课程目标

1、通过分析Spring源码，深刻掌握核心原理和设计思想。
2、通过本课的学习，完全掌握Spring AOP的重要细节。
3、手绘Spring AOP运行时序图。

内容定位

1、Spring 使用不熟练者不适合学习本章内容。
2、先掌握执行流程，再理解设计思想，这个过程至少要花1个月。
3、Spring 源码非常经典，体系也非常庞大，看一遍是远远不够的。

1. Spring AOP 初体验

1.1. 再述 Spring AOP应用场景

AOP是OOP的延续，是Aspect Oriented Programming的缩写，意思是面向切面编程。可以通过预编译方式和运行期动态代理实现在不修改源代码的情况下给程序动态统一添加功能的一种技术。AOP设计模式孜孜不倦追求的是调用者和被调用者之间的解耦，AOP可以说也是这种目标的一种实现。

我们现在做的一些非业务，如：日志、事务、安全等都会写在业务代码中（也即是说，这些非业务类横切于业务类），但这些代码往往是重复，复制粘贴式的代码会给程序的维护带来不便，AOP就实现了把这些业务需求与系统需求分开来做。这种解决的方式也称代理机制。

1.2. AOP中必须明白的几个概念

1.2.1. 切面(Aspect)

官方的抽象定义为“一个关注点的模块化，这个关注点可能会横切多个对象”。“切面”在ApplicationContext 中`<aop:aspect>`来配置。

连接点（Joinpoint）：程序执行过程中的某一行为，例如，MemberService.get的调用或者MemberService.delete 抛出异常等行为。

1.2.2. 通知(Advice)

“切面”对于某个“连接点”所产生的动作。其中，一个“切面”可以包含多个“Advice”。

1.2.3. 切入点(Pointcut)

匹配连接点的断言，在AOP中通知和一个切入点表达式关联。切面中的所有通知所关注的连接点，都由切入点表达式来决定。

1.2.4. 目标对象(Target Object)

被一个或者多个切面所通知的对象。例如，AServcieImpl和BServiceImpl，当然在实际运行时，Spring AOP采用代理实现，实际AOP操作的是TargetObject的代理对象。

1.2.5. AOP切面(AOP Proxy)

在Spring AOP中有两种代理方式，JDK动态代理和CGLib代理。默认情况下，TargetObject实现了接口时，则采用JDK动态代理，例如，AServiceImpl；反之，采用CGLib代理，例如，BServiceImpl。

强制使用CGLib代理需要将`<aop:config>`的proxy-target-class属性设为true。

通知（Advice）类型：

1.2.6. 前置通知(Before Advice)

在某连接点（JoinPoint）之前执行的通知，但这个通知不能阻止连接点前的执行。ApplicationContext中在 `<aop:aspect>` 里面使用 `<aop:before>` 元素进行声明。例如，TestAspect中的 doBefore方法。

1.2.7. 后置通知(After Advice)

当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。ApplicationContext中在 `<aop:aspect>` 里面使用 `<aop:after>` 元素进行声明。例如，ServiceAspect中的returnAfter方法，所以Teser 中调用UserService.delete 抛出异常时，returnAfter方法仍然执行。

1.2.8. 返回后通知(After Return Advice)

在某连接点正常完成后执行的通知，不包括抛出异常的情况。ApplicationContext中在 `<aop:aspect>` 里面使用 `<after-returning>` 元素进行声明。

1.2.9. 环绕通知(Around Advice)

包围一个连接点的通知，类似Web中Servlet规范中的Filter的doFilter方法。可以在方法的调用前后完成自定义的行为，也可以选择不执行。ApplicationContext 中在 `<aop:aspect>` 里面使用 `<aop:around>` 元索进行声明。例如，ServiceAspect中的around方法。

1.2.10. 异常通知(After Throwing Advice)

在方法抛出异常退出时执行的通知。ApplicationContext中在 `<aop:aspect>` 里面使用 `<aopafter-throwing>` 元素进行声明。例如，ServiceAspect中的returnThrow方法。

