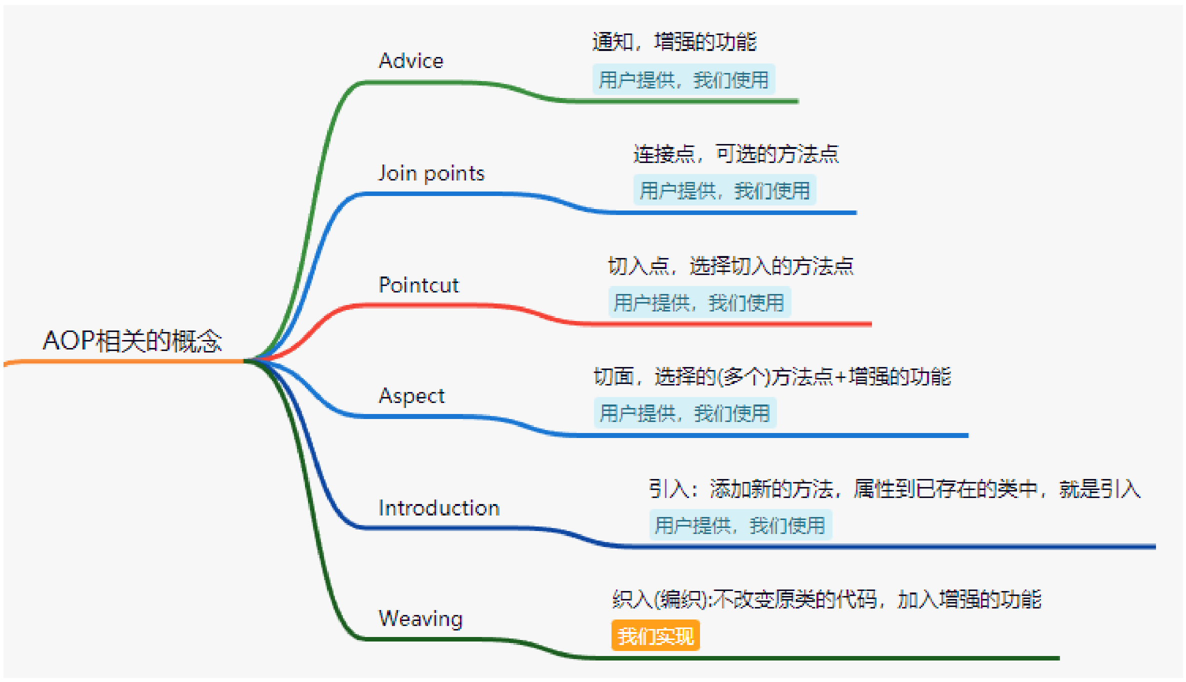
# Spring源码-AOP分析

# 一、手写AOP回顾

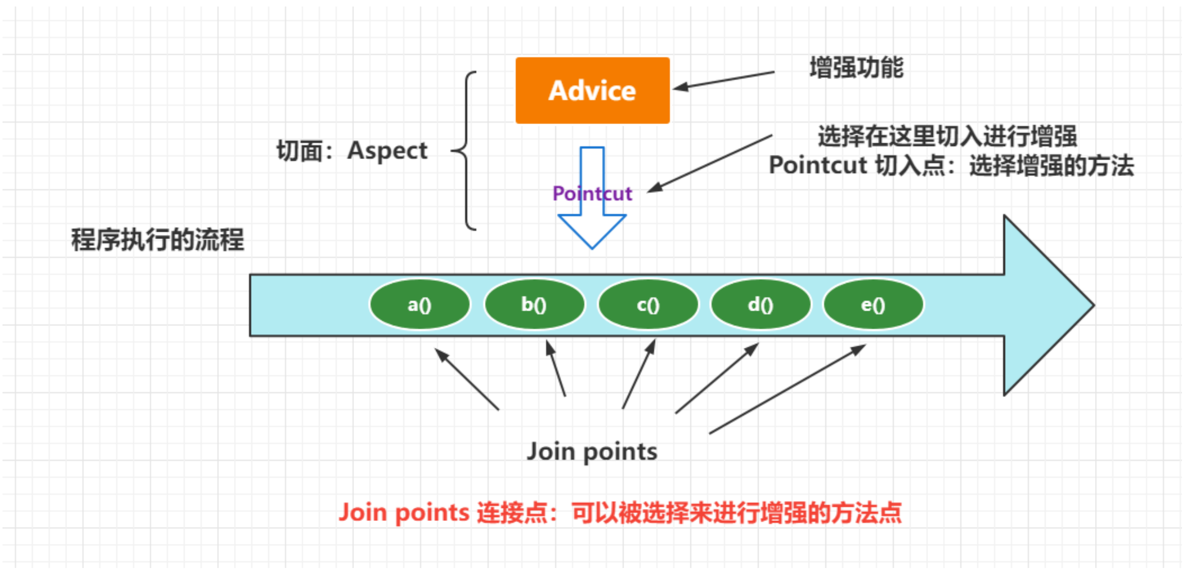
本文我们开始讲解Spring中的AOP原理和源码，我们前面手写了AOP的实现，了解和自己实现AOP应该要具备的内容，我们先回顾下，这对我们理解Spring的AOP是非常有帮助的。

## 1. 涉及的相关概念

先回顾下核心的概念，比如：Advice，Pointcut，Aspect等

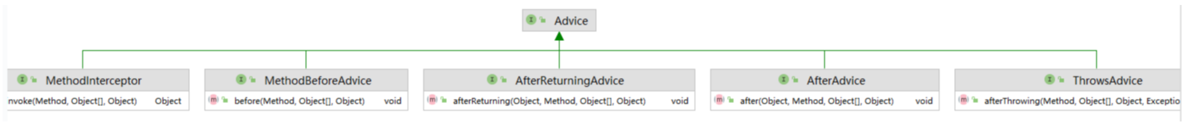


更加形象的描述：

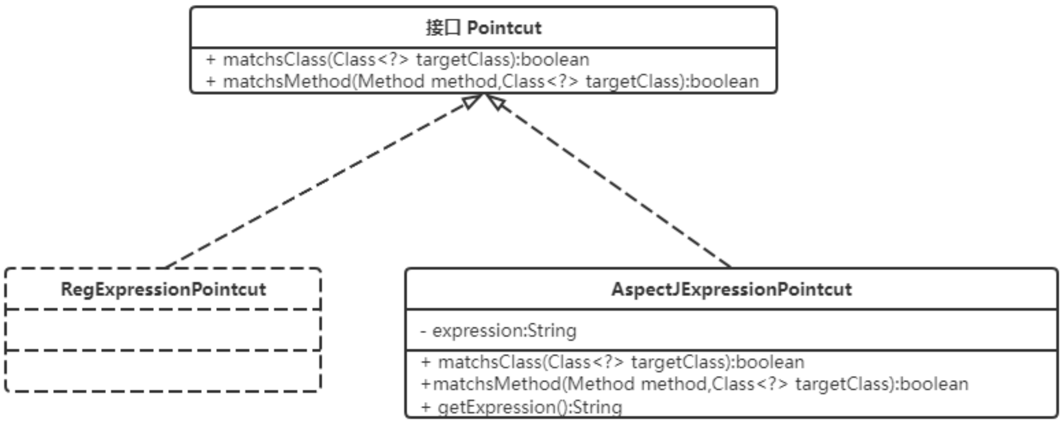


## 2. 相关核心的设计

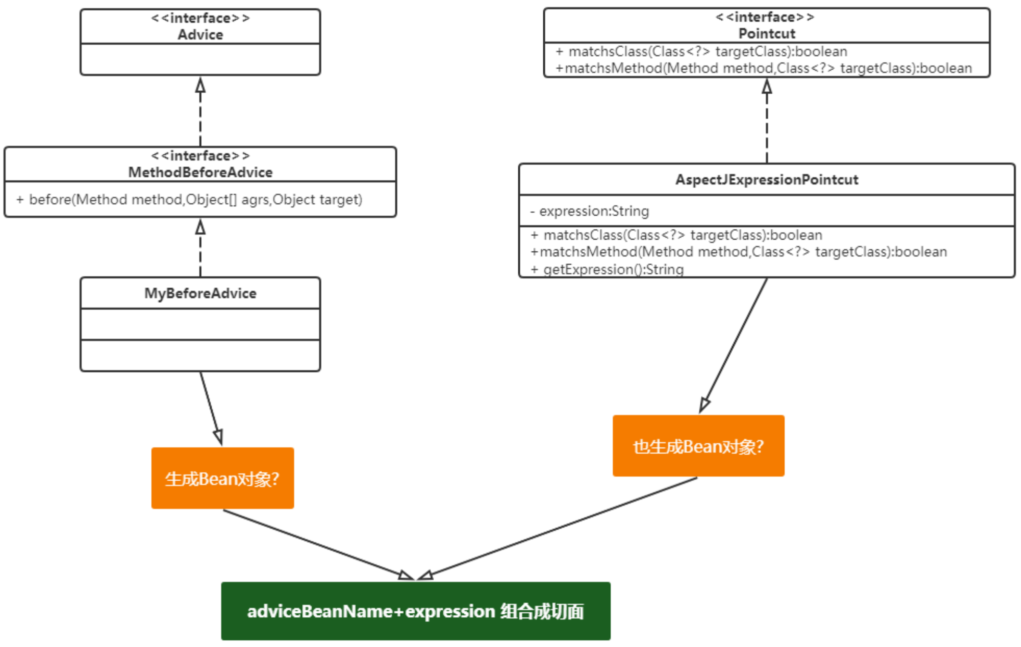
Advice:


Advice
MethodInterceptor
nvoke(Method, Object[], Object) Object
MethodBeforeAdvice
before(Method, Object[], Object) void
AfterReturningAdvice
afterReturning(Object, Method, Object[], Object) void
AfterAdvice
after(Object, Method, Object[], Object) void
ThrowsAdvice
afterThrowing(Method, Object[], Object, Exceptio

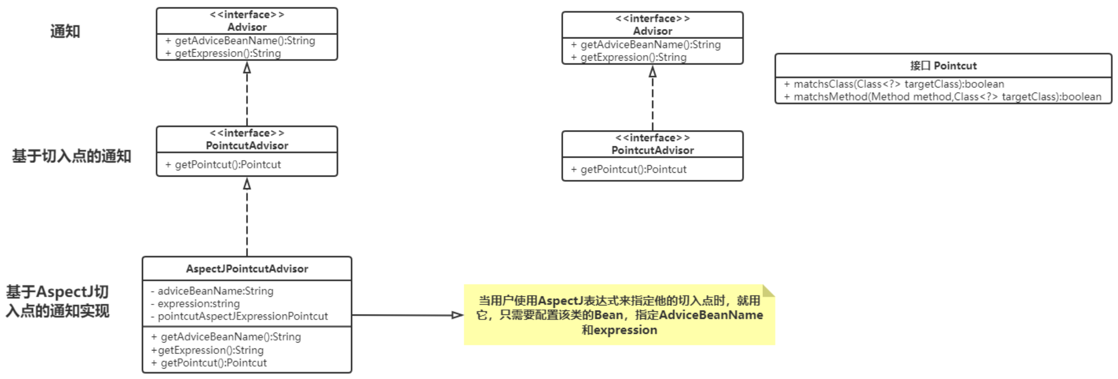
Pointcut:


接口 Pointcut
+ matchsClass(Class<?> targetClass):boolean
+ matchsMethod(Method method,Class<?> targetClass):boolean
RegExpressionPointcut
AspectJExpressionPointcut
- expression:String
+ matchsClass(Class<?> targetClass):boolean
+matchsMethod(Method method,Class<?> targetClass):boolean
+ getExpression():String

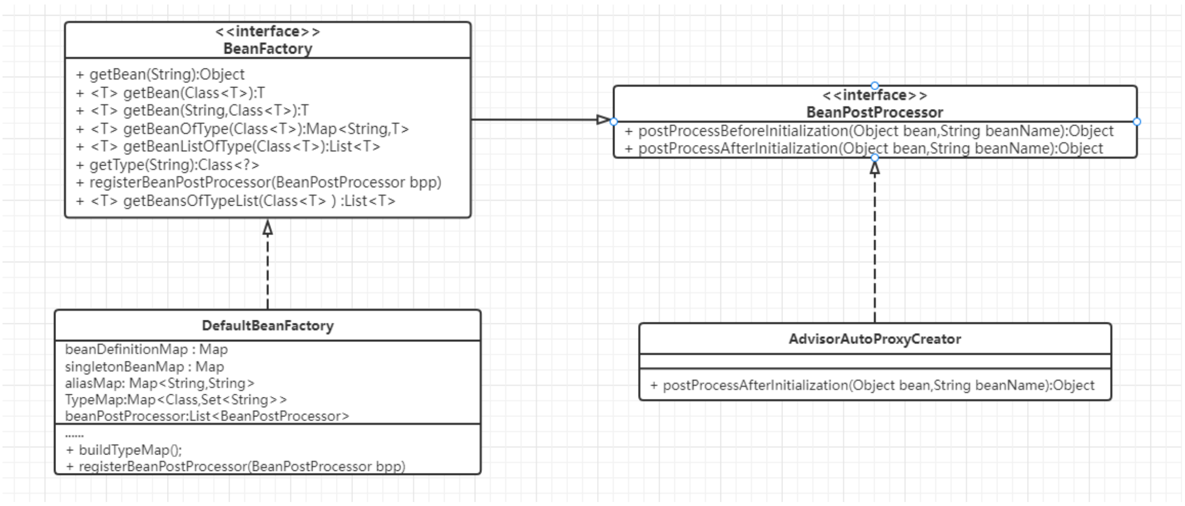
Aspect:


