Spring注解驱动开发3 (AOP篇)

第一模块AOP基础

介绍5大通知类型

1. 前置通知（@Befor）：目标方法执行之前
2. 后置通知（@After）：目标方法执行之后
3. 异常通知 （@AfterThrowing）：目标方法执行之后抛出异常时执行
4. 最终通知（@AfterReturining）：目标方法执行之后,最后执行的通知

以上通知记录程序执行状态

5. 环绕通知 (@Around): 目标方法执行之前之后都要执行，环绕通知能控制目标方法执行

切点的注解（@Pointcut）其中切点表达式有多种语法，最常用的是@AspectJ风格，比如

@Pointcut("execution(* springAnnotationAndSourceTest.aop.MyCalculate.*(..))")

让我们看个实例来了解下如何用注解开发AOP

```
//定义一个切面类

/**
 * 定义一个切面类
 * 使用@Aspect注解声明这个类为一个切面，严格来说，其不属于一种Advice，该注解主要用在类声明上，指明当前类　　是一个组织了切面逻辑的类，并且该注解中可以指定当前类是何种实例化方式，主要有三种：singleton、perthis　　和pertarget
 */
@Aspect
public class CalculateAop {

    /**
     * 定义一个切点，表明这个切面类的通知方法会作用于springAnnotationAndSourceTest包下的所有方法
     */
    @Pointcut("execution(* springAnnotationAndSourceTest.aop.MyCalculate.*(..))")
    public void ponitcut(){}

    /**
     * 定义前置通知，在方法执行前执行
     * @param joinPoint
     */
    @Before("ponitcut()")
    public void beforeAdvice(JoinPoint joinPoint){
        //获取传入的参数
        Object[] args = joinPoint.getArgs();
        for(int i=0;i<args.length;i++){
            if(args[i] instanceof  Integer){
                System.out.println("拦截到"+args[i]+"类型为Integr");
            }else if (args[i] instanceof Double){
                System.out.println("拦截到"+args[i]+"类型为Double");
            }
        }
```

```
 System.out.println(joinPoint.getSignature().getName()+"...beforeAdvice...arg="+
Arrays.asList(args));
    }

    /**
     * 方法结束时调用，不管是不是抛出异常
     * @param joinPoint
     */
    @After("ponitcut()")
    public void afterAdvice(JoinPoint joinPoint){
        System.out.println(joinPoint.getSignature().getName()+"...afterAdvice");
    }

    /**
     * 方法结束且返回值的时候调用
     * 其中： @AfterReturning 的 returning 参数代表方法的返回值，赋值给res参数
     *一定要注意JoinPoint必须要声明在参数列表的第一位，否则spring不认识
     */
    @AfterReturning(value = "ponitcut()",returning = "res")
    public  void afterReturningAdvice(JoinPoint joinPoint , Object res){

 System.out.println(joinPoint.getSignature().getName()+"...afterReturning"+"....
res="+res);
    }

    /**
     * 方法发现异常，则会执行这个通知
     * throwing代表抛出的异常信息会被参数e接收
     */
    @AfterThrowing(value = "ponitcut()",throwing = "e")
    public void afterThrowing(JoinPoint joinPoint,Exception e)
    {
        System.out.println(joinPoint.getSignature().getName()+"....afterThrowing
抛出异常:"+e);
    }


}
```

上面我们定义了个切面并定义了通知的结构，以及切点表达式（这些通知会作用在哪些包的哪些类的哪些方法上）

接下来我们需要开启注解AOP模式，我们需要写一个配置类

```
/**
 * @EnableAspectJAutoProxy 开启基于注解的aop模式，必须要有，否则aop无法正常运行
 */
@EnableAspectJAutoProxy
@Configuration
public class ConfigForAop {

    @Bean
    public  MyCalculate myCalculate(){
       return new MyCalculate();
```

```
    }

    @Bean
     public  CalculateAop calculateAop(){
         return  new CalculateAop();
    }
}
```

其中MyCalculate的定义如下

```
/**
 * 计算器类
 */
public class MyCalculate {

    public double add(int i,int k){
        return i+k;
    }

    public double div(double i,double k) throws DivException {
        //检测分母是否为0，如果为0则抛出异常
        if(k==0.0)throw  new DivException("分母不得为0");
        return i/k;
    }
}


//异常类定义
public class DivException extends Exception {
    public DivException(String message) {
        super(message);
    }
}
```

完成了上面的定义，我们的目标是将MyCalculate的所有方法通过aop进行代理,现在测试一下

```
    @Test
    public void testConfigAop() throws DivException {
        //创建ioc容器，注意这里不要传参数！有参构造的话会直接执行到
applicationContext.refresh();之后在调用
applicationContext.getEnvironment().setActiveProfiles("test,dev");会不管用!
        AnnotationConfigApplicationContext applicationContext = new
AnnotationConfigApplicationContext(ConfigForAop.class);
        MyCalculate myCalculate=applicationContext.getBean(MyCalculate.class);
        myCalculate.add(1,2);
        //这里传进分母为0，意图在于验证@AfterThrowing注解是否起作用
        myCalculate.div(1.1,0.0);
    }
```

得到的结果截图：

可以发现aop代理起作用了！注意@After注解在方法的处理逻辑完成后才调用，而@AfterReturning在方法有返回值的时候调用。

第二模块 @EnableAspectJAutoProxy的底层原理

研究的意义： 我们必须要在配置类上标注@EnableAspectJAutoProxy注解才能够使注册进来的切面工作。否则是不会工作的。 故此我们来探索下究竟为什么以及@EnableAspectJAutoProxy注解究竟配置了什么才能使切面的通知组件都生效。

源码研究

找到Aop的配置类（为了方便依旧使用第一模块的配置类）

```
/**
 * @EnableAspectJAutoProxy 开启基于注解的aop模式，必须要有，否则aop无法正常运行
 */
@EnableAspectJAutoProxy
@Configuration
public class ConfigForAop {

    @Bean
    public  MyCalculate myCalculate(){
       return new MyCalculate();
   }

   @Bean
    public  CalculateAop calculateAop(){
        return  new CalculateAop();
   }
}
```

我们直接点进@EnableAspectJAutoProxy注解内部

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
//这里内部又引入了AspectJAutoProxyRegistrar这个bean
@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 {@code false}.
     */
```

```
    boolean proxyTargetClass() default false;

    /**
     * Indicate that the proxy should be exposed by the AOP framework as a
{@code ThreadLocal}
     * for retrieval via the {@link
org.springframework.aop.framework.AopContext} class.
     * Off by default, i.e. no guarantees that {@code AopContext} access will
work.
     * @since 4.3.1
     */
    boolean exposeProxy() default false;

}
```

我们将重点看到@Import(AspectJAutoProxyRegistrar.class)，导入了AspectJAutoProxyRegistrar这个bean，那它究竟做什么的呢，继续点进去

```
class AspectJAutoProxyRegistrar implements ImportBeanDefinitionRegistrar {

    /**
     * Register, escalate, and configure the AspectJ auto proxy creator based on
the value
     * of the @{@link EnableAspectJAutoProxy#proxyTargetClass()} attribute on
the importing
     * {@code @Configuration} class.
       简单的讲，这里就是配置一些导入bean的定义
     */
    @Override
    public void registerBeanDefinitions(
            AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry
registry) {
        //重点

AopConfigUtils.registerAspectJAnnotationAutoProxyCreatorIfNecessary(registry);

        AnnotationAttributes enableAspectJAutoProxy =
                AnnotationConfigUtils.attributesFor(importingClassMetadata,
EnableAspectJAutoProxy.class);
        //检测@EnableAspectJAutoProxy是否自定义了一些目标代理类等
        if (enableAspectJAutoProxy.getBoolean("proxyTargetClass")) {
            AopConfigUtils.forceAutoProxyCreatorToUseClassProxying(registry);
        }
        if (enableAspectJAutoProxy.getBoolean("exposeProxy")) {
            AopConfigUtils.forceAutoProxyCreatorToExposeProxy(registry);
        }
    }