注：可以将多个通知应用到一个目标对象上，即可以将多个切面织入到同一目标对象。

使用Spring AOP可以基于两种方式，一种是比较方便和强大的注解方式，另一种则是中规中矩的xml配置方式。

先说注解，使用注解配置Spring AOP总体分为两步，第一步是在xml文件中声明激活自动扫描组件功能，同时激活自动代理功能（来测试AOP的注解功能）：

在此之前, 需要先引入相关依赖

```
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.0.2.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
```

```
            <artifactId>spring-aspects</artifactId>
            <version>5.0.2.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-beans</artifactId>
            <version>5.0.2.RELEASE</version>
        </dependency>
    </dependencies>
```

然后编写xml配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">

    <context:component-scan base-package="cn.sitedev"/>
    <context:annotation-config/>
    <!-- 启动@aspectj的自动代理支持-->
    <aop:aspectj-autoproxy/>

</beans>
```

第二步是为Aspect 切面类添加注解：

```
package cn.sitedev.aspect;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.*;
import org.springframework.stereotype.Component;

```

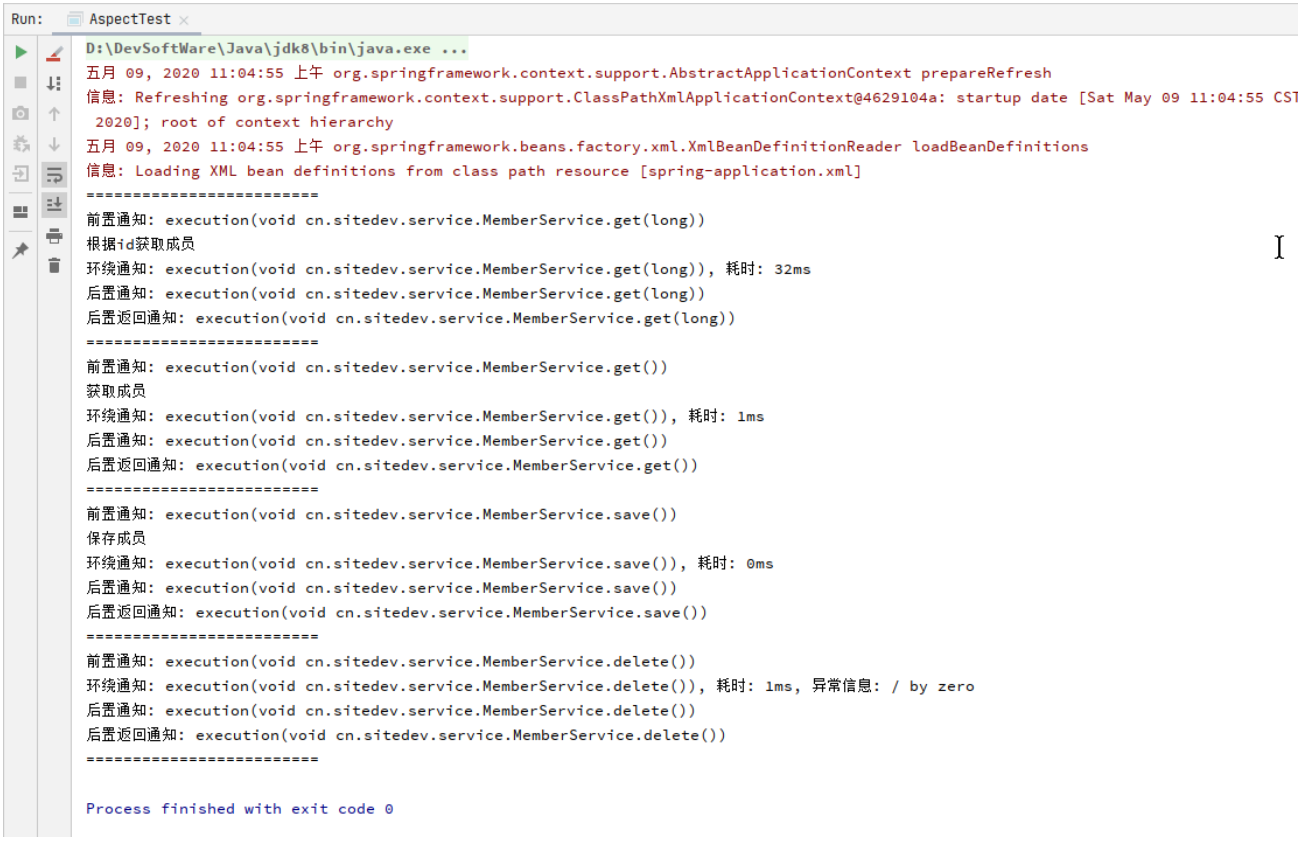
```java
// 声明这是一个组件
@Component
// 声明这是一个切面Bean
@Aspect
public class AnnotationAspect {