<<interface>>
Advice
<<interface>>
MethodBeforeAdvice
+ before(Method method,Object[] agrs,Object target)
MyBeforeAdvice
生成Bean对象?
<<interface>>
Pointcut
+ matchsClass(Class<?> targetClass):boolean
+matchsMethod(Method method,Class<?> targetClass):boolean
AspectJExpressionPointcut
- expression:String
+ matchsClass(Class<?> targetClass):boolean
+matchsMethod(Method method,Class<?> targetClass):boolean
+ getExpression():String
也生成Bean对象?
adviceBeanName+expression 组合成切面

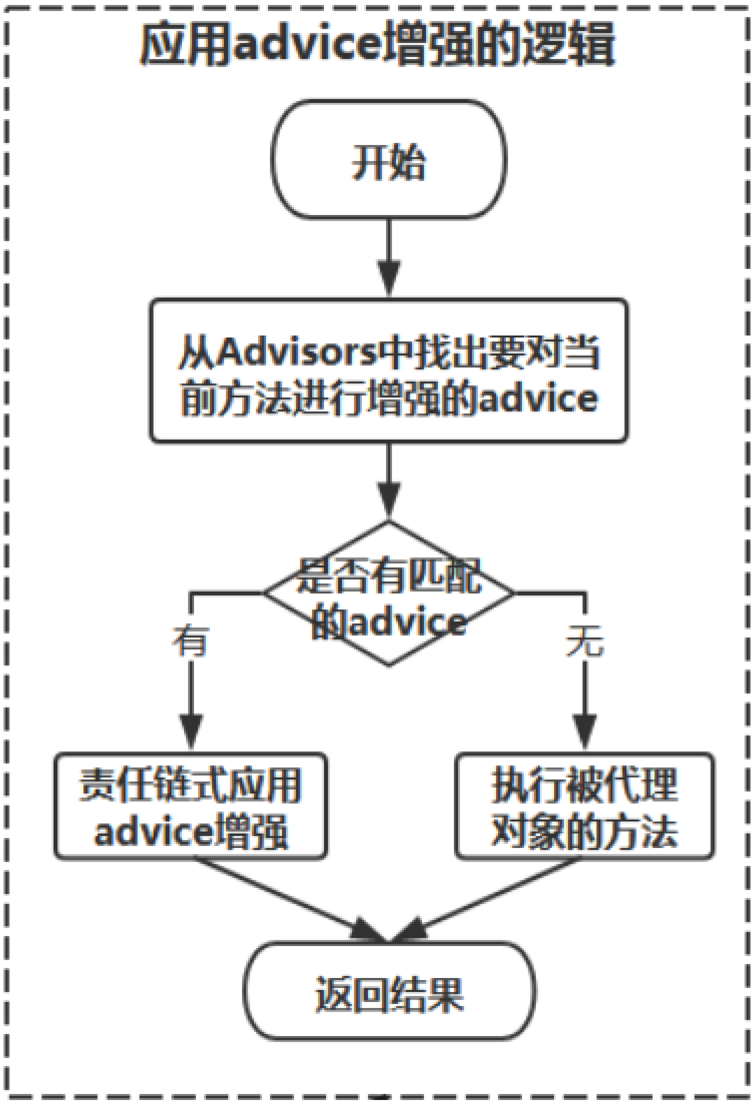
Advisor:


通知
<<interface>>
Advisor
+ getAdviceBeanName():String
+ getExpression():String
基于切入点的通知
<<interface>>
PointcutAdvisor
+ getPointcut():Pointcut
基于AspectJ切入点的通知实现
AspectJPointcutAdvisor
- adviceBeanName:String
- expression:string
- pointcutAspectJExpressionPointcut
+ getAdviceBeanName():String
+getExpression():String
+ getPointcut():Pointcut
当用户使用AspectJ表达式来指定他的切入点时，就用它，只需要配置该类的Bean，指定AdviceBeanName和expression
<<interface>>
Advisor
+ getAdviceBeanName():String
+ getExpression():String
<<interface>>
PointcutAdvisor
+ getPointcut():Pointcut
接口 Pointcut
+ matchsClass(Class<?> targetClass):boolean
+ matchsMethod(Method method,Class<?> targetClass):boolean

织入：



<<interface>>
BeanFactory
+ getBean(String):Object
+ <T> getBean(Class<T>):T
+ <T> getBean(String,Class<T>):T
+ <T> getBeanOfType(Class<T>):Map<String,T>
+ <T> getBeanListOfType(Class<T>):List<T>
+ getType(String):Class<?>
+ registerBeanPostProcessor(BeanPostProcessor bpp)
+ <T> getBeansOfTypeList(Class<T> ) :List<T>
<<interface>>
BeanPostProcessor
+ postProcessBeforeInitialization(Object bean,String beanName):Object
+ postProcessAfterInitialization(Object bean,String beanName):Object
DefaultBeanFactory
beanDefinitionMap : Map
singletonBeanMap : Map
aliasMap: Map<String,String>
TypeMap:Map<Class,Set<String>>
beanPostProcessor:List<BeanPostProcessor>
......
+ buildTypeMap();
+ registerBeanPostProcessor(BeanPostProcessor bpp)
AdvisorAutoProxyCreator
+ postProcessAfterInitialization(Object bean,String beanName):Object

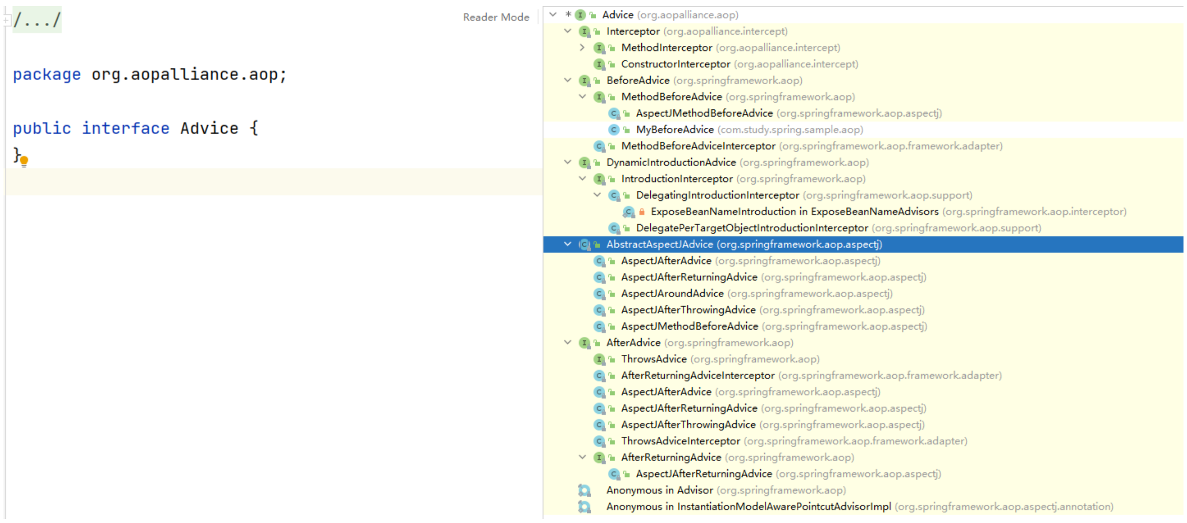


## 二、AOP相关概念的类结构

回顾了前面的内容，然后我们来看看Spring中AOP是如何来实现的了。

### 1. Advice类结构

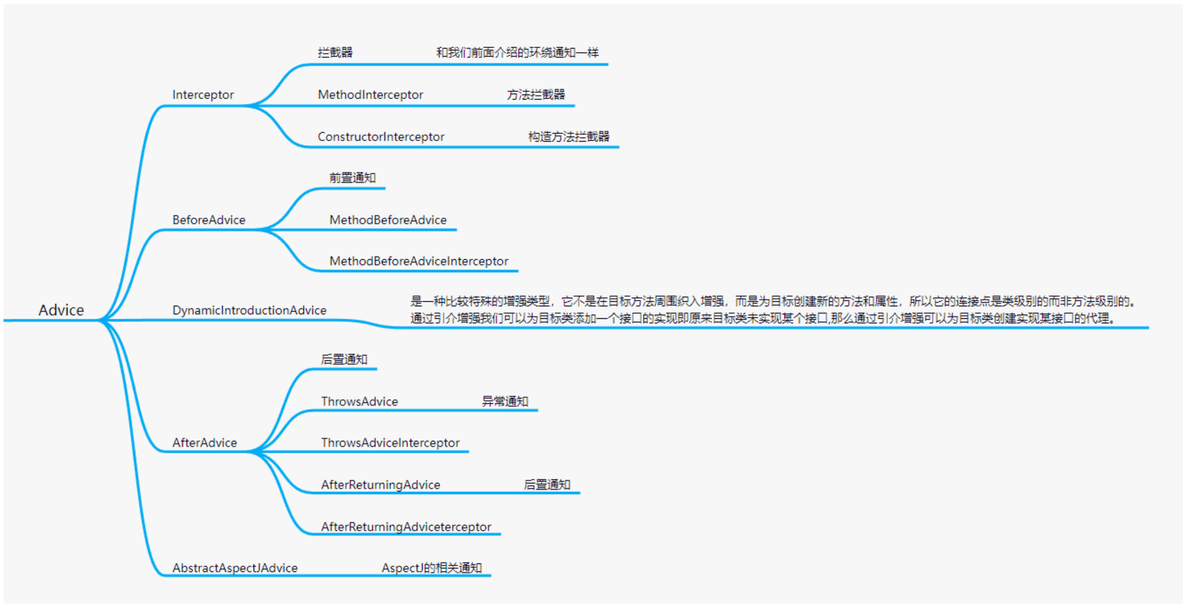
我们先来看看Advice的类结构，advice--》通知，需要增强的功能



```
/.../

package org.aopalliance.aop;

public interface Advice {
}
```

```
* Advice (org.aopalliance.aop)
  Interceptor (org.aopalliance.intercept)
    MethodInterceptor (org.aopalliance.intercept)
    ConstructorInterceptor (org.aopalliance.intercept)
  BeforeAdvice (org.springframework.aop)
    MethodBeforeAdvice (org.springframework.aop)
      AspectJMethodBeforeAdvice (org.springframework.aop.aspectj)
      MyBeforeAdvice (com.study.spring.sample.aop)
    MethodBeforeAdviceInterceptor (org.springframework.aop.framework.adapter)
  DynamicIntroductionAdvice (org.springframework.aop)
    IntroductionInterceptor (org.springframework.aop)
      DelegatingIntroductionInterceptor (org.springframework.aop.support)
        ExposeBeanNameIntroduction in ExposeBeanNameAdvisors (org.springframework.aop.interceptor)
      DelegatePerTargetObjectIntroductionInterceptor (org.springframework.aop.support)
  AbstractAspectJAdvice (org.springframework.aop.aspectj)
    AspectJAfterAdvice (org.springframework.aop.aspectj)
    AspectJAfterReturningAdvice (org.springframework.aop.aspectj)
    AspectJAroundAdvice (org.springframework.aop.aspectj)
    AspectJAfterThrowingAdvice (org.springframework.aop.aspectj)
    AspectJMethodBeforeAdvice (org.springframework.aop.aspectj)
  AfterAdvice (org.springframework.aop)
    ThrowsAdvice (org.springframework.aop)
    AfterReturningAdviceInterceptor (org.springframework.aop.framework.adapter)
    AspectJAfterAdvice (org.springframework.aop.aspectj)
    AspectJAfterReturningAdvice (org.springframework.aop.aspectj)
    AspectJAfterThrowingAdvice (org.springframework.aop.aspectj)
    ThrowsAdviceInterceptor (org.springframework.aop.framework.adapter)
    AfterReturningAdvice (org.springframework.aop)
      AspectJAfterReturningAdvice (org.springframework.aop.aspectj)
  Anonymous in Advisor (org.springframework.aop)
  Anonymous in InstantiationModelAwarePointcutAdvisorImpl (org.springframework.aop.aspectj.annotation)
```