}
```

继续看到这个类它实现了**ImportBeanDefinitionRegistrar** 接口，是不是很熟悉，在@Import标签内部曾经我们使用过，这个接口的方法允许你直接像spring容器注册进一些bean，通过registry注册。看到**registerBeanDefinitions** 这个方法内部，AopConfigUtils.registerAspectJAnnotationAutoProxyCreatorIfNecessary(registry); 这里大概含义就是给spring容器注册进一个AspectJAnnotationAutoProxyCreator ，究竟长什么样呢，点进去继续看。

```
public static BeanDefinition
registerAspectJAnnotationAutoProxyCreatorIfNecessary(BeanDefinitionRegistry
registry) {    return
registerAspectJAnnotationAutoProxyCreatorIfNecessary(registry, (Object)null);}
```

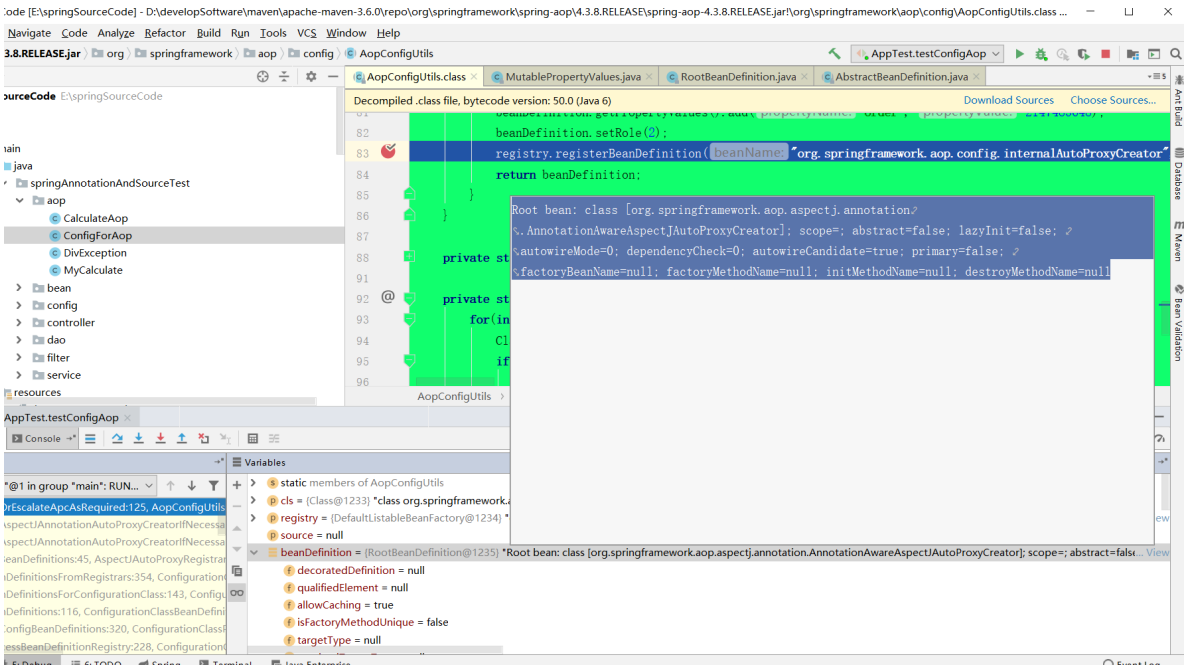
ok，到这里，已经知道大致的意思了，这里就是进行注册AspectJAnnotationAutoProxyCreator，如果你觉得还要要更清楚点，我们继续进入到最底层

```
 private static BeanDefinition registerOrEscalateApcAsRequired(Class<?> cls,
BeanDefinitionRegistry registry, Object source) {
        Assert.notNull(registry, "BeanDefinitionRegistry must not be null");
        if
(registry.containsBeanDefinition("org.springframework.aop.config.internalAutoPro
xyCreator")) {
            BeanDefinition apcDefinition =
registry.getBeanDefinition("org.springframework.aop.config.internalAutoProxyCrea
tor");
            if (!cls.getName().equals(apcDefinition.getBeanClassName())) {
                int currentPriority =
findPriorityForClass(apcDefinition.getBeanClassName());
                int requiredPriority = findPriorityForClass(cls);
                if (currentPriority < requiredPriority) {
                    apcDefinition.setBeanClassName(cls.getName());
                }
            }

            return null;
        } else {
            RootBeanDefinition beanDefinition = new RootBeanDefinition(cls);
            beanDefinition.setSource(source);
            beanDefinition.getPropertyValues().add("order", -2147483648);
            beanDefinition.setRole(2);

 registry.registerBeanDefinition("org.springframework.aop.config.internalAutoPro
xyCreator", beanDefinition);
            return beanDefinition;
        }
    }
```

你可以在这个方法内部打个断点，我们观测下registry到底注册了什么东西



看到beanDefinition的class，最终像容器种注册了AnnotationAwareAspectJAutoProxyCreator，这个类到底起到了什么作用呢，我们带着疑问搜索这个类（这里重点不是类的内容，而是它的继承关系），这里列出这个类的继承关系。

```
AnnotationAwareAspectJAutoProxyCreator
   -->AspectJAwareAdvisorAutoProxyCreator
      -->AbstractAdvisorAutoProxyCreator
         -->AbstractAutoProxyCreator extends ProxyProcessorSupport
            implements SmartInstantiationAwareBeanPostProcessor,
BeanFactoryAware
```

好，到此位置，我们看到关键的一个类AbstractAutoProxyCreator 的实现，它实现了SmartInstantiationAwareBeanPostProcessor和 BeanFactoryAware， BeanFactoryAware可以获取spring底层的BeanFactory。而最种要的接口在于**SmartInstantiationAwareBeanPostProcessor**看到它的形式是xxxBeanPostProcessor,在bean的初始化前做了些操作。点进去