    // 配置切入点，该方法无方法体，主要为方便同类中其他方法使用此处配置的切入点
    @Pointcut("execution(* cn.sitedev.service..*(..))")
    public void pointcut() {

    }

    // 配置前置通知，使用在方法pointcut()上注册的切入点
    // 同时接收JoinPoint切入点对象，可以没有该参数
    @Before("pointcut()")
    public void before(JoinPoint joinPoint) {
        System.out.println("前置通知: " + joinPoint);
    }

    // 配置后置通知，使用在方法pointcut()上注册的切入点
    @After("pointcut()")
    public void after(JoinPoint joinPoint) {
        System.out.println("后置通知: " + joinPoint);
    }

    // 配置环绕通知，使用在方法pointcut()上注册的切入点
    @Around("pointcut()")
    public void around(JoinPoint joinPoint) {
        long start = System.currentTimeMillis();
        try {
            ((ProceedingJoinPoint) joinPoint).proceed();
            long end = System.currentTimeMillis();
            System.out.println("环绕通知: " + joinPoint + ", 耗时: " + (end - start) +
        } catch (Throwable throwable) {
            long end = System.currentTimeMillis();
            System.out.println("环绕通知: " + joinPoint + ", 耗时: " + (end - start) +
        }
    }

    // 配置后置返回通知，使用在方法pointcut()上注册的切入点
    @AfterReturning("pointcut()")
    public void afterReturning(JoinPoint joinPoint) {
        System.out.println("后置返回通知: " + joinPoint);
```

```
    }

    // 配置抛出异常后通知，使用在方法pointcut()上注册的切入点
    @AfterThrowing(pointcut = "pointcut()", throwing = "exception")
    public void afterThrowing(JoinPoint joinPoint, Exception exception) {
        System.out.println("抛出异常后通知: " + joinPoint + ", 异常信息: " + exception.
    }
}
```

测试代码

```
package cn.sitedev;

import cn.sitedev.service.MemberService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class AspectTest {
    public static void main(String[] args) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("spring-application
        MemberService memberService = ctx.getBean(MemberService.class);
        System.out.println("=========================");
        memberService.get(12345);
        System.out.println("=========================");
        memberService.get();
        System.out.println("=========================");
        memberService.save();
        System.out.println("=========================");
        memberService.delete();
        System.out.println("=========================");
    }
}
```

控制台输出如下：



可以看到，正如我们预期的那样，虽然我们并没有对MemberService类包括其调用方式做任何改变，但是Spring仍然拦截到了其中方法的调用，或许这正是AOP的魔力所在。

再简单说一下xml配置方式，其实也一样简单：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">

    <bean id="memberService" class="cn.sitedev.service.MemberService"/>

    <bean id="xmlAspect" class="cn.sitedev.aspect.XmlAspect"/>

    <!--AOP配置-->
    <aop:config>
        <!--声明一个切面类，并注入切面bean，相当于@Aspect-->
        <aop:aspect ref="xmlAspect">
            <!--配置一个切入点，相当于@Pointcut-->
            <aop:pointcut id="pointcut" expression="execution(* cn.sitedev.service..*(.
```

```
            <!--配置通知，相当于@Before，@After，@Around，@AfterReturning，@AfterThrowi
            <aop:before method="before" pointcut-ref="pointcut"/>
            <aop:after method="after" pointcut-ref="pointcut"/>
            <aop:around method="around" pointcut-ref="pointcut"/>
            <aop:after-returning method="afterReturning" pointcut-ref="pointcut"/>
            <aop:after-throwing method="afterThrowing" pointcut-ref="pointcut" throwing
        </aop:aspect>
    </aop:config>

</beans>
```

