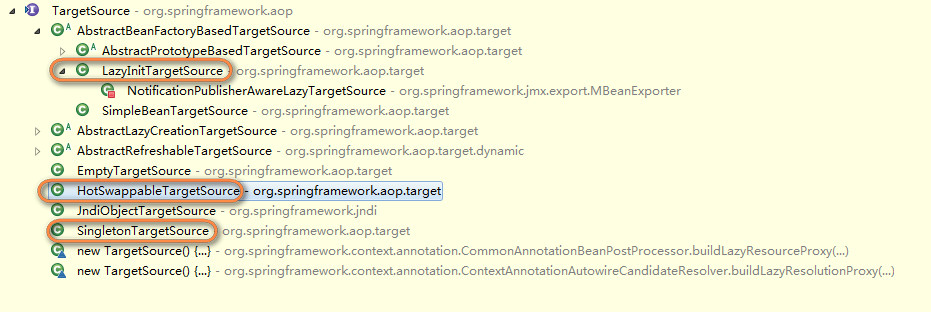
这样就做了一个非常复杂的抽象过程 就是右下角的类图 ---- 有LazyInitTargetSourcCreator【模仿Spring中原有的延迟加载的属性 但是 对的是proxy】 ----- 还有一个QuickTargetSourceCreator 这个有下面的ThreadLocalTargetSource ComonsPool2TargetSource和PrototypeTargetSource



这里面TargetSource的实现很多 有一个SingonTargetSource 还有一个HotSwappableTargetSource ---- 这个可热交换的TargetSource非常有趣