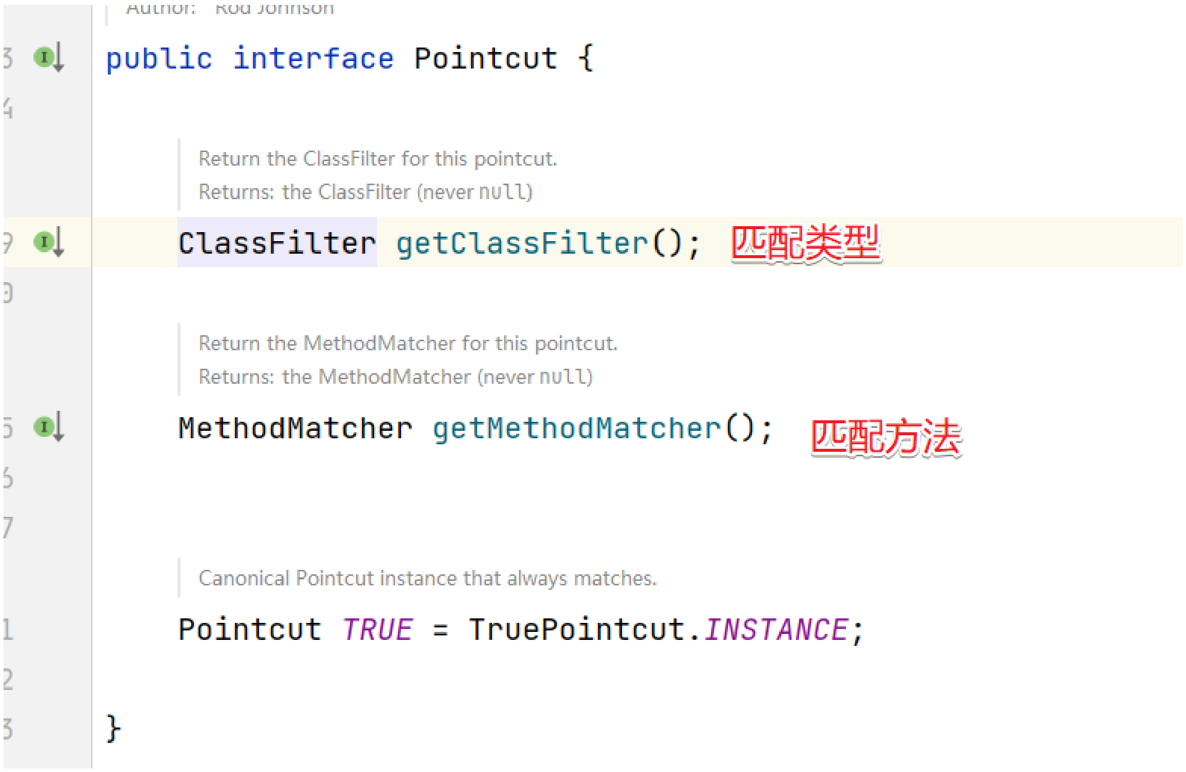
相关的说明



是一种比较特殊的增强类型，它不是在目标方法周围织入增强，而是为目标创建新的方法和属性，所以它的连接点是类级别的而非方法级别的。
通过引介增强我们可以为目标类添加一个接口的实现即原来目标类未实现某个接口,那么通过引介增强可以为目标类创建实现某接口的代理。

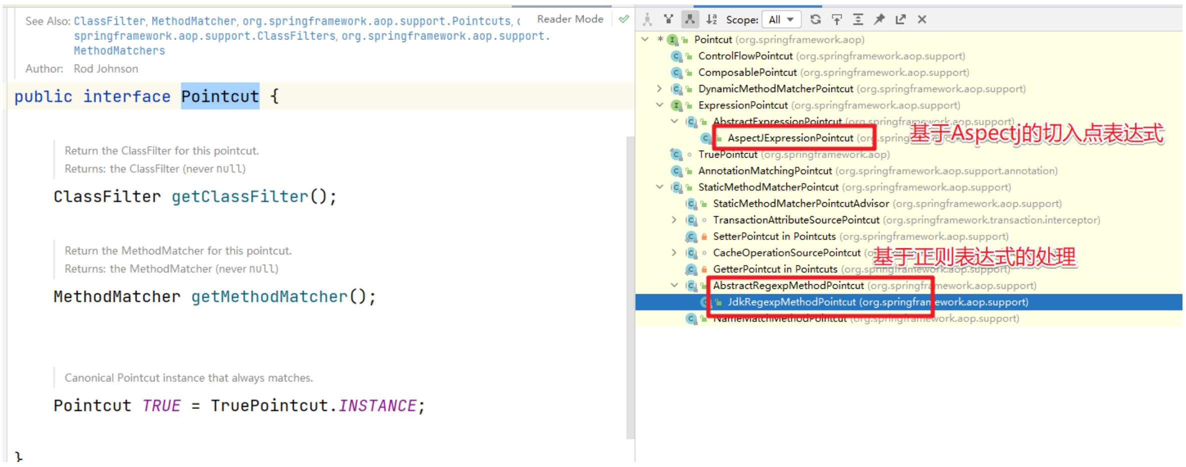
## 2. Pointcut类结构

然后来看看Pointcut的设计，也就是切入点的处理。



```
Author: Rod Johnson
public interface Pointcut {

    Return the ClassFilter for this pointcut.
    Returns: the ClassFilter (never null)
    ClassFilter getClassFilter();

    Return the MethodMatcher for this pointcut.
    Returns: the MethodMatcher (never null)
    MethodMatcher getMethodMatcher();

    Canonical Pointcut instance that always matches.
    Pointcut TRUE = TruePointcut.INSTANCE;

}
```

Pointcut的两种实现方式

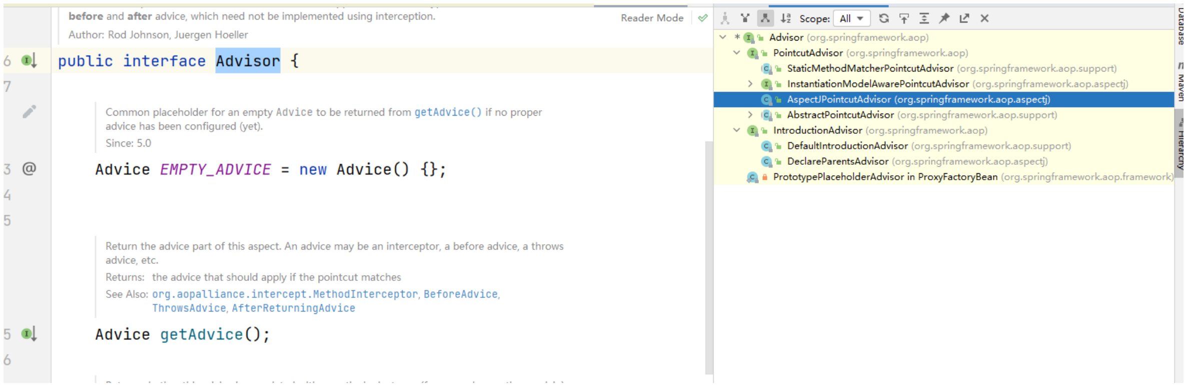


```
See Also: ClassFilter, MethodMatcher, org.springframework.aop.support.Pointcuts, (
          springframework.aop.support.ClassFilters, org.springframework.aop.support.
          MethodMatchers
Author:   Rod Johnson
public interface Pointcut {

    Return the ClassFilter for this pointcut.
    Returns: the ClassFilter (never null)
    ClassFilter getClassFilter();

    Return the MethodMatcher for this pointcut.
    Returns: the MethodMatcher (never null)
    MethodMatcher getMethodMatcher();

    Canonical Pointcut instance that always matches.
    Pointcut TRUE = TruePointcut.INSTANCE;

}
```

```
Scope: All
Pointcut (org.springframework.aop)
  ControlFlowPointcut (org.springframework.aop.support)
  ComposablePointcut (org.springframework.aop.support)
  DynamicMethodMatcherPointcut (org.springframework.aop.support)
  ExpressionPointcut (org.springframework.aop.support)
    AbstractExpressionPointcut (org.springframework.aop.support)
      AspectJExpressionPointcut (org.spring...
  TruePointcut (org.springframework.aop)
  AnnotationMatchingPointcut (org.springframework.aop.support.annotation)
  StaticMethodMatcherPointcut (org.springframework.aop.support)
    StaticMethodMatcherPointcutAdvisor (org.springframework.aop.support)
    TransactionAttributeSourcePointcut (org.springframework.transaction.interceptor)
    SetterPointcut in Pointcuts (org.springframework.aop.support)
    CacheOperationSourcePointcut (o...
    GetterPointcut in Pointcuts (org.springframework.aop.support)
    AbstractRegexpMethodPointcut (org.springframework.aop.support)
      JdkRegexpMethodPointcut (org.springframework.aop.support)
    NameMatchMethodPointcut (org.springframework.aop.support)
```

## 3. Advisor类结构

Advisor的类结构比较简单。一个是PointcutAdvisor,一个是IntroductionAdvisor



```
before and after advice, which need not be implemented using interception.
Author: Rod Johnson, Juergen Hoeller
public interface Advisor {

    Common placeholder for an empty Advice to be returned from getAdvice() if no proper
    advice has been configured (yet).
    Since: 5.0
    Advice EMPTY_ADVICE = new Advice() {};


    Return the advice part of this aspect. An advice may be an interceptor, a before advice, a throws
    advice, etc.
    Returns:  the advice that should apply if the pointcut matches
    See Also: org.aopalliance.intercept.MethodInterceptor, BeforeAdvice,
              ThrowsAdvice, AfterReturningAdvice
    Advice getAdvice();
```

```
Scope: All
Advisor (org.springframework.aop)
  PointcutAdvisor (org.springframework.aop)
    StaticMethodMatcherPointcutAdvisor (org.springframework.aop.support)
    InstantiationModelAwarePointcutAdvisor (org.springframework.aop.aspectj)
    AspectJPointcutAdvisor (org.springframework.aop.aspectj)
    AbstractPointcutAdvisor (org.springframework.aop.support)
  IntroductionAdvisor (org.springframework.aop)
    DefaultIntroductionAdvisor (org.springframework.aop.support)
    DeclareParentsAdvisor (org.springframework.aop.aspectj)
  PrototypePlaceholderAdvisor in ProxyFactoryBean (org.springframework.aop.framework)
```

我们要看的重点是 PointcutAdvisor 及实现 AspectJPointcutAdvisor。

## 三、织入的实现

# 1. BeanPostProcessor

## 1.1 案例演示

我们通过案例来看，首先使用AOP来增强。

定义切面类

```java
/**
 * 切面类
 */
@Component
@EnableAspectJAutoProxy
@Aspect
public class AspectAdviceBeanUseAnnotation {

    // 定义一个全局的Pointcut
    @Pointcut("execution(* com.study.spring.sample.aop.*.do*(..))")
    public void doMethods() {
    }

    @Pointcut("execution(* com.study.spring.sample.aop.*.service*(..))")
    public void services() {
    }

    // 定义一个Before Advice
    @Before("doMethods() and args(tk,..)")
    public void before3(String tk) {
        System.out.println("----------- AspectAdviceBeanUseAnnotation before3
增强  参数tk= " + tk);
    }

    @Around("services() and args(name,..)")
    public Object around2(ProceedingJoinPoint pjp, String name) throws Throwable
{
        System.out.println("--------- AspectAdviceBeanUseAnnotation arround2 参数
name=" + name);
        System.out.println("----------- AspectAdviceBeanUseAnnotation arround2
环绕-前增强 for " + pjp);
        Object ret = pjp.proceed();
        System.out.println("----------- AspectAdviceBeanUseAnnotation arround2
环绕-后增强 for " + pjp);
        return ret;
    }

    @AfterReturning(pointcut = "services()", returning = "retValue")
    public void afterReturning(Object retValue) {
        System.out.println("----------- AspectAdviceBeanUseAnnotation
afterReturning 增强 , 返回值为:  " + retValue);
    }

    @AfterThrowing(pointcut = "services()", throwing = "e")
    public void afterThrowing(JoinPoint jp, Exception e) {
        System.out.println("----------- AspectAdviceBeanUseAnnotation
afterThrowing 增强  for " + jp);
        System.out.println("----------- AspectAdviceBeanUseAnnotation
afterThrowing 增强  异常 : " + e);
```

```
    }

    @After("doMethods()")
    public void after(JoinPoint jp) {
        System.out.println("----------- AspectAdviceBeanUseAnnotation after 增强
for " + jp);
    }

    /*
     * BeanDefinitionRegistryPostProcessor BeanFactoryPostProcessor
     * InstantiationAwareBeanPostProcessor Bean实例创建前后 BeanPostProcessor
     */
}
```