切面类

```
package cn.sitedev.aspect;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;

public class XmlAspect {

    // 配置切入点，该方法无方法体，主要为方便同类中其他方法使用此处配置的切入点
    public void pointcut() {

    }

    // 配置前置通知，使用在方法pointcut()上注册的切入点
    // 同时接收JoinPoint切入点对象，可以没有该参数
    public void before(JoinPoint joinPoint) {
        System.out.println("前置通知：" + joinPoint);
    }

    // 配置后置通知，使用在方法pointcut()上注册的切入点
    public void after(JoinPoint joinPoint) {
        System.out.println("后置通知：" + joinPoint);
    }

    // 配置环绕通知，使用在方法pointcut()上注册的切入点
    public void around(JoinPoint joinPoint) {
        long start = System.currentTimeMillis();
        try {
            ((ProceedingJoinPoint) joinPoint).proceed();
```

```
            long end = System.currentTimeMillis();
            System.out.println("环绕通知: " + joinPoint + ", 耗时: " + (end - start) +
        } catch (Throwable throwable) {
            long end = System.currentTimeMillis();
            System.out.println("环绕通知: " + joinPoint + ", 耗时: " + (end - start) +
        }
    }

    // 配置后置返回通知, 使用在方法pointcut()上注册的切入点
    public void afterReturning(JoinPoint joinPoint) {
        System.out.println("后置返回通知: " + joinPoint);
    }

    // 配置抛出异常后通知, 使用在方法pointcut()上注册的切入点
    public void afterThrowing(JoinPoint joinPoint, Exception exception) {
        System.out.println("抛出异常后通知: " + joinPoint + ", 异常信息: " + exception.
    }
}
```

测试类

```
package cn.sitedev;

import cn.sitedev.service.MemberService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class XmlAspectTest {
    public static void main(String[] args) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("spring-application
        MemberService memberService = ctx.getBean(MemberService.class);
        System.out.println("=========================");
        memberService.get(12345);
        System.out.println("=========================");
        memberService.get();
        System.out.println("=========================");
        memberService.save();
        System.out.println("=========================");
        memberService.delete();
        System.out.println("=========================");
    }
```

```
21 }
```

控制台输出

应该说学习Spring AOP有两个难点，第一点在于理解AOP的理念和相关概念，第二点在于灵活掌握和使用切入点表达式。概念的理解通常不在一朝一夕，慢慢浸泡的时间长了，自然就明白了，下面我们简单地介绍一下切入点表达式的配置规则吧。

通常情况下，表达式中使用”execution“就可以满足大部分的要求。表达式格式如下：

`execution(<修饰符模式>?<返回类型模式><方法所在类的完全限定名称模式>(<参数模式>)<异常模式>?)`

`execution(modifiers-pattern? ret-type-pattern fully-qualified-class-name(param-pattern) throws-pattern?)`

modifiers-pattern：方法的操作权限

ret-type-pattern：返回值

declaring-type-pattern：方法所在的包

name-pattern：方法名parm-pattern：参数名

throws-pattern：异常

其中，除ret-type-pattern和name-pattern之外，其他都是可选的。上例中，`execution（* cn.sitedev.service.**（.））`表示cn.sitedev.service包下，返回值为任意类型；方法名任意；参数不作限制的所有方法。

最后说一下通知参数，可以通过args来绑定参数，这样就可以在通知（Advice）中访问具体参数了。
例如，`<aop:aspect>`配置如下：

```
    <!--AOP配置-->
    <aop:config>
        <!--声明一个切面类，并注入切面bean，相当于@Aspect-->
        <aop:aspect ref="xmlAspect">
            <!--配置一个切入点，相当于@Pointcut-->
            <aop:pointcut id="pointcut" expression="execution(* cn.sitedev.service..*(.
```