```
public interface SmartInstantiationAwareBeanPostProcessor extends
InstantiationAwareBeanPostProcessor
//再次点进InstantiationAwareBeanPostProcessor
public interface InstantiationAwareBeanPostProcessor extends BeanPostProcessor
```

最终的结果说明SmartInstantiationAwareBeanPostProcessor是一个BeanPostProcessor!

现在呢，我们知道了AnnotationAwareAspectJAutoProxyCreator的继承关系，我们从最底层的AbstractAutoProxyCreator向上研究。

1. 首先AbstractAutoProxyCreator内部的方法postProcessBeforeInstantiation(Class<?> beanClass, String beanName)有具体的beanPostProcessor的执行逻辑，我们暂且不管这些什么含义。
2. 继续的我们向上一层到AbstractAdvisorAutoProxyCreator，可以看到这一层并没有BeanPostProcessor的处理逻辑，但是重写了setBeanFactory的一些逻辑，并且setBeanFactory内部调用了initBeanFactory。
3. 我们接着看向AspectJAwareAdvisorAutoProxyCreator ，发现其中并没有像要的（BeanPostProcessor的处理逻辑）
4. 然后我们到最顶层AnnotationAwareAspectJAutoProxyCreator,这个类内部呢有一个initBeanFactory初始化bean工厂的方法，上面**AbstractAdvisorAutoProxyCreator的**

setBeanFactory方法也调用了initBeanFactory方法。由于AbstractAdvisorAutoProxyCreator又是AnnotationAwareAspectJAutoProxyCreator的父类，故此AnnotationAwareAspectJAutoProxyCreator种的initBeanFactory方法会被**AbstractAdvisorAutoProxyCreator的setBeanFactory方法**调用

在这些步骤清楚了之后，我们看下AnnotationAwareAspectJAutoProxyCreator这个类是如何被注册到spring容器种的。

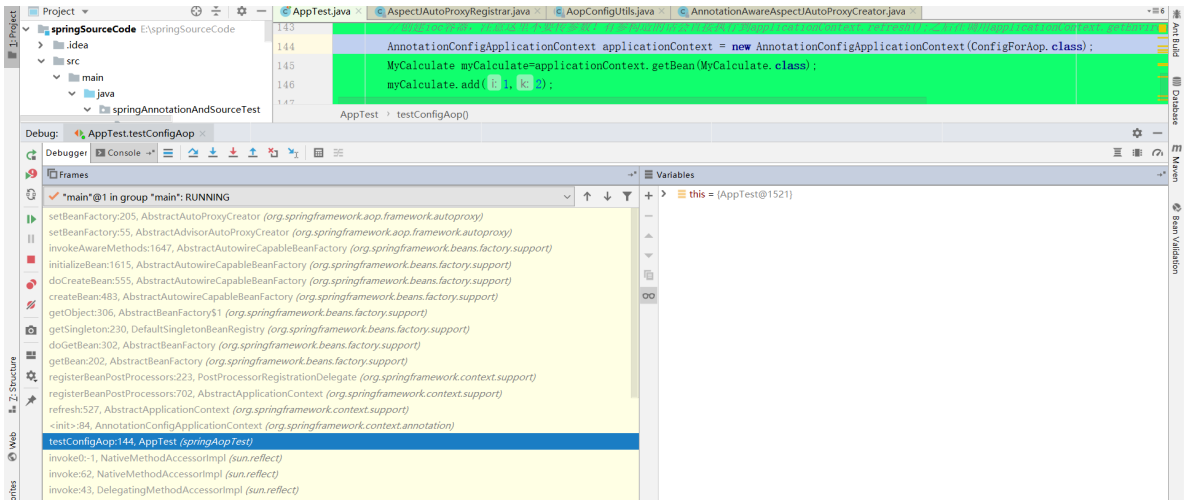
在开始之前，我们首先要打几个断点，通过上面的分析可只AnnotationAwareAspectJAutoProxyCreator是一个实现了BeanPostProcessor和Aware接口的类，故此我们要研究如何注入进spring的，就必须要在Aware接口和BeanPostProcessor接口的主要方法处打上断点(因为spring在注册bean的时候会专门加载类是否实现BeanPostProcessor和xxxAware接口，再进行装配)。

要找这两个接口的方法入口，就定位到最底层的类AbstractAutoProxyCreator中，我们在其内部的setBeanFactory方法上打上断点（实现BeanFactoryAware中的方法）。 其次我们查找BeanPostProcessor的实现方法，也在这个类中找到postProcessBeforeInstantiation(Class<?> beanClass, String beanName)和 postProcessAfterInitialization(Object bean, String beanName)打上断点(这些事实现BeanPostProcessor的方法)

同样我们需要在配置类上打断点，如下图

```
3  import org.springframework.context.annotation.*;
4
5
6  /**
7   * @EnableAspectJAutoProxy 开启基于注解的aop模式，必须要有，否则aop无法正常运行
8   */
9  @EnableAspectJAutoProxy
10 @Configuration
11 public class ConfigForAop {
12
13     @Bean
14     public MyCalculate myCalculate() { return new MyCalculate(); }
17
18     @Bean
19     public CalculateAop calculateAop() { return new CalculateAop(); }
22 }
23
24
```

然后进入Debug模式，首先观测到我们的方法调用栈



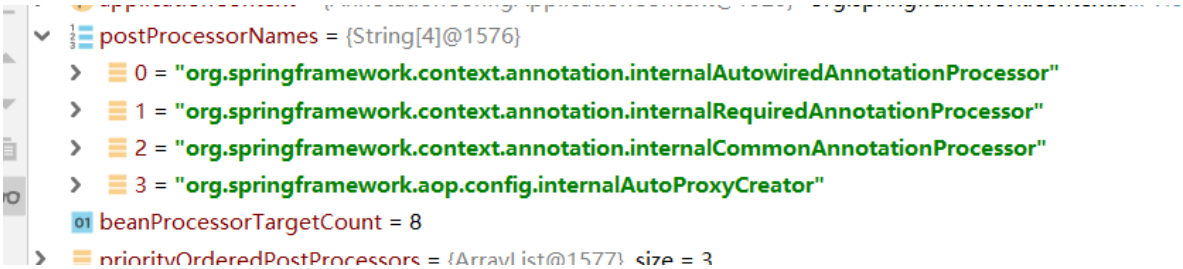
在蓝色标记处往上一次研究

注册AnnotationAwareAspectJAutoProxyCreator类的流程

1. 调用AnnotationConfigApplicationContext(Class<?>... annotatedClasses)并传入配置类，创建ioc容器
2. 注册配置类AnnotationConfigApplicationContext(Class<?>... annotatedClasses)调用refresh（）方法开始创建bean加载各种组件
3. `registerBeanPostProcessors(ConfigurableListableBeanFactory beanFactory)`

   专门用来注册bean的BeanPostProcessor来方便拦截bean的创建

3.1 我们点进去registerBeanPostProcessors这个方法内部，看到beanFactory.getBeanNamesForType(BeanPostProcessor.class, true, false);这一条语句，它主要获取所有已经定义的后置处理器，我们可以看下到底取出了哪些。



其中【internalAutoProxyCreator】是注册AnnotationAwareAspectJAutoProxyCreator的时候的名字（还没有创建对象，知识bean的类定义）

3.2 在registerBeanPostProcessors这个方法内部往下看，有beanFactory.addBeanPostProcessor(new BeanPostProcessorChecker(beanFactory, beanProcessorTargetCount)); 这里还会导入一些spring自己的beanPostProcessor

3.3 看向下面的代码，这部分主要是对实现了PriorityOrdered和Ordered还有没有实现前面两个接口的BeanPostProcessors进行分批次初始化。首先创建实现PriorityOrdered的BeanPostProcessor，其次是Ordered最后是创建没有实现上面两者的BeanPostProcessor

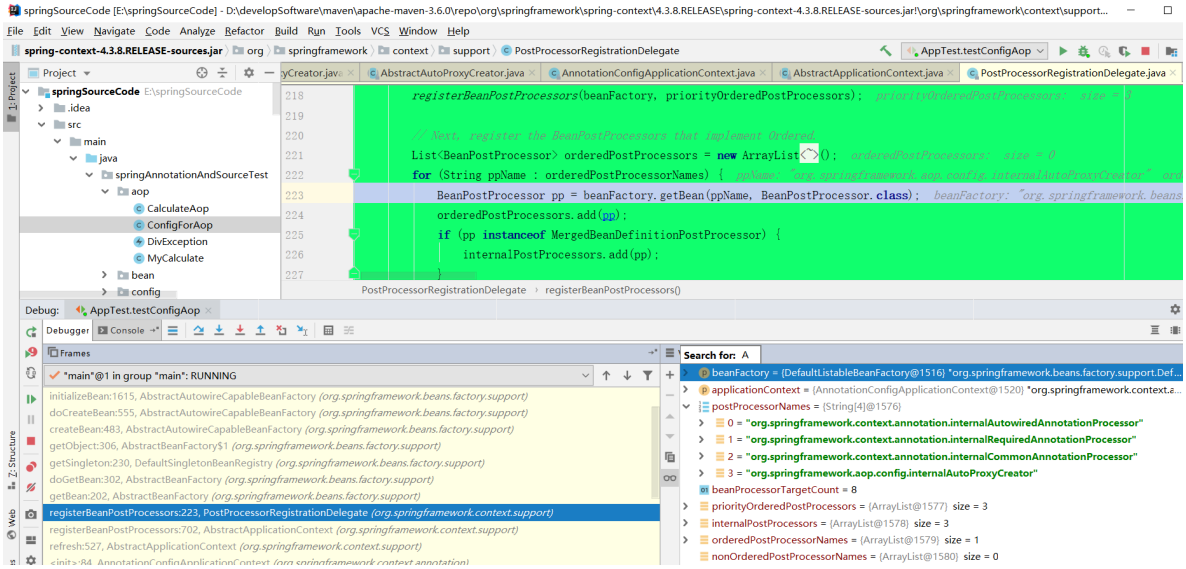
```
        // Separate between BeanPostProcessors that implement PriorityOrdered,
        // Ordered, and the rest.
        List<BeanPostProcessor> priorityOrderedPostProcessors = new
ArrayList<BeanPostProcessor>();
        List<BeanPostProcessor> internalPostProcessors = new
ArrayList<BeanPostProcessor>();
        List<String> orderedPostProcessorNames = new ArrayList<String>();
        List<String> nonOrderedPostProcessorNames = new ArrayList<String>();
        for (String ppName : postProcessorNames) {
            if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
                BeanPostProcessor pp = beanFactory.getBean(ppName,
BeanPostProcessor.class);
                priorityOrderedPostProcessors.add(pp);
                if (pp instanceof MergedBeanDefinitionPostProcessor) {
                    internalPostProcessors.add(pp);
                }
            }
            else if (beanFactory.isTypeMatch(ppName, Ordered.class)) {
                orderedPostProcessorNames.add(ppName);
            }
            else {
```