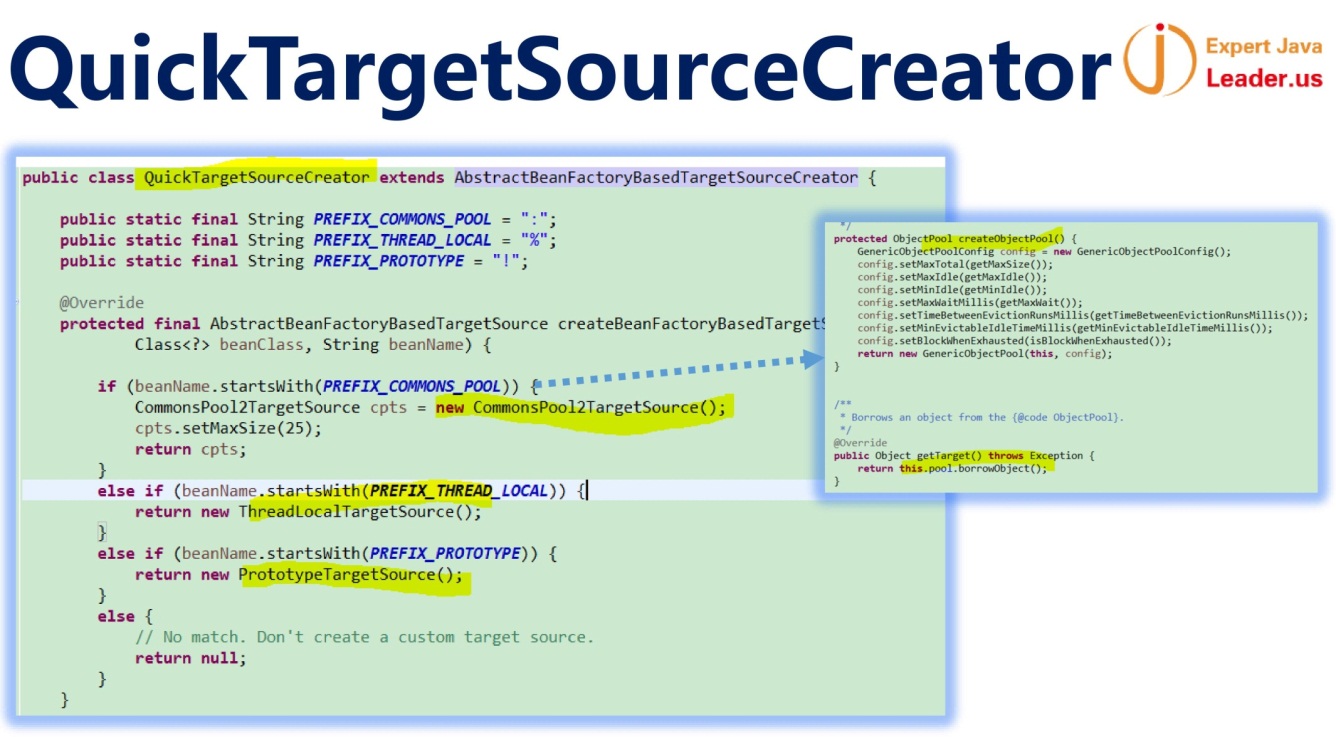
支持热替换



Refreshable可以刷新 产生一个新的target Object

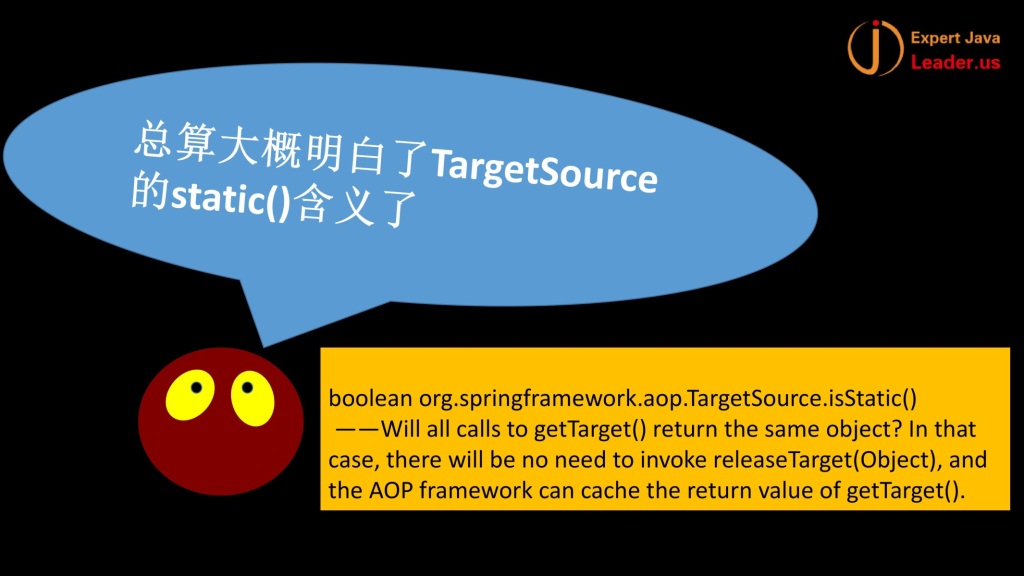
Spring默认用的是Singleton的TargetSource 这儿里面 有一个setTargetSource -----