需要增强的目标类

```
@Component
public class BeanQ {

    public void do1(String task, int time) {
        System.out.println("-------------do1 do " + task + " time:" + time);
    }

    public String service1(String name) {
        System.out.println("-------------servce1 do " + name);
        return name;
    }

    public String service2(String name) {
        System.out.println("-------------servce2 do " + name);
        if (!"s1".equals(name)) {
            throw new IllegalArgumentException("参数 name ! = s1, name=" + name);
        }

        return name + " hello!";
    }
}
```

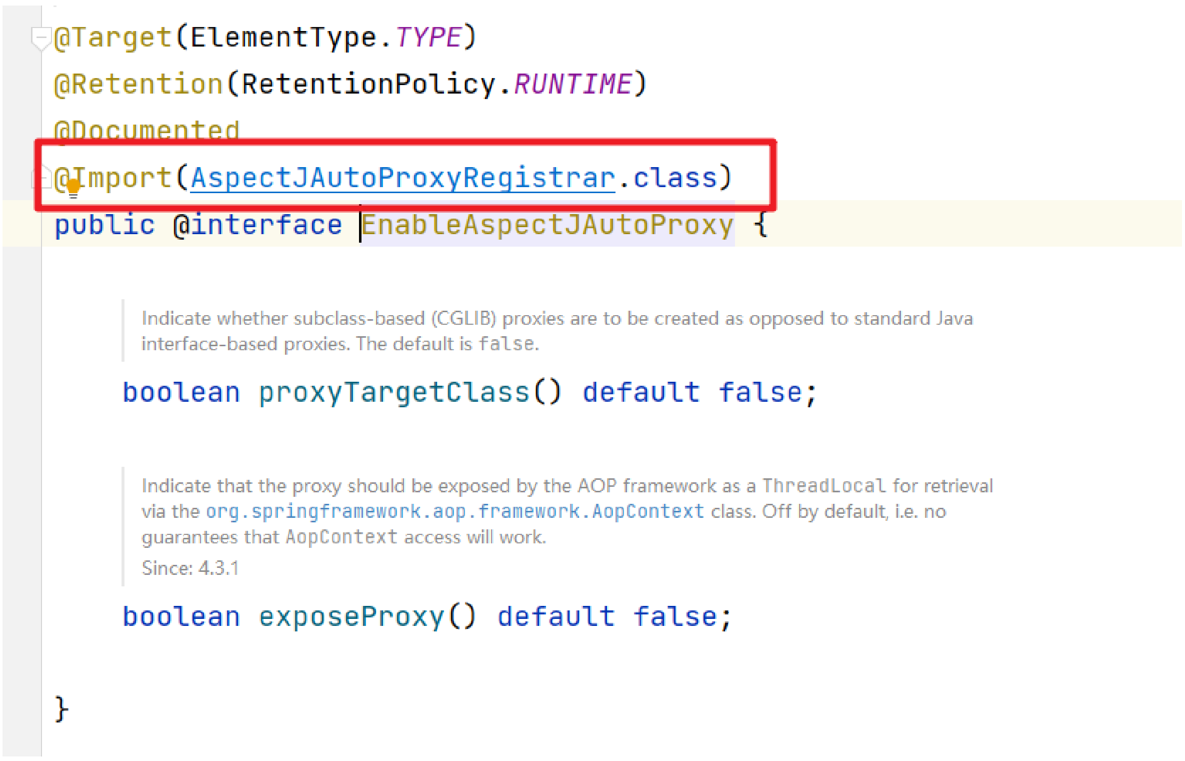
测试代码

```
@Configuration
@ComponentScan
public class AopMainAnno {
    public static void main(String[] args) {
        ApplicationContext context = new
AnnotationConfigApplicationContext(AopMainAnno.class);
        BeanQ bq = context.getBean(BeanQ.class);
        bq.do1("task1", 20);
        System.out.println();

        bq.service1("service1");

        System.out.println();
        bq.service2("s1");
    }
}
```

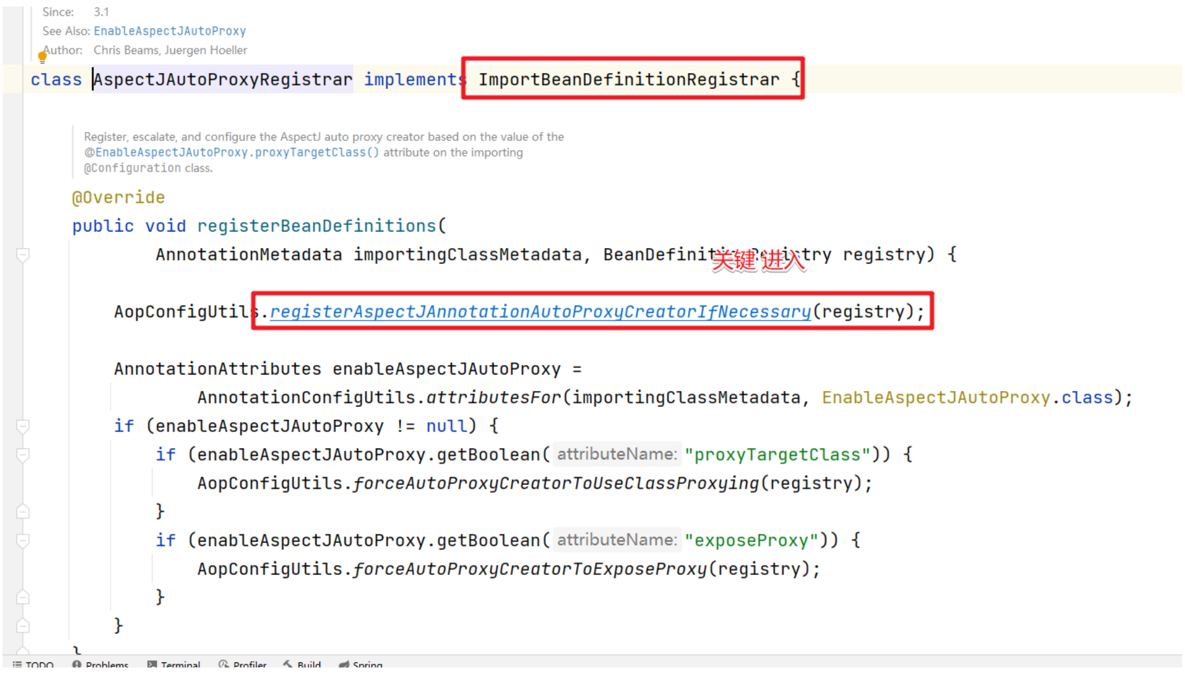
执行即可看到增强的效果

## 1.2 @EnableAspectJAutoProxy

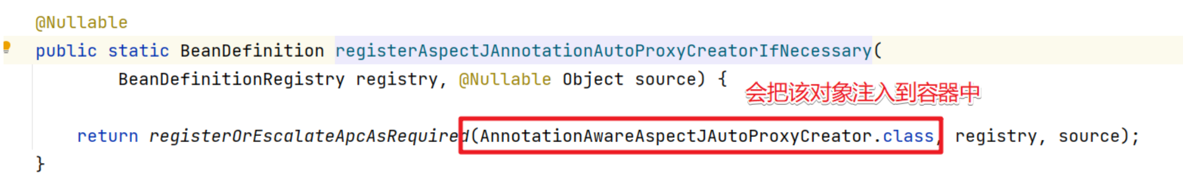
我们需要使用代理增强处理，必须添加@EnableAspectJAutoProxy才生效。我们来看看他做了什么事情



```java
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(AspectJAutoProxyRegistrar.class)
public @interface EnableAspectJAutoProxy {

    Indicate whether subclass-based (CGLIB) proxies are to be created as opposed to standard Java
    interface-based proxies. The default is false.
    boolean proxyTargetClass() default false;

    Indicate that the proxy should be exposed by the AOP framework as a ThreadLocal for retrieval
    via the org.springframework.aop.framework.AopContext class. Off by default, i.e. no
    guarantees that AopContext access will work.
    Since: 4.3.1
    boolean exposeProxy() default false;

}
```

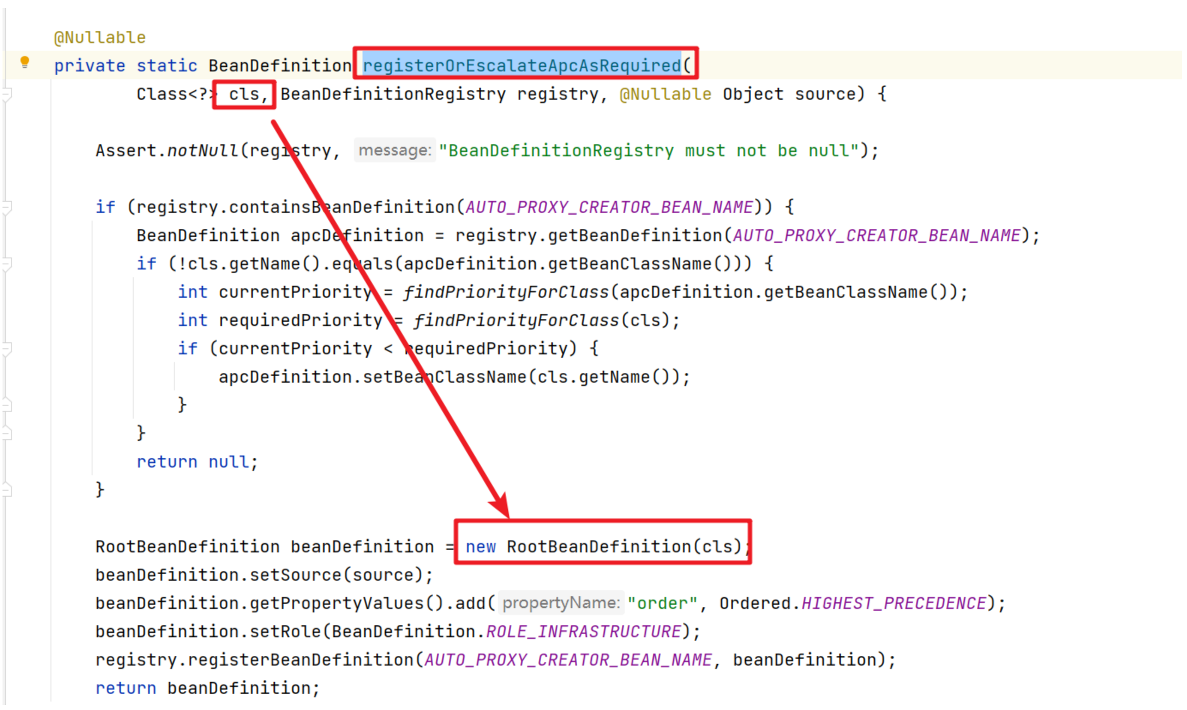


```java
Since:    3.1
See Also: EnableAspectJAutoProxy
Author:   Chris Beams, Juergen Hoeller
class AspectJAutoProxyRegistrar implements ImportBeanDefinitionRegistrar {

    Register, escalate, and configure the AspectJ auto proxy creator based on the value of the
    @EnableAspectJAutoProxy.proxyTargetClass() attribute on the importing
    @Configuration class.
    @Override
    public void registerBeanDefinitions(
            AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {

        AopConfigUtils.registerAspectJAnnotationAutoProxyCreatorIfNecessary(registry);

        AnnotationAttributes enableAspectJAutoProxy =
                AnnotationConfigUtils.attributesFor(importingClassMetadata, EnableAspectJAutoProxy.class);
        if (enableAspectJAutoProxy != null) {
            if (enableAspectJAutoProxy.getBoolean( attributeName: "proxyTargetClass")) {
                AopConfigUtils.forceAutoProxyCreatorToUseClassProxying(registry);
            }
            if (enableAspectJAutoProxy.getBoolean( attributeName: "exposeProxy")) {
                AopConfigUtils.forceAutoProxyCreatorToExposeProxy(registry);
            }
        }
```



```java
@Nullable
public static BeanDefinition registerAspectJAnnotationAutoProxyCreatorIfNecessary(
        BeanDefinitionRegistry registry, @Nullable Object source) {

    return registerOrEscalateApcAsRequired(AnnotationAwareAspectJAutoProxyCreator.class, registry, source);
}
```