```
                nonOrderedPostProcessorNames.add(ppName);
            }
        }

        // First, register the BeanPostProcessors that implement
PriorityOrdered.
        sortPostProcessors(beanFactory, priorityOrderedPostProcessors);
        registerBeanPostProcessors(beanFactory, priorityOrderedPostProcessors);

        // Next, register the BeanPostProcessors that implement Ordered.
        List<BeanPostProcessor> orderedPostProcessors = new
ArrayList<BeanPostProcessor>();
        for (String ppName : orderedPostProcessorNames) {
            BeanPostProcessor pp = beanFactory.getBean(ppName,
BeanPostProcessor.class);
            orderedPostProcessors.add(pp);
            if (pp instanceof MergedBeanDefinitionPostProcessor) {
                internalPostProcessors.add(pp);
            }
        }
        sortPostProcessors(beanFactory, orderedPostProcessors);
        registerBeanPostProcessors(beanFactory, orderedPostProcessors);
```

看到debug其中一步



在orderedPostProcessors链表中加入了我们的internalAutoProxyCreator（也就是AnnotationAwareAspectJAutoProxyCreator因为它的父类实现了Ordered接口），这一步会去拿到internalAutoProxyCreator对应的AnnotationAwareAspectJAutoProxyCreator，主要又分为3个小步骤

3.4 .1 创建目标bean（doCreateBean）

3.4.2 populateBean 给bean赋值

3.4.3 initializeBean 初始化bean ,其中初始化bean又分为几步去初始化

```
//initializeBean的源码
protected Object initializeBean(final String beanName, final Object bean,
RootBeanDefinition mbd) {
        if (System.getSecurityManager() != null) {
            AccessController.doPrivileged(new PrivilegedAction<Object>() {
                @Override
                public Object run() {
```

```
                    invokeAwareMethods(beanName, bean);
                    return null;
                }
            }, getAccessControlContext());
        }
        else {
              //3.4.3.1
            invokeAwareMethods(beanName, bean);
        }

        Object wrappedBean = bean;
        if (mbd == null || !mbd.isSynthetic()) {
              //3.4.3.2
            wrappedBean =
applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);
        }

        try {
           //3.4.3.3
            invokeInitMethods(beanName, wrappedBean, mbd);
        }
        catch (Throwable ex) {
            throw new BeanCreationException(
                    (mbd != null ? mbd.getResourceDescription() : null),
                    beanName, "Invocation of init method failed", ex);
        }

        if (mbd == null || !mbd.isSynthetic()) {
            //3.4.3.4
            wrappedBean =
applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);
        }
        return wrappedBean;
    }
```

3.4.3.1 invokeAwareMethods 主要是查看当前的bean是否是以下3中Aware的拓展接口实现类，如果是就做相应的赋值

```
private void invokeAwareMethods(final String beanName, final Object bean) {
        if (bean instanceof Aware) {
            if (bean instanceof BeanNameAware) {
                ((BeanNameAware) bean).setBeanName(beanName);
            }
            if (bean instanceof BeanClassLoaderAware) {
                ((BeanClassLoaderAware)
bean).setBeanClassLoader(getBeanClassLoader());
            }
            if (bean instanceof BeanFactoryAware) {
                ((BeanFactoryAware)
bean).setBeanFactory(AbstractAutowireCapableBeanFactory.this);
            }
        }
    }
```

3.4.3.2 执行applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName)执行bean的BeanPostProcessor在初始化方法前执行的逻辑，我们看看其中的方法逻辑

```
@Override

    public Object applyBeanPostProcessorsBeforeInitialization(Object
existingBean, String beanName)
            throws BeansException {

        Object result = existingBean;
        //获取所有beanPostProcessor
        for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
            //调用其中的beanPostProcessor在初始化方法前执行的逻辑
            result = beanProcessor.postProcessBeforeInitialization(result,
beanName);
            if (result == null) {
                return result;
            }
        }
        return result;
    }
```

3.4.3.3 invokeInitMethods(beanName, wrappedBean, mbd); 执行自定义的初始化方法

3.4.3.4 applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);同3.4.3.2中的代码，只不过执行的是BeanPostProcessor在初始化方法**后**执行的逻辑

这一步完成后，bean基本就初始化完成了，随后一路放行，spring利用beanFactory.addBeanPostProcessor(postProcessor)给bean添加BeanPostProcessor

以上是创建和注册AnnotationAwareAspectJAutoProxyCreator(实质就是BeanPostProcessor和Aware接口实现类) 的过程

在注册完成BeanPostProcessor后，也就是 registerBeanPostProcessors(beanFactory);方法执行完后，spring又调用用了finishBeanFactoryInitialization(beanFactory);完成beanFactory的创建工作，创建剩下的单实例bean

```
//refresh() 的步骤
@Override
    public void refresh() throws BeansException, IllegalStateException {
        synchronized (this.startupShutdownMonitor) {
            // Prepare this context for refreshing.
            prepareRefresh();

            // Tell the subclass to refresh the internal bean factory.
            ConfigurableListableBeanFactory beanFactory =
obtainFreshBeanFactory();

            // Prepare the bean factory for use in this context.
            prepareBeanFactory(beanFactory);

            try {
                // Allows post-processing of the bean factory in context
subclasses.
                postProcessBeanFactory(beanFactory);
```

```
                // Invoke factory processors registered as beans in the context.
                invokeBeanFactoryPostProcessors(beanFactory);

                // Register bean processors that intercept bean creation.
                registerBeanPostProcessors(beanFactory);

                // Initialize message source for this context.
                initMessageSource();

                // Initialize event multicaster for this context.
                initApplicationEventMulticaster();

                // Initialize other special beans in specific context
subclasses.
                onRefresh();

                // Check for listener beans and register them.
                registerListeners();

                // 初始化剩下的单实例bean
                finishBeanFactoryInitialization(beanFactory);

                // Last step: publish corresponding event.
                finishRefresh();
            }

            catch (BeansException ex) {
                if (logger.isWarnEnabled()) {
                    logger.warn("Exception encountered during context
initialization - " +
                            "cancelling refresh attempt: " + ex);
                }

                // Destroy already created singletons to avoid dangling
resources.
                destroyBeans();

                // Reset 'active' flag.
                cancelRefresh(ex);

                // Propagate exception to caller.
                throw ex;
            }

            finally {
                // Reset common introspection caches in Spring's core, since we
                // might not ever need metadata for singleton beans anymore...
                resetCommonCaches();
            }
        }
    }
```

在finishBeanFactoryInitialization的内部，又继续调用了preInstantiateSingletons( );

我们点进去继续看其方法内部实现(节选)

```
@Override
    public void preInstantiateSingletons() throws BeansException {
        if (this.logger.isDebugEnabled()) {
            this.logger.debug("Pre-instantiating singletons in " + this);
        }

        //遍历当前加载到的bean名称
        List<String> beanNames = new ArrayList<String>
(this.beanDefinitionNames);