所以 讲了半天的的spring aop中的target 实际上调用的是SingletonTargetSource ----- 这个每次调用都会产生一个新的



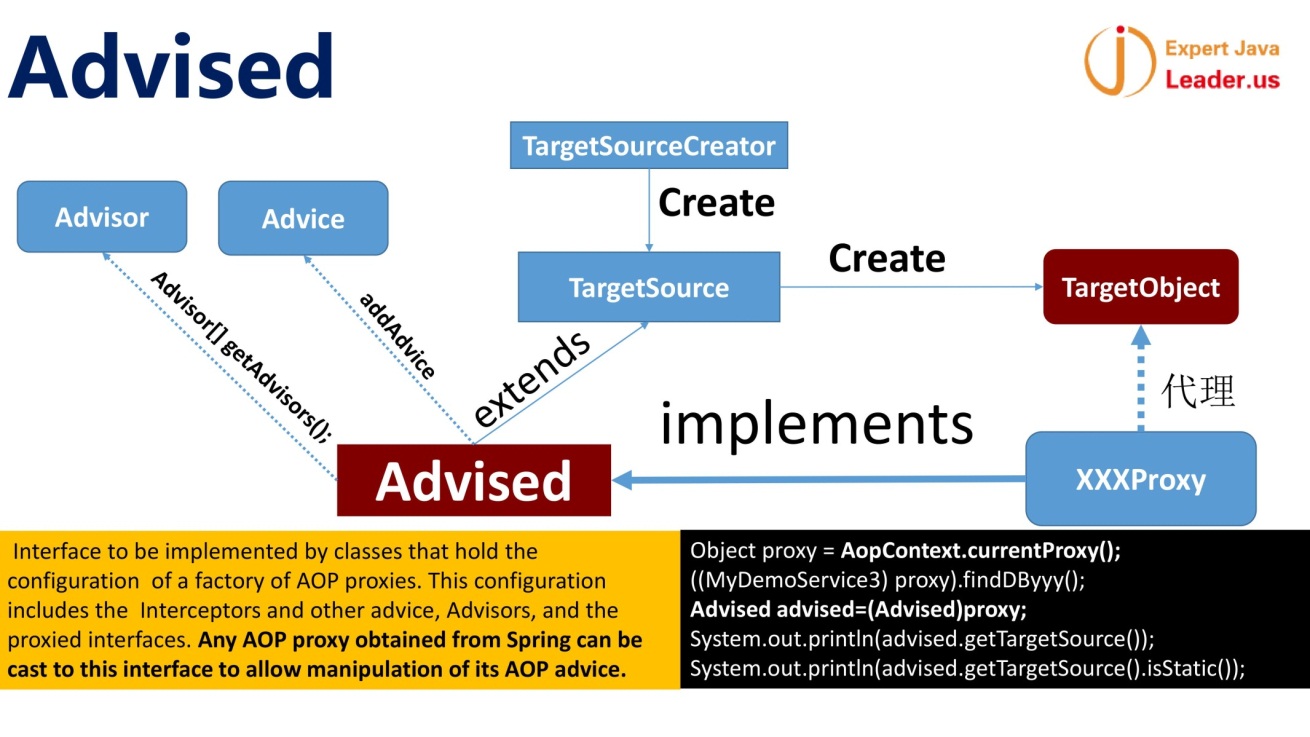
做法来看 根据beanName的前缀名称 比如PREFIX\_COMMONS\_POOL是:开头 这个开头 使用CommonsPool2TargetSource ---- 最右边的图 就是 createObjectPool() ---- 这个里面设定了config ---- 然后get的时候 --- 就是getTarget就是从pool里面获取一个 ----- 可能有得不到的问题

ThreadLocal模式 就是 放在每一个都放在ThreadLocal变量中



TargetSource有一个isStatic() ---- 就是看每次山城的是否是一样的 ---如果是一样的 就不用释放 否则就会告诉spring aop进行释放

【有时间要看下TargetSource的使用】



前面所有对象被继承到Advised里面 ----- Advice转换成Advisor ----- Advised上面的类图是 扩展了TargetSource接口

【Advised->TargetSource接口】这样 Advised就**可以创建Target对象本身** + *可以添加Advisor Advice* 得到所有的Advisor ---- **这样 Adviced接口具备了我们创建一个Proxy的基本条件**

----- 从Advisor还可以得到所有的拦截器 ===== =所以 最终的XxxxProxy对象 实际上是实现了Advised接口

【Advised是被动语态 --- 被增强的 ----- 是一个接口 】

代码可以这样测试：

Object proxy = AopContext.currentProxy();

((MyDemoService3)proxy).findDByyy();

Advised advised = (Advised)proxy;

Syso(advised.getTargetSource());

Syso(advised.getTargetSource().isStatic());

运行时候 得到相关的信息可以安全转换成Advised接口

*【TargetSource的原理还没有讲清楚！！！】*