在registerOrEscalateApcAsRequired方法中会把上面的Java类注入到容器中。

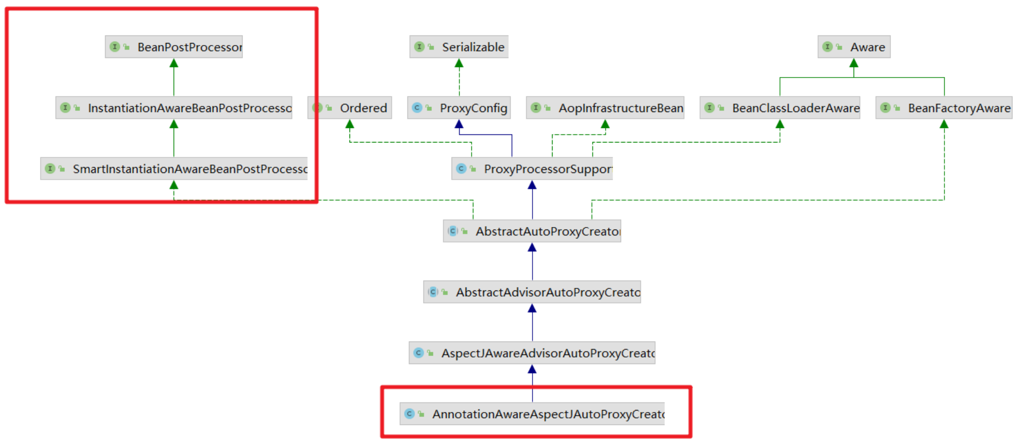


```java
@Nullable
private static BeanDefinition registerOrEscalateApcAsRequired(
        Class<?> cls, BeanDefinitionRegistry registry, @Nullable Object source) {

    Assert.notNull(registry, message: "BeanDefinitionRegistry must not be null");

    if (registry.containsBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME)) {
        BeanDefinition apcDefinition = registry.getBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME);
        if (!cls.getName().equals(apcDefinition.getBeanClassName())) {
            int currentPriority = findPriorityForClass(apcDefinition.getBeanClassName());
            int requiredPriority = findPriorityForClass(cls);
            if (currentPriority < requiredPriority) {
                apcDefinition.setBeanClassName(cls.getName());
            }
        }
        return null;
    }

    RootBeanDefinition beanDefinition = new RootBeanDefinition(cls);
    beanDefinition.setSource(source);
    beanDefinition.getPropertyValues().add( propertyName: "order", Ordered.HIGHEST_PRECEDENCE);
    beanDefinition.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
    registry.registerBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME, beanDefinition);
    return beanDefinition;
```

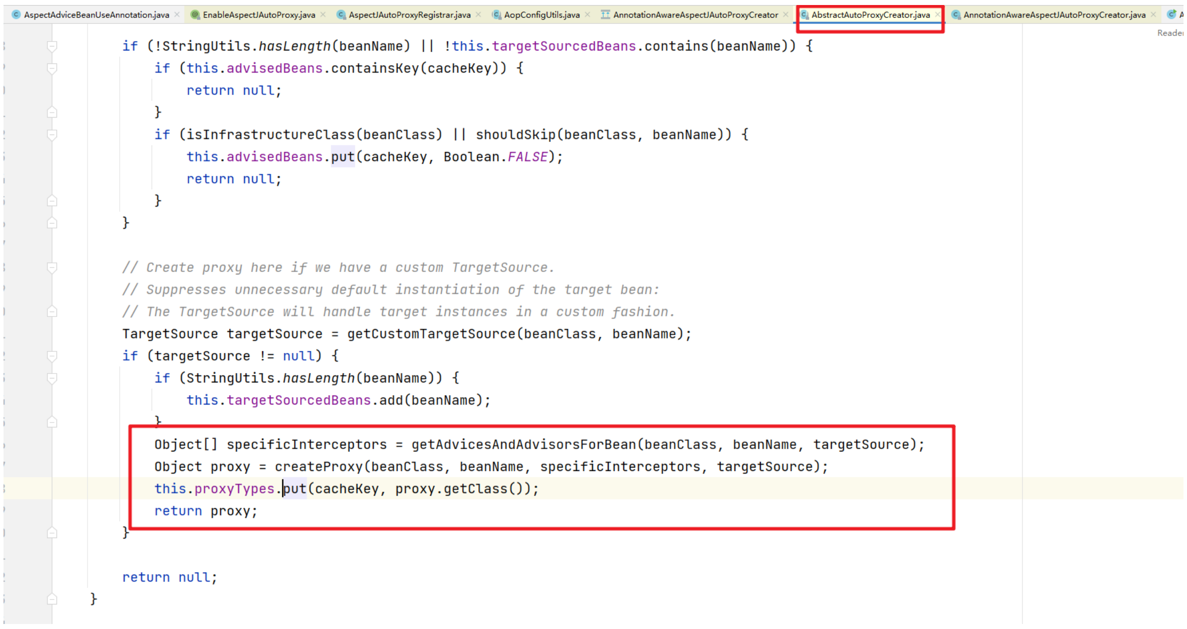
所以我们需要看看 AnnotationAwareAspectJAutoProxyCreator 的结构

## 1.3 AnnotationAwareAspectJAutoProxyCreator

我们直接来看类图结构，可以发现其本质就是一个 BeanPostProcessor ,只是扩展了更多的功能。


BeanPostProcessor
Serializable
Aware
InstantiationAwareBeanPostProcesso
Ordered
ProxyConfig
AopInfrastructureBean
BeanClassLoaderAware
BeanFactoryAware
SmartInstantiationAwareBeanPostProcessc
ProxyProcessorSuppor
AbstractAutoProxyCreator
AbstractAdvisorAutoProxyCreato
AspectJAwareAdvisorAutoProxyCreato
AnnotationAwareAspectJAutoProxyCreato

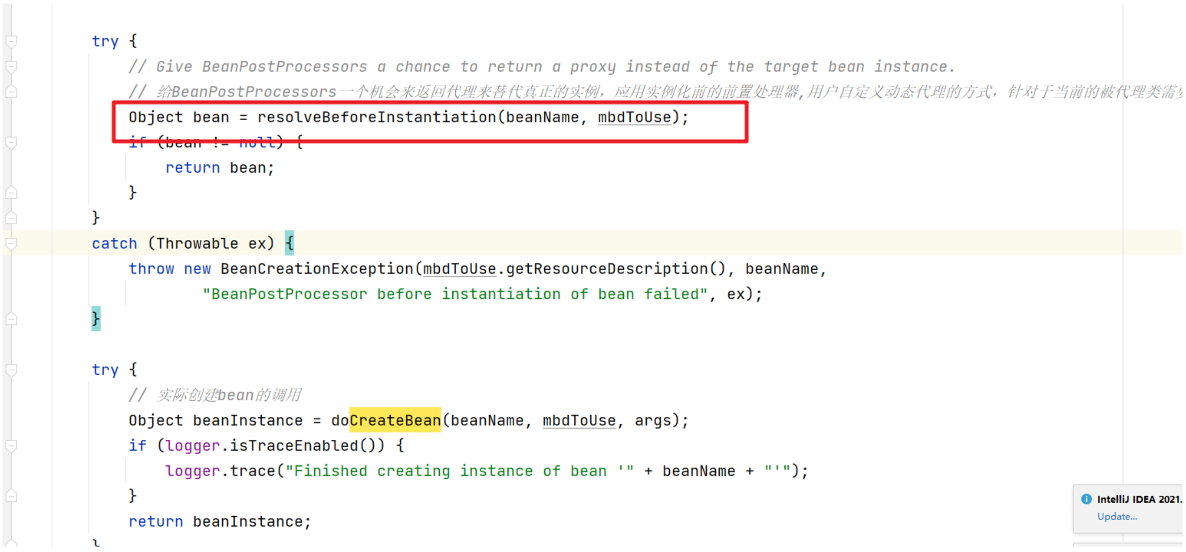
那么具体处理的逻辑


AspectAdviceBeanUseAnnotation.java
EnableAspectJAutoProxy.java
AspectJAutoProxyRegistrar.java
AopConfigUtils.java
AnnotationAwareAspectJAutoProxyCreator
AbstractAutoProxyCreator.java
AnnotationAwareAspectJAutoProxyCreator.java
Reade

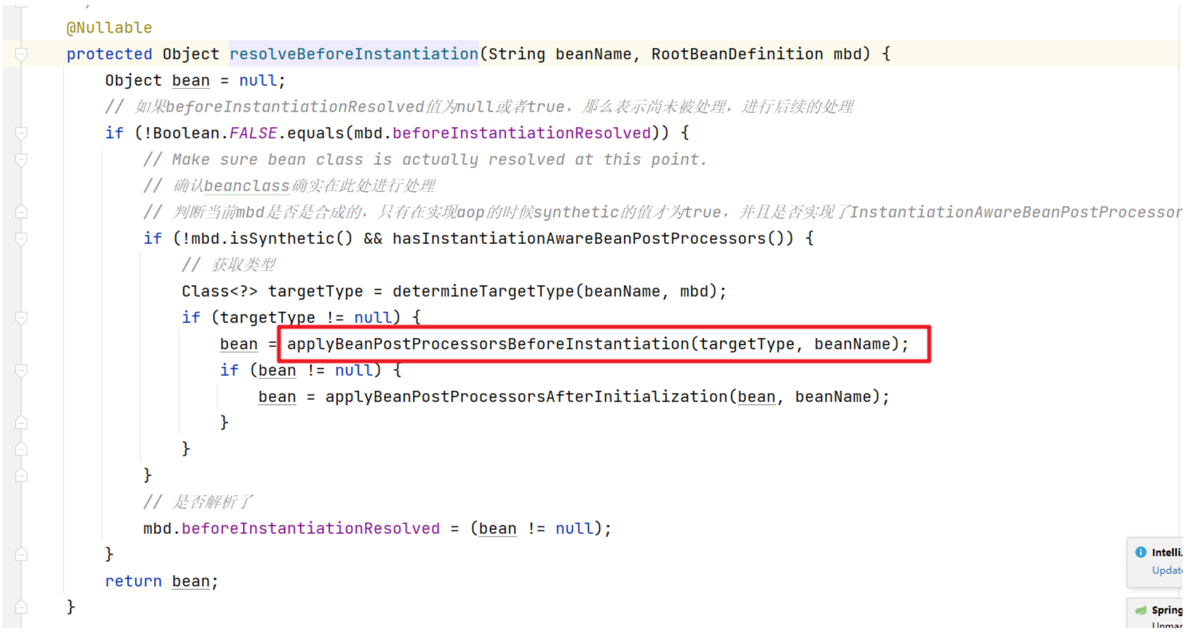
```java
        if (!StringUtils.hasLength(beanName) || !this.targetSourcedBeans.contains(beanName)) {
            if (this.advisedBeans.containsKey(cacheKey)) {
                return null;
            }
            if (isInfrastructureClass(beanClass) || shouldSkip(beanClass, beanName)) {
                this.advisedBeans.put(cacheKey, Boolean.FALSE);
                return null;
            }
        }

        // Create proxy here if we have a custom TargetSource.
        // Suppresses unnecessary default instantiation of the target bean:
        // The TargetSource will handle target instances in a custom fashion.
        TargetSource targetSource = getCustomTargetSource(beanClass, beanName);
        if (targetSource != null) {
            if (StringUtils.hasLength(beanName)) {
                this.targetSourcedBeans.add(beanName);
            }
            Object[] specificInterceptors = getAdvicesAndAdvisorsForBean(beanClass, beanName, targetSource);
            Object proxy = createProxy(beanClass, beanName, specificInterceptors, targetSource);
            this.proxyTypes.put(cacheKey, proxy.getClass());
            return proxy;
        }

        return null;
    }
```

## 1.4 如何串联

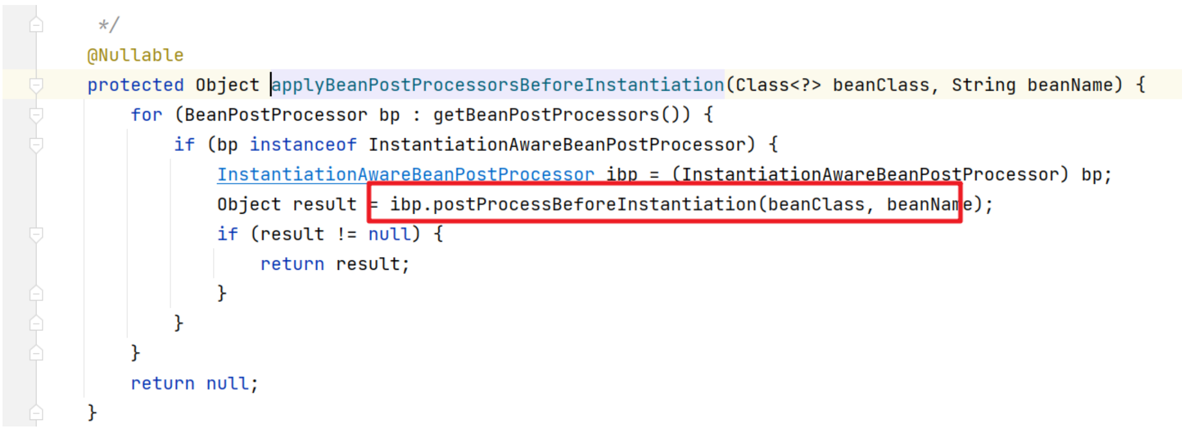
Bean的IoC是如何和对应的BeanPostProcessor串联的呢？我们来看看。