        //遍历所有bean名称（非懒加载）
        for (String beanName : beanNames) {

            RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);
            //校验是否是单例且不是抽象且不是懒加载的
            if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
                //检测bean是不是实现了FactoryBean接口
                if (isFactoryBean(beanName)) {
                    final FactoryBean<?> factory = (FactoryBean<?>)
getBean(FACTORY_BEAN_PREFIX + beanName);
                    boolean isEagerInit;
                    if (System.getSecurityManager() != null && factory
instanceof SmartFactoryBean) {
                        isEagerInit = AccessController.doPrivileged(new
PrivilegedAction<Boolean>() {
                            @Override
                            public Boolean run() {
                                return ((SmartFactoryBean<?>)
factory).isEagerInit();
                            }
                        }, getAccessControlContext());
                    }
                    else {
                        isEagerInit = (factory instanceof SmartFactoryBean &&
                                ((SmartFactoryBean<?>) factory).isEagerInit());
                    }
                    if (isEagerInit) {
                        getBean(beanName);
                    }
                }
                //如果不是FactoryBean
                else {
                    //直接调用getBean方法，把名字传入
                    getBean(beanName);
                }
            }

        }
        。。。。
    }
```

继续，看到上面代码的getBean（beanName）处，我们点进去

```
@Override
    public Object getBean(String name) throws BeansException {
        return doGetBean(name, null, null, false);
    }
```

再点进doGetBean(name, null, null, false)方法

```
protected <T> T doGetBean(
            final String name, final Class<T> requiredType, final Object[] args,
boolean typeCheckOnly)
            throws BeansException {

        final String beanName = transformedBeanName(name);
        Object bean;

        // Eagerly check singleton cache for manually registered singletons.
        //这一步查看是否缓存中早已存在beanName对应的bean，如果存在则取出
        Object sharedInstance = getSingleton(beanName);
        if (sharedInstance != null && args == null) {
            if (logger.isDebugEnabled()) {
                if (isSingletonCurrentlyInCreation(beanName)) {
                    logger.debug("Returning eagerly cached instance of singleton
bean '" + beanName +
                                 "' that is not fully initialized yet - a consequence
of a circular reference");
                }
                else {
                    logger.debug("Returning cached instance of singleton bean '"
+ beanName + "'");
                }
            }
            bean = getObjectForBeanInstance(sharedInstance, name, beanName,
null);
        }

        else {
            // Fail if we're already creating this bean instance:
            // We're assumably within a circular reference.
            if (isPrototypeCurrentlyInCreation(beanName)) {
                throw new BeanCurrentlyInCreationException(beanName);
            }

            // Check if bean definition exists in this factory.

            BeanFactory parentBeanFactory = getParentBeanFactory();
            if (parentBeanFactory != null && !containsBeanDefinition(beanName))
{
                // Not found -> check parent.
                String nameToLookup = originalBeanName(name);
                if (args != null) {
                    // Delegation to parent with explicit args.
                    return (T) parentBeanFactory.getBean(nameToLookup, args);
                }
                else {
                    // No args -> delegate to standard getBean method.
```

```
                    return parentBeanFactory.getBean(nameToLookup,
requiredType);
                }
            }

            if (!typeCheckOnly) {
                markBeanAsCreated(beanName);
            }

            try {
                final RootBeanDefinition mbd =
getMergedLocalBeanDefinition(beanName);
                checkMergedBeanDefinition(mbd, beanName, args);

                // Guarantee initialization of beans that the current bean
depends on.
                String[] dependsOn = mbd.getDependsOn();
                if (dependsOn != null) {
                    for (String dep : dependsOn) {
                        if (isDependent(beanName, dep)) {
                            throw new
BeanCreationException(mbd.getResourceDescription(), beanName,
                                    "Circular depends-on relationship between '"
+ beanName + "' and '" + dep + "'");
                        }
                        registerDependentBean(dep, beanName);
                        getBean(dep);
                    }
                }

                // Create bean instance.
                if (mbd.isSingleton()) {
                    //关注点
                    sharedInstance = getSingleton(beanName, new
ObjectFactory<Object>() {
                        @Override
                        public Object getObject() throws BeansException {
                            try {
                                return createBean(beanName, mbd, args);
                            }
                            catch (BeansException ex) {
                                // Explicitly remove instance from singleton
cache: It might have been put there
                                // eagerly by the creation process, to allow for
circular reference resolution.
                                // Also remove any beans that received a
temporary reference to the bean.
                                destroySingleton(beanName);
                                throw ex;
                            }
                        }
                    });
                    bean = getObjectForBeanInstance(sharedInstance, name,
beanName, mbd);
                }

                else if (mbd.isPrototype()) {
                    // It's a prototype -> create a new instance.
```

```
                    Object prototypeInstance = null;
                    try {
                        beforePrototypeCreation(beanName);
                        prototypeInstance = createBean(beanName, mbd, args);
                    }
                    finally {
                        afterPrototypeCreation(beanName);
                    }
                    bean = getObjectForBeanInstance(prototypeInstance, name,
beanName, mbd);
                }

                else {
                    String scopeName = mbd.getScope();
                    final Scope scope = this.scopes.get(scopeName);
                    if (scope == null) {
                        throw new IllegalStateException("No Scope registered for
scope name '" + scopeName + "'");
                    }
                    try {
                        Object scopedInstance = scope.get(beanName, new
ObjectFactory<Object>() {
                            @Override
                            public Object getObject() throws BeansException {
                                beforePrototypeCreation(beanName);
                                try {
                                    return createBean(beanName, mbd, args);
                                }
                                finally {
                                    afterPrototypeCreation(beanName);
                                }
                            }
                        });
                        bean = getObjectForBeanInstance(scopedInstance, name,
beanName, mbd);
                    }
                    catch (IllegalStateException ex) {
                        throw new BeanCreationException(beanName,
                                "Scope '" + scopeName + "' is not active for the
current thread; consider " +
                                "defining a scoped proxy for this bean if you
intend to refer to it from a singleton",
                                ex);
                    }
                }
            }
            catch (BeansException ex) {
                cleanupAfterBeanCreationFailure(beanName);
                throw ex;
            }
        }

        // Check if required type matches the type of the actual bean instance.
        if (requiredType != null && bean != null &&
!requiredType.isAssignableFrom(bean.getClass())) {
            try {
                return getTypeConverter().convertIfNecessary(bean,
requiredType);
```

```
            }
            catch (TypeMismatchException ex) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Failed to convert bean '" + name + "' to
required type '" +
                            ClassUtils.getQualifiedName(requiredType) + "'",
ex);
                }
                throw new BeanNotOfRequiredTypeException(name, requiredType,
bean.getClass());
            }
        }
        return (T) bean;
    }
```

其中sharedInstance = getSingleton这一步之前，spring尝试去获取bean，如果实在获取不了，才会调用这一步去创建bean。**spring创建的bean都会被缓存起来**。

然后getSingleton 又调用createBean来创建bean（这里才是创建bean，前面都是获取bean），继续点进createBean方法

```
    @Override
    protected Object createBean(String beanName, RootBeanDefinition mbd,
Object[] args) throws BeanCreationException {
        if (logger.isDebugEnabled()) {
            logger.debug("Creating instance of bean '" + beanName + "'");
        }
        RootBeanDefinition mbdToUse = mbd;

        // Make sure bean class is actually resolved at this point, and
        // clone the bean definition in case of a dynamically resolved Class
        // which cannot be stored in the shared merged bean definition.
        Class<?> resolvedClass = resolveBeanClass(mbd, beanName);
        if (resolvedClass != null && !mbd.hasBeanClass() &&
mbd.getBeanClassName() != null) {
            mbdToUse = new RootBeanDefinition(mbd);
            mbdToUse.setBeanClass(resolvedClass);
        }

        // Prepare method overrides.
        try {
            mbdToUse.prepareMethodOverrides();
        }
        catch (BeanDefinitionValidationException ex) {
            throw new
BeanDefinitionStoreException(mbdToUse.getResourceDescription(),
                    beanName, "Validation of method overrides failed", ex);
        }

        try {
            // Give BeanPostProcessors a chance to return a proxy instead of the
target bean instance.
            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);
        }

        Object beanInstance = doCreateBean(beanName, mbdToUse, args);
        if (logger.isDebugEnabled()) {
            logger.debug("Finished creating instance of bean '" + beanName +
"'");
        }
        return beanInstance;
    }
```

resolveBeforeInstantiation(beanName, mbdToUse); 解析BeforeInstantiation并希望后置处理器能够返回代理对象,如果能返回代理对象则将代理对象返回，如果不能就**Object beanInstance = doCreateBean(beanName, mbdToUse, args);** 执行这条语句，这才是真正的创建bean实例，其中doCreateBean内部就会给bean赋值，调用bean实现的Aware，BeanPostProcessor,自定义的初始化方法去创建和初始化bean。

我们不妨看一下 resolveBeforeInstantiation(beanName, mbdToUse);的内部是如何去通过**后置处理器获取代理对象的**，上源码

```
/**
     * Apply before-instantiation post-processors, resolving whether there is a
     * before-instantiation shortcut for the specified bean.
     * @param beanName the name of the bean
     * @param mbd the bean definition for the bean
     * @return the shortcut-determined bean instance, or {@code null} if none
     主要是调用 before-instantiation的post-processors
     */
    protected Object resolveBeforeInstantiation(String beanName,
RootBeanDefinition mbd) {
        Object bean = null;
        if (!Boolean.FALSE.equals(mbd.beforeInstantiationResolved)) {
            // Make sure bean class is actually resolved at this point.
            if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors())
{
                Class<?> targetType = determineTargetType(beanName, mbd);
                if (targetType != null) {
                    //这里调用BeanPostProcessors的BeforeInstantiation方法尝试获取
bean的代理对象
                    bean =
applyBeanPostProcessorsBeforeInstantiation(targetType, beanName);
                    if (bean != null) {
                        bean = applyBeanPostProcessorsAfterInitialization(bean,
beanName);
                    }
                }
            }
            mbd.beforeInstantiationResolved = (bean != null);
        }
        return bean;
    }
```

我们看向applyBeanPostProcessorsBeforeInstantiation的内部实现

```
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;
    }
```

这里一开始遍历所有的后置处理器，并且检查后置处理器是否是**InstantiationAwareBeanPostProcessor**这个类型的。如果是的话，就调用它的postProcessBeforeInstantiation得到代理对象！为什么单独判断InstantiationAwareBeanPostProcessor呢？ 它和普通的BeanPostProcessor存在者一些不同之处:

区别：【BeanPostProcessor】是在bean对象创建完成，初始化方法调用前后进行调用的。

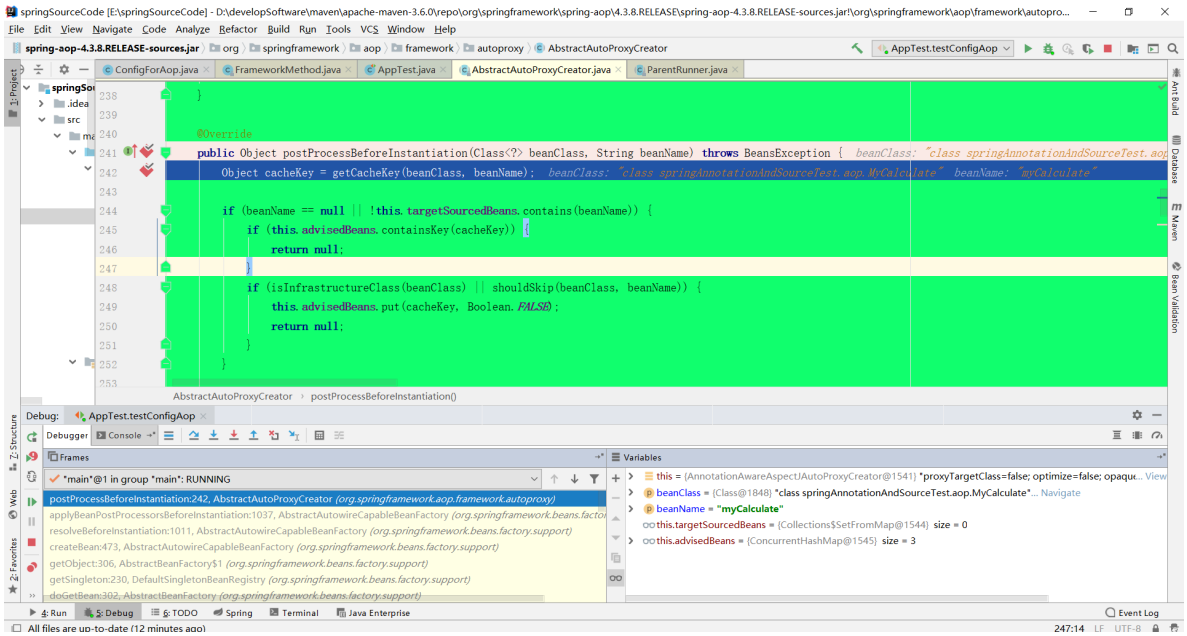
【InstantiationAwareBeanPostProcessor 】**是在创建bean实例之前尝试用后置处理器返回对象的!!**

我们研究的 AnnotationAwareAspectJAutoProxyCreator就是实现了InstantiationAwareBeanPostProcessor这个后置处理器的！所以会在任何bean创建之前会进行拦截，尝试返回bean的实例，会调用**postProcessBeforeInstantiation**方法。

接下来AOP是如何生成代理类的，以及通知方法到底是如何加载到bean的方法中是本文的重点

要了解这些流程，我们就要从上面提到的**postProcessBeforeInstantiation** 这个方法，代理类就是从这个方法内部产生的。

我们之前打了两个断点在AbstractAutoProxyCreator 的postProcessBeforeInstantiation和postProcessAfterInstantiation处。我们需要重新debug，然后我们会发现，ioc容器中的每个bean都会被这两个方法所捕获进行校验，我们只关注切面bean还有我们的Calculate那个bean，故此我们放行与我们关注点无关的bean，当Calculate那个bean被我们捕获的时候，首先他会来到postProcessBeforeInstantiation这个方法，如下图:



接下来我们上 postProcessBeforeInstantiation源码

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

        if (beanName == null || !this.targetSourcedBeans.contains(beanName)) {
            //判断增强bean集合是否包含当前的bean，第一次加载的时候是不包含的
            if (this.advisedBeans.containsKey(cacheKey)) {
                return null;
            }
            //判断是否是基础类型以及是不是可以跳过
            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.
        if (beanName != null) {
            TargetSource targetSource = getCustomTargetSource(beanClass,
beanName);
            if (targetSource != null) {
                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;
    }
```

讲解下上面代码最终执行的步骤：

1. 每一个bean在创建的时候都会来到AnnotationAwareAspectJAutoProxyCreator这个类中的postProcessBeforeInstantiation做处理。
2. 在postProcessBeforeInstantiation内部又有以下判断

   2.1 判断当前bean是否存在于advisedBeans中（保存了所有需要增强的bean），**增强bean的意思是这个bean不是普通的bean，而是需要进行代理的bean（比如方法执行前后做些通知处理）**