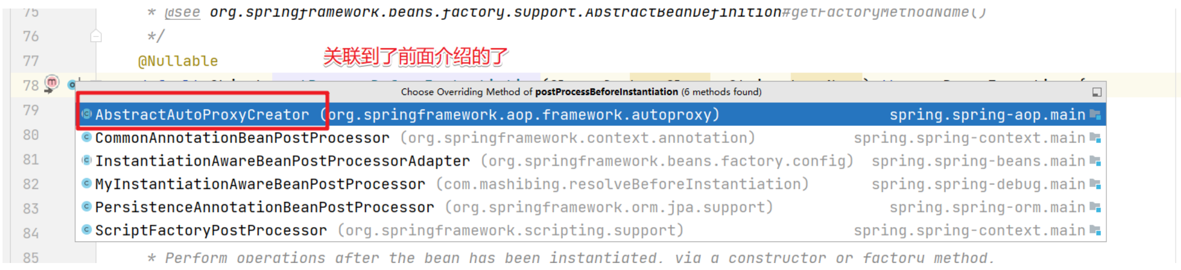
```java
    try {
        // Give BeanPostProcessors a chance to return a proxy instead of the target bean instance.
        // 给BeanPostProcessors一个机会来返回代理来替代真正的实例，应用实例化前的前置处理器,用户自定义动态代理的方式，针对于当前的被代理类需
        Object bean = resolveBeforeInstantiation(beanName, mbdToUse);
        if (bean != null) {
            return bean;
        }
    }
    catch (Throwable ex) {
        throw new BeanCreationException(mbdToUse.getResourceDescription(), beanName,
                "BeanPostProcessor before instantiation of bean failed", ex);
    }

    try {
        // 实际创建bean的调用
        Object beanInstance = doCreateBean(beanName, mbdToUse, args);
        if (logger.isTraceEnabled()) {
            logger.trace("Finished creating instance of bean '" + beanName + "'");
        }
        return beanInstance;
```



```java
@Nullable
protected Object resolveBeforeInstantiation(String beanName, RootBeanDefinition mbd) {
    Object bean = null;
    // 如果beforeInstantiationResolved值为null或者true，那么表示尚未被处理，进行后续的处理
    if (!Boolean.FALSE.equals(mbd.beforeInstantiationResolved)) {
        // Make sure bean class is actually resolved at this point.
        // 确认beanclass确实在此处进行处理
        // 判断当前mbd是否是合成的，只有在实现aop的时候synthetic的值才为true，并且是否实现了InstantiationAwareBeanPostProcessor
        if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
            // 获取类型
            Class<?> targetType = determineTargetType(beanName, mbd);
            if (targetType != null) {
                bean = applyBeanPostProcessorsBeforeInstantiation(targetType, beanName);
                if (bean != null) {
                    bean = applyBeanPostProcessorsAfterInitialization(bean, beanName);
                }
            }
        }
        // 是否解析了
        mbd.beforeInstantiationResolved = (bean != null);
    }
    return bean;
}
```



```java
	 */
	@Nullable
	protected Object applyBeanPostProcessorsBeforeInstantiation(Class<?> beanClass, String beanName) {
		for (BeanPostProcessor bp : getBeanPostProcessors()) {
			if (bp instanceof InstantiationAwareBeanPostProcessor) {
				InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;
				Object result = ibp.postProcessBeforeInstantiation(beanClass, beanName);
				if (result != null) {
					return result;
				}
			}
		}
		return null;
	}
```

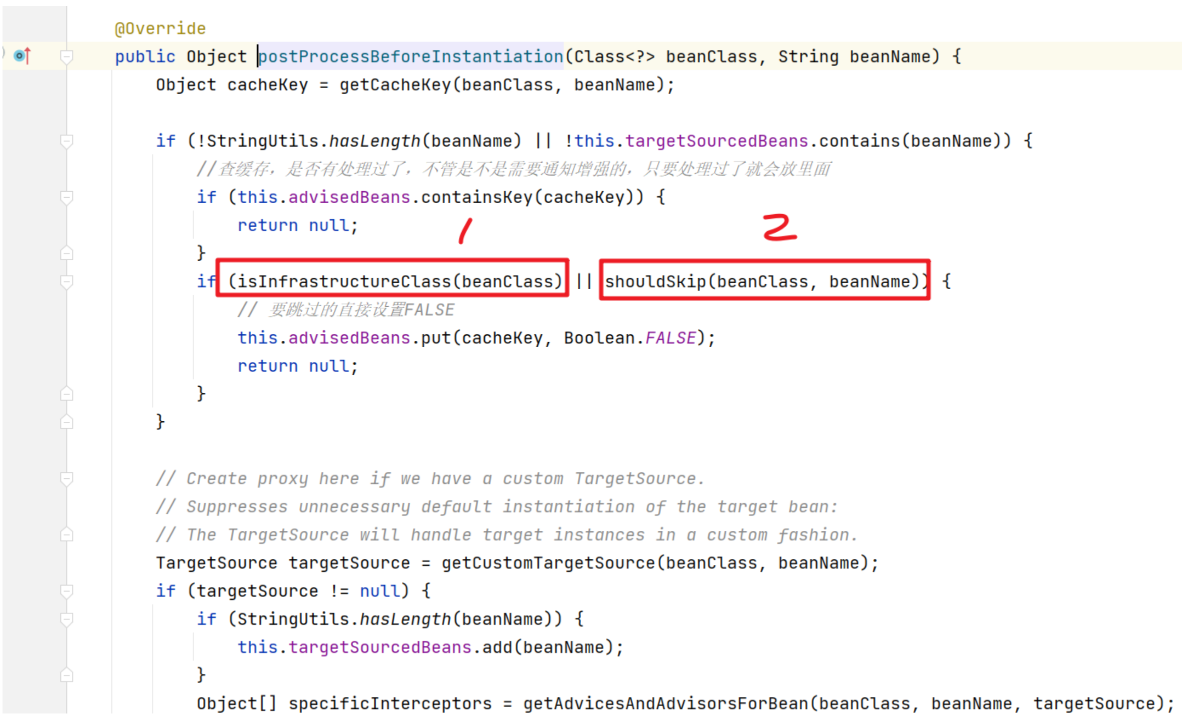


```java
	 * @see org.springframework.beans.factory.support.AbstractBeanDefinition#getFactoryMethodName()
	 */
	@Nullable
```

关联到了前面介绍的了

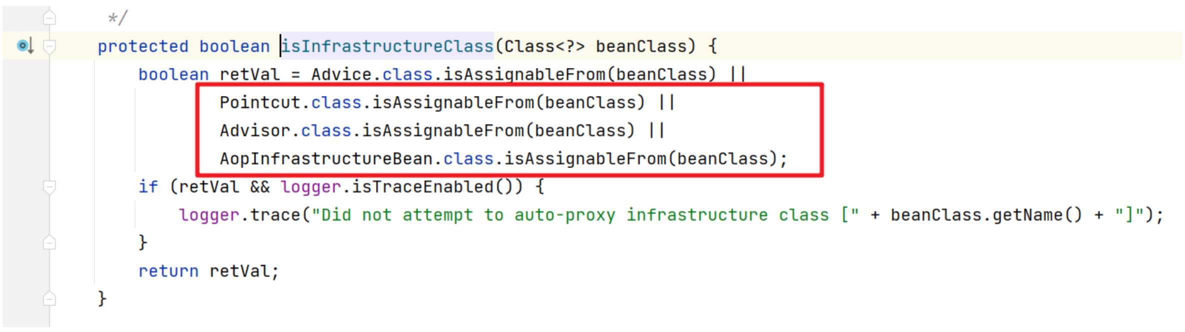
```
Choose Overriding Method of postProcessBeforeInstantiation (6 methods found)
AbstractAutoProxyCreator (org.springframework.aop.framework.autoproxy)                        spring.spring-aop.main
CommonAnnotationBeanPostProcessor (org.springframework.context.annotation)               spring.spring-context.main
InstantiationAwareBeanPostProcessorAdapter (org.springframework.beans.factory.config)   spring.spring-beans.main
MyInstantiationAwareBeanPostProcessor (com.mashibing.resolveBeforeInstantiation)          spring.spring-debug.main
PersistenceAnnotationBeanPostProcessor (org.springframework.orm.jpa.support)                spring.spring-orm.main
ScriptFactoryPostProcessor (org.springframework.scripting.support)                       spring.spring-context.main
```

```java
	 * Perform operations after the bean has been instantiated, via a constructor or factory method.
```



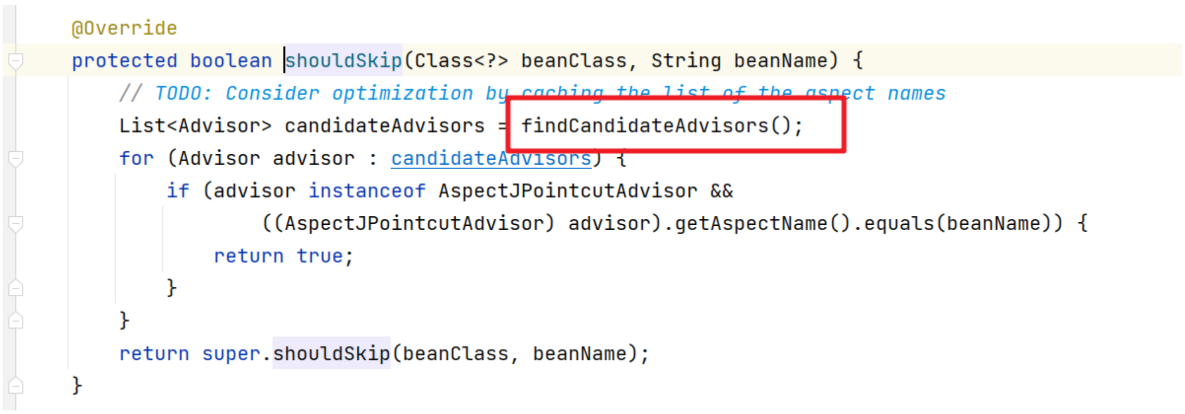
```java
	@Override
	public Object postProcessBeforeInstantiation(Class<?> beanClass, String beanName) {
		Object cacheKey = getCacheKey(beanClass, beanName);

		if (!StringUtils.hasLength(beanName) || !this.targetSourcedBeans.contains(beanName)) {
			//查缓存，是否有处理过了，不管是不是需要通知增强的，只要处理过了就会放里面
			if (this.advisedBeans.containsKey(cacheKey)) {
				return null;
			}
			if (isInfrastructureClass(beanClass) || shouldSkip(beanClass, beanName)) {
				// 要跳过的直接设置FALSE
				this.advisedBeans.put(cacheKey, Boolean.FALSE);
				return null;
			}
		}

		// Create proxy here if we have a custom TargetSource.
		// Suppresses unnecessary default instantiation of the target bean:
		// The TargetSource will handle target instances in a custom fashion.
		TargetSource targetSource = getCustomTargetSource(beanClass, beanName);
		if (targetSource != null) {
			if (StringUtils.hasLength(beanName)) {
				this.targetSourcedBeans.add(beanName);
			}
			Object[] specificInterceptors = getAdvicesAndAdvisorsForBean(beanClass, beanName, targetSource);
```

isInfrastructureClass方法判断是否是基础设施



```java
	 */
	protected boolean isInfrastructureClass(Class<?> beanClass) {
		boolean retVal = Advice.class.isAssignableFrom(beanClass) ||
				Pointcut.class.isAssignableFrom(beanClass) ||
				Advisor.class.isAssignableFrom(beanClass) ||
				AopInfrastructureBean.class.isAssignableFrom(beanClass);
		if (retVal && logger.isTraceEnabled()) {
			logger.trace("Did not attempt to auto-proxy infrastructure class [" + beanClass.getName() + "]");
		}
		return retVal;
	}
```

shouldSkip：是否应该跳过，会完成相关的advisor的收集



```java
@Override
protected boolean shouldSkip(Class<?> beanClass, String beanName) {
    // TODO: Consider optimization by caching the list of the aspect names
    List<Advisor> candidateAdvisors = findCandidateAdvisors();
    for (Advisor advisor : candidateAdvisors) {
        if (advisor instanceof AspectJPointcutAdvisor &&
                ((AspectJPointcutAdvisor) advisor).getAspectName().equals(beanName)) {
            return true;
        }
    }
    return super.shouldSkip(beanClass, beanName);
}
```