   2.2 接下来通过isInfrastructureClass(beanClass)方法判断bean是否是基础类型，我们看下什么是基础类型

```
    @Override
    protected boolean isInfrastructureClass(Class<?> beanClass) {
        return (super.isInfrastructureClass(beanClass) ||
this.aspectJAdvisorFactory.isAspect(beanClass));
    }


//super.isInfrastructureClass的源码
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;
    }

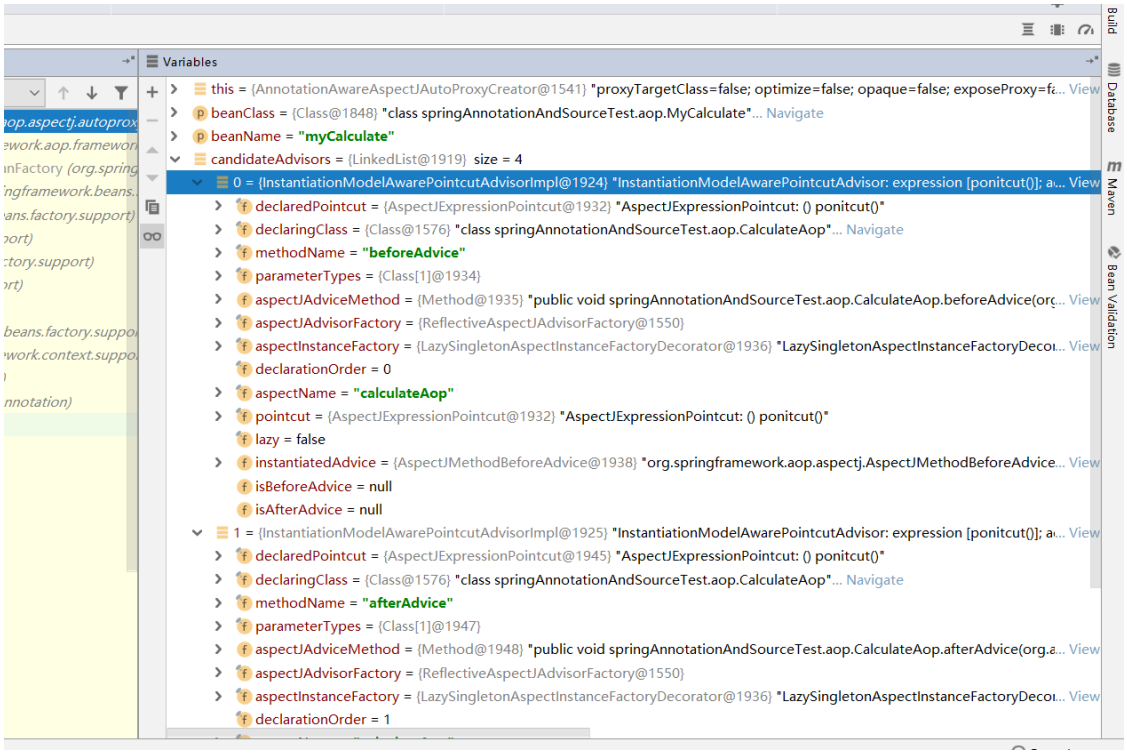

//this.aspectJAdvisorFactory.isAspect的部分实现
@Override
    public boolean isAspect(Class<?> clazz) {
        return (hasAspectAnnotation(clazz) && !compiledByAjc(clazz));
    }

    private boolean hasAspectAnnotation(Class<?> clazz) {
        return (AnnotationUtils.findAnnotation(clazz, Aspect.class) != null);
    }
```

从上面源码来看，基础类型可以理解为Aop的标注，Advice/Pointcut/Advisor/AopInfrastructureBean（Aop功能的实现bean），以及是否是标注了@Aspect注解的切面类

3. 紧接着判断是否sholudSkip，是否跳过

   3.1 获取所有的候选增强器（切面中的通知方法）【List candidateAdvisors】，每一个封装通知方法的增强器的类型是InstantiationModelAwarePointcutAdvisor



通过debug我们是可以得到验证的。然后判断每一个增强器是否是AspectJPointcutAdvisor类型的，如果是就返回true

```
@Override
    protected boolean shouldSkip(Class<?> beanClass, String beanName) {
        // TODO: Consider optimization by caching the list of the aspect
names
        //这里查询所有的候选的增强器（就是切面的通知方法）
        List<Advisor> candidateAdvisors = findCandidateAdvisors();
        for (Advisor advisor : candidateAdvisors) {
            if (advisor instanceof AspectJPointcutAdvisor) {
                if (((AbstractAspectJAdvice)
advisor.getAdvice()).getAspectName().equals(beanName)) {
                    return true;
                }
            }
        }
        return super.shouldSkip(beanClass, beanName);
    }
```

4. postProcessBeforeInstantiation方法执行完毕，就执行创建Calculator对象，然后开始执行postProcessAfterInstantiation

   其内部代码：

```
@Override
    public Object postProcessAfterInitialization(Object bean, String
beanName) throws BeansException {
        if (bean != null) {
            Object cacheKey = getCacheKey(bean.getClass(), beanName);
            if (!this.earlyProxyReferences.contains(cacheKey)) {
                //包装bean
                return wrapIfNecessary(bean, beanName, cacheKey);
            }
        }
        return bean;
    }
```

看到wrapIfNecessary(bean, beanName, cacheKey)这个方法内部就是产生代理对象的基本了！点进去

```
protected Object wrapIfNecessary(Object bean, String beanName, Object
cacheKey) {
        if (beanName != null && this.targetSourcedBeans.contains(beanName))
{
            return bean;
        }
        if (Boolean.FALSE.equals(this.advisedBeans.get(cacheKey))) {
            return bean;
        }
        if (isInfrastructureClass(bean.getClass()) ||
shouldSkip(bean.getClass(), beanName)) {
            this.advisedBeans.put(cacheKey, Boolean.FALSE);
            return bean;
        }

        // Create proxy if we have advice.
        //获取当前bean
        Object[] specificInterceptors =
getAdvicesAndAdvisorsForBean(bean.getClass(), beanName, null);
        if (specificInterceptors != DO_NOT_PROXY) {
            this.advisedBeans.put(cacheKey, Boolean.TRUE);
            Object proxy = createProxy(
                    bean.getClass(), beanName, specificInterceptors, new
SingletonTargetSource(bean));
            this.proxyTypes.put(cacheKey, proxy.getClass());
            return proxy;
        }

        this.advisedBeans.put(cacheKey, Boolean.FALSE);
        return bean;
    }
```

看到这个方法getAdvicesAndAdvisorsForBean(bean.getClass(), beanName, null)，就是获取所有增强器（通知方法）（通过findEligibleAdvisors获取可用的增强器）

```
    ```java
   protected List<Advisor> findEligibleAdvisors(Class<?> beanClass, String
beanName) {
```
```

```
            //获取所有定义的通知
        List<Advisor> candidateAdvisors = findCandidateAdvisors();
            //获取当前bean可用的通知
        List<Advisor> eligibleAdvisors =
findAdvisorsThatCanApply(candidateAdvisors, beanClass, beanName);
        extendAdvisors(eligibleAdvisors);
        if (!eligibleAdvisors.isEmpty()) {
            eligibleAdvisors = sortAdvisors(eligibleAdvisors);
        }
        return eligibleAdvisors;
    }
```

点进去findAdvisorsThatCanApply这个类，内部的逻辑

```
    public static List<Advisor> findAdvisorsThatCanApply(List<Advisor>
candidateAdvisors, Class<?> clazz) {
        if (candidateAdvisors.isEmpty()) {
            return candidateAdvisors;
        }

        List<Advisor> eligibleAdvisors = new LinkedList<Advisor>();
       //遍历所有通知，看是不是IntroductionAdvisor类型的
        for (Advisor candidate : candidateAdvisors) {
            if (candidate instanceof IntroductionAdvisor && canApply(candidate,
clazz)) {
                eligibleAdvisors.add(candidate);
            }
        }