具体的处理流程

```java
    public List<Advisor> findAdvisorBeans() {
        // Determine list of advisor bean names, if not cached already.
        String[] advisorNames = this.cachedAdvisorBeanNames;
        if (advisorNames == null) {
            // Do not initialize FactoryBeans here: We need to leave all regular
beans
            // uninitialized to let the auto-proxy creator apply to them!
            // 获取当前BeanFactory中所有实现了Advisor接口的bean的名称
            advisorNames = BeanFactoryUtils.beanNamesForTypeIncludingAncestors(
                    this.beanFactory, Advisor.class, true, false);
            this.cachedAdvisorBeanNames = advisorNames;
        }
        if (advisorNames.length == 0) {
            return new ArrayList<>();
        }

        // 对获取到的实现Advisor接口的bean的名称进行遍历
        List<Advisor> advisors = new ArrayList<>();
        // 循环所有的beanName，找出对应的增强方法
        for (String name : advisorNames) {
            // isEligibleBean()是提供的一个hook方法，用于子类对Advisor进行过滤，这里默认
返回值都是true
            if (isEligibleBean(name)) {
                // 如果当前bean还在创建过程中，则略过，其创建完成之后会为其判断是否需要织入
切面逻辑
                if (this.beanFactory.isCurrentlyInCreation(name)) {
                    if (logger.isTraceEnabled()) {
                        logger.trace("Skipping currently created advisor '" +
name + "'");
                    }
                }
                else {
                    try {
                        // 将当前bean添加到结果中
                        advisors.add(this.beanFactory.getBean(name,
Advisor.class));
                    }
                    catch (BeanCreationException ex) {
                        // 对获取过程中产生的异常进行封装
                        Throwable rootCause = ex.getMostSpecificCause();
                        if (rootCause instanceof
BeanCurrentlyInCreationException) {
```

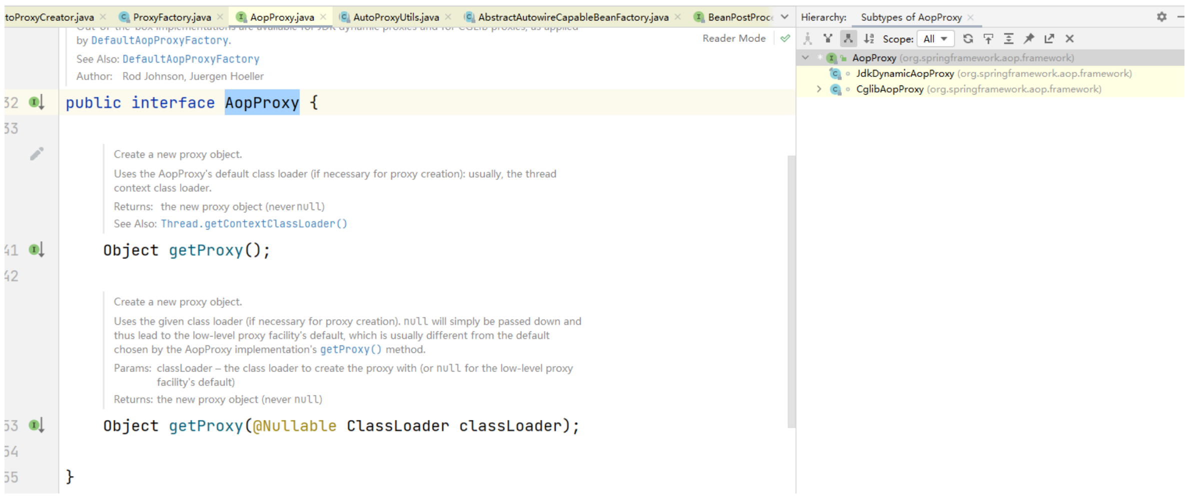
```
                            BeanCreationException bce = (BeanCreationException)
rootCause;
                            String bceBeanName = bce.getBeanName();
                            if (bceBeanName != null &&
this.beanFactory.isCurrentlyInCreation(bceBeanName)) {
                                if (logger.isTraceEnabled()) {
                                    logger.trace("Skipping advisor '" + name +
                                            "' with dependency on currently
created bean: " + ex.getMessage());
                                }
                                // Ignore: indicates a reference back to the
bean we're trying to advise.
                                // We want to find advisors other than the
currently created bean itself.
                                continue;
                            }
                        }
                        throw ex;
                    }
                }
            }
        }
        return advisors;
    }
```

## 2. 代理类的结构

在上面的分析中出现了很多代理相关的代码，为了更好的理解，我们来梳理下Spring中的代理相关的结构

### 2.1 AopProxy

在Spring中创建代理对象都是通过AopProxy这个接口的两个具体实现类来实现的，也就是jdk和cglib两种方式。

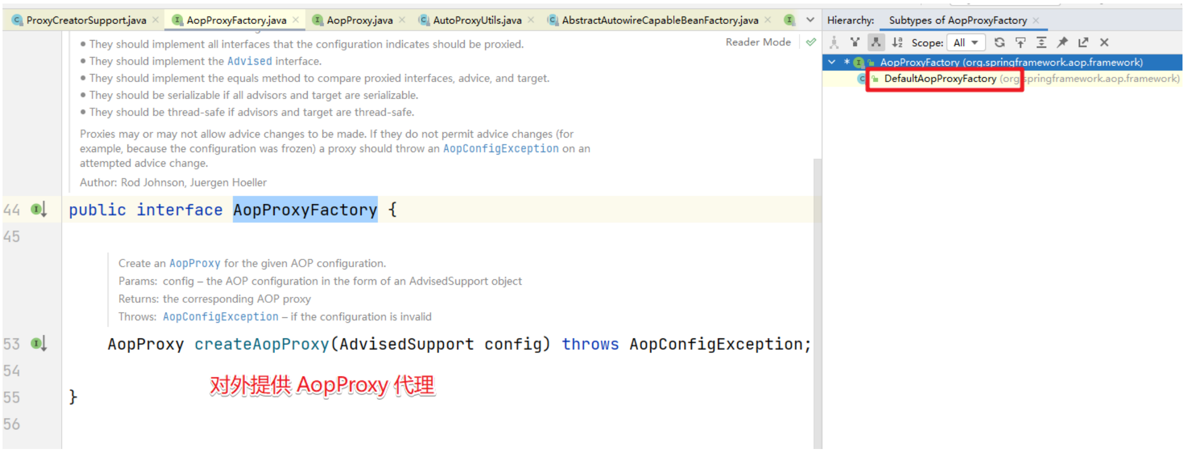


```
by DefaultAopProxyFactory.
See Also: DefaultAopProxyFactory
Author:   Rod Johnson, Juergen Hoeller
public interface AopProxy {

    Create a new proxy object.
    Uses the AopProxy's default class loader (if necessary for proxy creation): usually, the thread
    context class loader.
    Returns:  the new proxy object (never null)
    See Also: Thread.getContextClassLoader()
    Object getProxy();

    Create a new proxy object.
    Uses the given class loader (if necessary for proxy creation). null will simply be passed down and
    thus lead to the low-level proxy facility's default, which is usually different from the default
    chosen by the AopProxy implementation's getProxy() method.
    Params: classLoader – the class loader to create the proxy with (or null for the low-level proxy
            facility's default)
    Returns: the new proxy object (never null)
    Object getProxy(@Nullable ClassLoader classLoader);

}
```

```
AopProxy (org.springframework.aop.framework)
    JdkDynamicAopProxy (org.springframework.aop.framework)
    CglibAopProxy (org.springframework.aop.framework)
```

### 2.2 AopProxyFactory

在Spring中通过AopProxyFactory这个工厂类来提供AopProxy。



```
• They should implement all interfaces that the configuration indicates should be proxied.
• They should implement the Advised interface.
• They should implement the equals method to compare proxied interfaces, advice, and target.
• They should be serializable if all advisors and target are serializable.
• They should be thread-safe if advisors and target are thread-safe.
Proxies may or may not allow advice changes to be made. If they do not permit advice changes (for
example, because the configuration was frozen) a proxy should throw an AopConfigException on an
attempted advice change.
Author: Rod Johnson, Juergen Hoeller
public interface AopProxyFactory {

    Create an AopProxy for the given AOP configuration.
    Params: config – the AOP configuration in the form of an AdvisedSupport object
    Returns: the corresponding AOP proxy
    Throws: AopConfigException – if the configuration is invalid
    AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException;

}
```

AopProxyFactory (org.springframework.aop.framework)
DefaultAopProxyFactory (org.springframework.aop.framework)

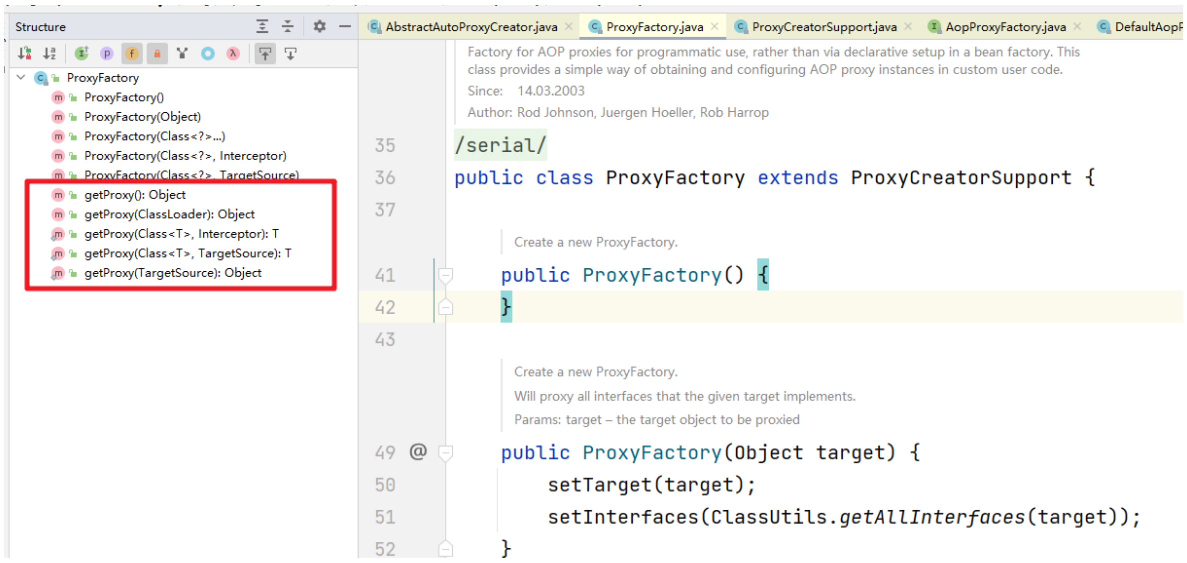
对外提供 AopProxy 代理

默认的实现类是DefaultAopProxyFactory

```
    /**
     * 真正的创建代理，判断一些列条件，有自定义的接口的就会创建jdk代理，否则就是cglib
     * @param config the AOP configuration in the form of an
     * AdvisedSupport object
     * @return
     * @throws AopConfigException
     */
    @Override
    public AopProxy createAopProxy(AdvisedSupport config) throws
AopConfigException {
        // 这段代码用来判断选择哪种创建代理对象的方式
        // config.isOptimize()   是否对代理类的生成使用策略优化 其作用是和
isProxyTargetClass是一样的 默认为false
        // config.isProxyTargetClass() 是否使用Cglib的方式创建代理对象 默认为false
        // hasNoUserSuppliedProxyInterfaces目标类是否有接口存在 且只有一个接口的时候接
口类型不是SpringProxy类型
        if (config.isOptimize() || config.isProxyTargetClass() ||
hasNoUserSuppliedProxyInterfaces(config)) {
            // 上面的三个方法有一个为true的话，则进入到这里
            // 从AdvisedSupport中获取目标类 类对象
            Class<?> targetClass = config.getTargetClass();
            if (targetClass == null) {
                throw new AopConfigException("TargetSource cannot determine
target class: " +
                        "Either an interface or a target is required for proxy
creation.");
            }
            // 判断目标类是否是接口 如果目标类是接口的话，则还是使用JDK的方式生成代理对象
            // 如果目标类是Proxy类型 则还是使用JDK的方式生成代理对象
            if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {
                return new JdkDynamicAopProxy(config);
            }
            // 配置了使用Cglib进行动态代理或者目标类没有接口,那么使用Cglib的方式创建代理对
象
            return new ObjenesisCglibAopProxy(config);
        }
        else {
            // 使用JDK的提供的代理方式生成代理对象
            return new JdkDynamicAopProxy(config);
        }
    }
```