        boolean hasIntroductions = !eligibleAdvisors.isEmpty();
        for (Advisor candidate : candidateAdvisors) {
            if (candidate instanceof IntroductionAdvisor) {
                // already processed
                continue;
            }
               //计算表达式看看增强器（通知）能否作用于这个类上，可以的话就加入可用增强器队列
            if (canApply(candidate, clazz, hasIntroductions)) {
                eligibleAdvisors.add(candidate);
            }
        }
       //返回这个类可用的增强器队列
        return eligibleAdvisors;
    }
```

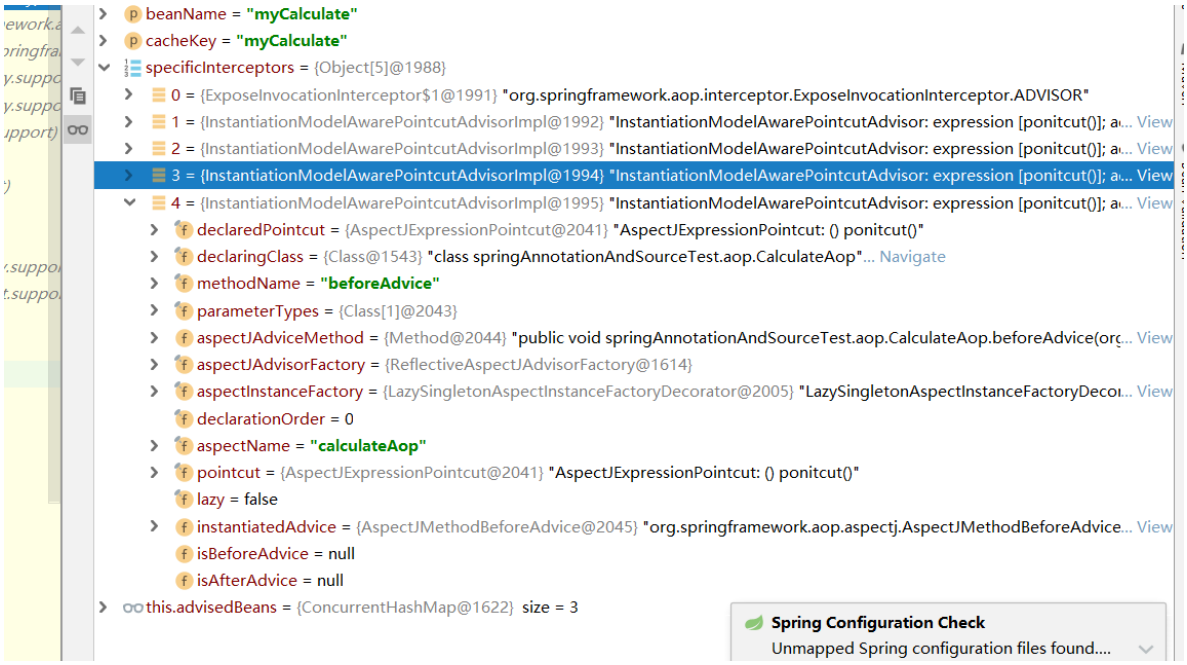
接着又返回这个类可用的增强器队列到findEligibleAdvisors方法中继续往下运行。给增强器进行排序sortAdvisors(eligibleAdvisors);因为通知也是有执行优先级的。然后当排完序后，就会返回排序后的增强器队列给上一级调用方法，也就是wrapIfNecessary中，继续向下执行

```
    //    wrapIfNecessary部分代码

    //specificInterceptors里面装载着这个bean可用的增强器，如下截图
        Object[] specificInterceptors =
getAdvicesAndAdvisorsForBean(bean.getClass(), beanName, null);
            //检查当前bean是否有可用的增强器
        if (specificInterceptors != DO_NOT_PROXY) {
                //如果有就将该bean进行缓存，放入增强bean中记录
```

```
            this.advisedBeans.put(cacheKey, Boolean.TRUE);
               //这一步最关键，为要用到增强器的bean创建处代理对象
            Object proxy = createProxy(
                    bean.getClass(), beanName, specificInterceptors, new
SingletonTargetSource(bean));
            this.proxyTypes.put(cacheKey, proxy.getClass());
            return proxy;
        }

        this.advisedBeans.put(cacheKey, Boolean.FALSE);
```



我们看到**createProxy** 这个方法，用于创建代理对象.点进去查看源码

```
    ** 为指定的bean创建AOP代理对象
     * Create an AOP proxy for the given bean.
     * @param beanClass the class of the bean
     * @param beanName the name of the bean
     * @param specificInterceptors the set of interceptors that is
     * specific to this bean (may be empty, but not null)
     * @param targetSource the TargetSource for the proxy,
     * already pre-configured to access the bean
     * @return the AOP proxy for the bean
     * @see #buildAdvisors
     */
    protected Object createProxy(
            Class<?> beanClass, String beanName, Object[] specificInterceptors,
TargetSource targetSource) {

        if (this.beanFactory instanceof ConfigurableListableBeanFactory) {
            AutoProxyUtils.exposeTargetClass((ConfigurableListableBeanFactory)
this.beanFactory, beanName, beanClass);
        }
           //生成代理工厂
        ProxyFactory proxyFactory = new ProxyFactory();
        proxyFactory.copyFrom(this);

        if (!proxyFactory.isProxyTargetClass()) {
```

```
            if (shouldProxyTargetClass(beanClass, beanName)) {
                proxyFactory.setProxyTargetClass(true);
            }
            else {
                evaluateProxyInterfaces(beanClass, proxyFactory);
            }
        }
           //获取所有增强器（通知方法）保存在代理工厂
        Advisor[] advisors = buildAdvisors(beanName, specificInterceptors);
        for (Advisor advisor : advisors) {
            proxyFactory.addAdvisor(advisor);
        }

        proxyFactory.setTargetSource(targetSource);
        customizeProxyFactory(proxyFactory);

        proxyFactory.setFrozen(this.freezeProxy);
        if (advisorsPreFiltered()) {
            proxyFactory.setPreFiltered(true);
        }
           //重点，利用代理工厂创建代理对象
        return proxyFactory.getProxy(getProxyClassLoader());
    }
```

我们点进getProxy

```
    public Object getProxy(ClassLoader classLoader) {
        return createAopProxy().getProxy(classLoader);
    }

   //点进 createAopProxy中，这里就是真正的创建了一个代理对象了
   @Override
    public AopProxy createAopProxy(AdvisedSupport config) throws
AopConfigException {
        if (config.isOptimize() || config.isProxyTargetClass() ||
hasNoUserSuppliedProxyInterfaces(config)) {
            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.");
            }
            if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {
                return new JdkDynamicAopProxy(config);
            }
            return new ObjenesisCglibAopProxy(config);
        }
        else {
            return new JdkDynamicAopProxy(config);
        }
    }
```

上面createAopProxy方法就是Spring自动决定到底是用JdkDynamicAopProxy（jdk动态代理）还是用ObjenesisCglibAopProxy（cglib动态代理）来创建动态代理，如果类实现了接口，就使用jdk动态代理来创建代理对象。

至此向上返回代理后的结果，给容器。

以后容器中获取得到的就是这个组件的代理对象，执行目标方法的时候，代理对象就会执行通知方法的流程

最后总结，AOP通知方法是如何作用在代理对象上的：

1. 首先spring容器会调用**registerBeanPostProcessors**注册**@EnableAspectJAutoProxy**中的**AnnotationAwareAspectJAutoProxyCreator**（等同于注册了InstantiationAwareBeanPostProcessor 而 InstantiationAwareBeanPostProcessor 又继承了BeanPostProcessor，也就相当于一个BeanPostProcessor），最后会调用BeanFactory的add将这个BeanPostProcessor加入到beanFactory中

   这里会将bean的通知方法注册好，加载进spring中
2. 其次spring在创建bean的时候，会查询是否已经创建了该bean，如果没有就调用GetBean方法，
3. createBean方法内部又调用resolveBeforeInstantiation去查找是否存在这个bean的代理对象，如果存在就返回，不存在就调用doCreateBean方法
4. doCreateBean方法内部先是使用populateBean给bean属性赋值，然后再去调用initializeBean方法初始化bean
5. initializeBean先调用到applyBeanPostProcessorsBeforeInitialization，执行BeanPostProcessor的BeforeInitialization，然后再调用invokeInitMethods，bean自定义的初始化方法，之后又调用applyBeanPostProcessorsAfterInitialization方法
6. 再applyBeanPostProcessorsAfterInitialization中又遍历了次BeanPostProcessors调用他们的postProcessAfterInitialization，之后调用**wrapIfNecessary**
7. wrapIfNecessary方法内 调用getAdvicesAndAdvisorsForBean，获取该bean所有可用的增强器，这里的步骤主要就是遍历寻找定义的增强器（最终是根据bean的名字解析项目中注册的通知方法，解析ponitcut，查找所有的通知方法是否可以作用再这个bean的方法中），最终得到可以作用再这个bean的方法中的增强器（通知方法）
8. 随后，我们将这个bean加入advisedBeans（增强bean集合）标注该bean已经被注册过了
9. 为这个bean生成代理对象， **createProxy** ，这个代理对象有两种代理类型，就是cglib还有jdk代理。
10. 将这个代理对象返回给spring容器中，以后根据beanName取bean的时候就是去到这个代理对象！