## 2.3 ProxyFactory

ProxyFactory代理对象的工厂类，用来创建代理对象的工厂。



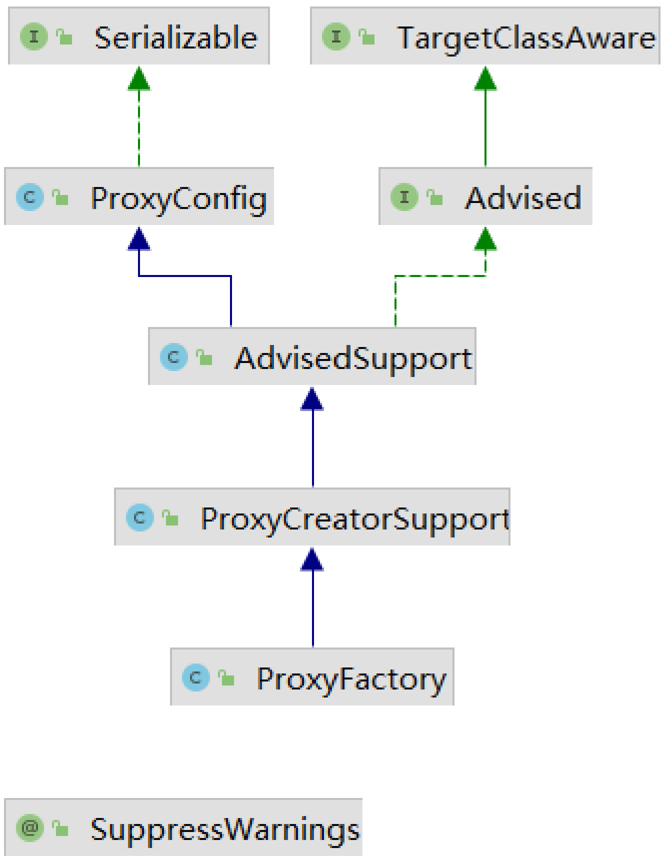
Structure

ProxyFactory
- ProxyFactory()
- ProxyFactory(Object)
- ProxyFactory(Class<?>...)
- ProxyFactory(Class<?>, Interceptor)
- ProxyFactory(Class<?>, TargetSource)
- getProxy(): Object
- getProxy(ClassLoader): Object
- getProxy(Class<T>, Interceptor): T
- getProxy(Class<T>, TargetSource): T
- getProxy(TargetSource): Object

AbstractAutoProxyCreator.java × ProxyFactory.java × ProxyCreatorSupport.java × AopProxyFactory.java × DefaultAopP

```
Factory for AOP proxies for programmatic use, rather than via declarative setup in a bean factory. This
class provides a simple way of obtaining and configuring AOP proxy instances in custom user code.
Since:   14.03.2003
Author: Rod Johnson, Juergen Hoeller, Rob Harrop
/serial/
public class ProxyFactory extends ProxyCreatorSupport {

    Create a new ProxyFactory.
    public ProxyFactory() {
    }

    Create a new ProxyFactory.
    Will proxy all interfaces that the given target implements.
    Params: target – the target object to be proxied
    public ProxyFactory(Object target) {
        setTarget(target);
        setInterfaces(ClassUtils.getAllInterfaces(target));
    }
```

然后我们来看看 ProxyFactory的体系结构



ProxyConfig

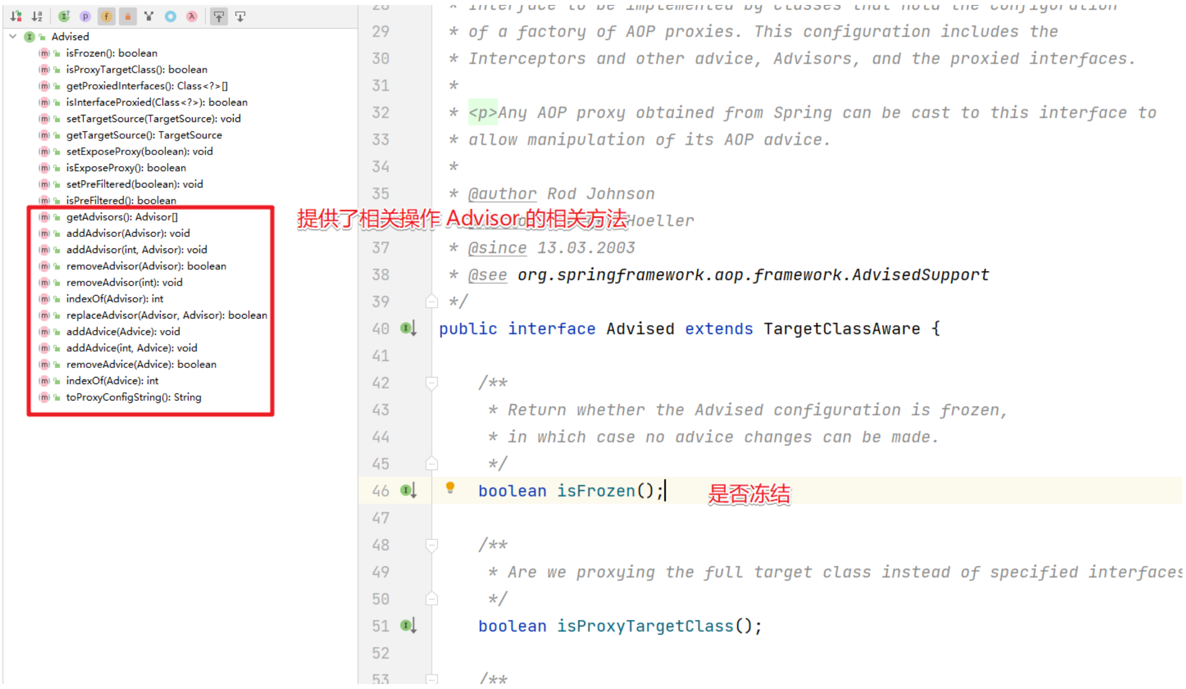
这个类主要保存代理的信息，如果是否使用类代理，是否要暴露代理等。

```
public class ProxyConfig implements Serializable {
```

```java
    /** use serialVersionUID from Spring 1.2 for interoperability. */
    private static final long serialVersionUID = -8409359707199703185L;

    // 是否代理的对象是类，动态代理分为代理接口和类，这里的属性默认是代理的接口
    private boolean proxyTargetClass = false;
    // 是否进行主动优化，默认是不会主动优化
    private boolean optimize = false;
    // 是否由此配置创建的代理不能被转成Advised类型，默认时候可转
    boolean opaque = false;
    // 是否会暴露代理在调用的时候，默认是不会暴露
    boolean exposeProxy = false;
    // 是否冻结此配置，不能被修改
    private boolean frozen = false;

}
```

Advised

由持有 AOP 代理工厂配置的类实现的接口。此配置包括拦截器和其他advice、advisor和代理接口。从 Spring 获得的任何 AOP 代理都可以转换为该接口，以允许操作其 AOP 通知。



```
Advised
    isFrozen(): boolean
    isProxyTargetClass(): boolean
    getProxiedInterfaces(): Class<?>[]
    isInterfaceProxied(Class<?>): boolean
    setTargetSource(TargetSource): void
    getTargetSource(): TargetSource
    setExposeProxy(boolean): void
    isExposeProxy(): boolean
    setPreFiltered(boolean): void
    isPreFiltered(): boolean
    getAdvisors(): Advisor[]
    addAdvisor(Advisor): void
    addAdvisor(int, Advisor): void
    removeAdvisor(Advisor): boolean
    removeAdvisor(int): void
    indexOf(Advisor): int
    replaceAdvisor(Advisor, Advisor): boolean
    addAdvice(Advice): void
    addAdvice(int, Advice): void
    removeAdvice(Advice): boolean
    indexOf(Advice): int
    toProxyConfigString(): String
```

提供了相关操作 Advisor 的相关方法

```java
 * of a factory of AOP proxies. This configuration includes the
 * Interceptors and other advice, Advisors, and the proxied interfaces.
 *
 * <p>Any AOP proxy obtained from Spring can be cast to this interface to
 * allow manipulation of its AOP advice.
 *
 * @author Rod Johnson
 * Hoeller
 * @since 13.03.2003
 * @see org.springframework.aop.framework.AdvisedSupport
 */
public interface Advised extends TargetClassAware {

    /**
     * Return whether the Advised configuration is frozen,
     * in which case no advice changes can be made.
     */
    boolean isFrozen();

    /**
     * Are we proxying the full target class instead of specified interfaces
     */
    boolean isProxyTargetClass();

    /**
```

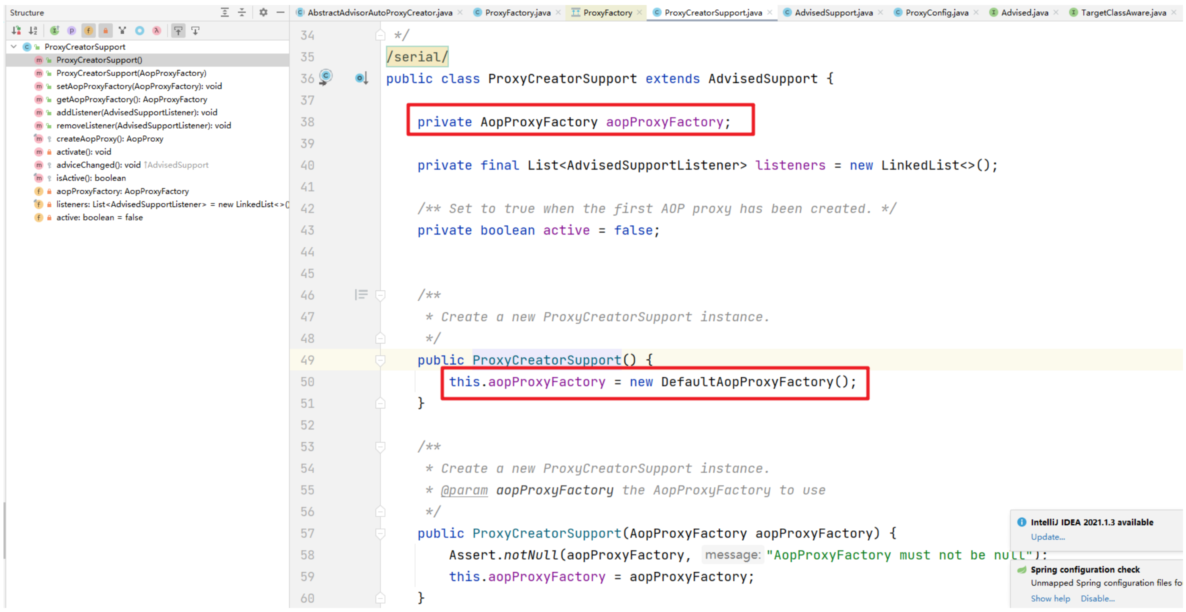
是否冻结

AdvisedSupport

- AOP代理配置管理器的基类。 此类的子类通常是工厂，从中可以直接获取 AOP 代理实例。此类可释放Advices和Advisor的内部管理子类，但实际上并没有实现代理创建方法，实现由子类提供
- AdvisedSupport实现了Advised中处理Advisor和Advice的方法，添加Advice时会被包装成一个Advisor，默认使用的Advisor是DefaultPointcutAdvisor，DefaultPointcutAdvisor默认的Pointcut是TruePointcut（转换为一个匹配所有方法调用的Advisor与代理对象绑定）。
- AdvisedSupport同时会缓存对于某一个方法对应的所有Advisor（Map<MethodCacheKey, List<Object>> methodCache），当Advice或Advisor发生变化时,会清空该缓存。getInterceptorsAndDynamicInterceptionAdvice用来获取对应代理方法对应有效的拦截器链 。

ProxyCreatorSupport

继承了AdvisedSupport,ProxyCreatorSupport正是实现代理的创建方法，ProxyCreatorSupport有一个成员变量AopProxyFactory，而该变量的值默认是DefaultAopProxyFactory



```
ProxyCreatorSupport
    ProxyCreatorSupport()
    ProxyCreatorSupport(AopProxyFactory)
    setAopProxyFactory(AopProxyFactory): void
    getAopProxyFactory(): AopProxyFactory
    addListener(AdvisedSupportListener): void
    removeListener(AdvisedSupportListener): void
    createAopProxy(): AopProxy
    activate(): void
    adviceChanged(): void ↑AdvisedSupport
    isActive(): boolean
    aopProxyFactory: AopProxyFactory
    listeners: List<AdvisedSupportListener> = new LinkedList<>(
    active: boolean = false
```

```java
 */
/serial/
public class ProxyCreatorSupport extends AdvisedSupport {

    private AopProxyFactory aopProxyFactory;

    private final List<AdvisedSupportListener> listeners = new LinkedList<>();

    /** Set to true when the first AOP proxy has been created. */
    private boolean active = false;


    /**
     * Create a new ProxyCreatorSupport instance.
     */
    public ProxyCreatorSupport() {
        this.aopProxyFactory = new DefaultAopProxyFactory();
    }

    /**
     * Create a new ProxyCreatorSupport instance.
     * @param aopProxyFactory the AopProxyFactory to use
     */
    public ProxyCreatorSupport(AopProxyFactory aopProxyFactory) {
        Assert.notNull(aopProxyFactory,  message: "AopProxyFactory must not be null");
        this.aopProxyFactory = aopProxyFactory;
    }
```

IntelliJ IDEA 2021.1.3 available
Update...

Spring configuration check
Unmapped Spring configuration files fo
Show help   Disable...

这个也就和前面的AopProxyFactory串联起来了。

## 3. 多个切面的责任链实现