上面的代码args（msg..）是指将切入点方法上的第一个String 类型参数添加到参数名为msg的通知的入参上，这样就可以直接使用该参数啦。

在上面的Aspect切面Bean中已经看到了，每个通知方法第一个参数都是JoinPoint。其实，在Spring中，任何通知（Advice）方法都可以将第一个参数定义为org.aspectj.lang.JoinPoint类型用以接受当前连接点对象。JoinPoint 接口提供了一系列有用的方法，比如getArgs()（返回方法参数）、getThis()（返回代理对象）、getTarget()（返回目标）、getSignature()（返回正在被通知的方法相关信息）和toString()（打印出正在被通知的方法的有用信息）。

2. Spring AOP 源码分析

2.1. 寻找入口

Spring的AOP是通过接入BeanPostProcessor后置处理器开始的，它是Spring IOC容器经常使用到的一个特性，这个Bean后置处理器是一个监听器，可以监听容器触发的Bean声明周期事件。后置处理器向容器注册以后，容器中管理的Bean就具备了接收IOC容器事件回调的能力。
BeanPostProcessor的使用非常简单，只需要提供一个实现接口BeanPostProcessor的实现类，然后在Bean的配置文件中设置即可。

2.1.1. BeanPostProcessor源码

```
package org.springframework.beans.factory.config;

import org.springframework.beans.BeansException;
import org.springframework.lang.Nullable;

```

```
public interface BeanPostProcessor {

    //为在Bean的初始化前提供回调入口
    @Nullable
    default Object postProcessBeforeInitialization(Object bean, String beanName) throws
        return bean;
    }

    //为在Bean的初始化之后提供回调入口
    @Nullable
    default Object postProcessAfterInitialization(Object bean, String beanName) throws
        return bean;
    }

}
```

这两个回调的入口都是和容器管理的Bean的生命周期事件紧密相关，可以为用户提供在Spring IOC容器初始化Bean过程中自定义的处理操作。

2.1.2. AbstractAutowireCapableBeanFactory 类对容器生成的Bean 添加后置处理器

BeanPostProcessor 后置处理器的调用发生在Spring IOC容器完成对Bean实例对象的创建和属性的依赖注入完成之后在对Spring 依赖注入的源码分析过程中我们知道，当应用程序第一次调用getBean()方法（lazy-init 预实例化除外）向Spring IOC容器索取指定Bean时触发Spring IOC容器创建Bean实例对象并进行依赖注入的过程，其中真正实现创建Bean对象并进行依赖注入的方法是AbstractAutowireCapableBeanFactory 类的 doCreateBean()方法，主要源码如下：

```
    //真正创建Bean的方法
    protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd,
            throws BeanCreationException {

        // Instantiate the bean.
        //封装被创建的Bean对象
        ...

        // Initialize the bean instance.
        //Bean对象的初始化，依赖注入在此触发
        //这个exposedObject在初始化完成之后返回作为依赖注入完成后的Bean
        Object exposedObject = bean;
        try {
```

```
            //将Bean实例对象封装，并且Bean定义中配置的属性值赋值给实例对象
            populateBean(beanName, mbd, instanceWrapper);
            //初始化Bean对象
            exposedObject = initializeBean(beanName, exposedObject, mbd);
        }
        catch (Throwable ex) {
            if (ex instanceof BeanCreationException && beanName.equals(((BeanCreationEx
                throw (BeanCreationException) ex;
            }
            else {
                throw new BeanCreationException(
                        mbd.getResourceDescription(), beanName, "Initialization of bean
            }
        }
        ...

        return exposedObject;
    }
```

从上面的代码中我们知道，为Bean实例对象添加BeanPostProcessor后置处理器的入口的是initializeBean()方法。

2.1.3. initializeBean()方法为容器产生的Bean 实例对象添加BeanPostProcessor 后置处理器

同样在AbstractAutowireCapableBeanFactory 类中，initializeBean()方法实现为容器创建的Bean实例对象添加BeanPostProcessor后置处理器，源码如下：

```
    protected Object initializeBean(final String beanName, final Object bean, @Nullable
        //JDK的安全机制验证权限
        if (System.getSecurityManager() != null) {
            //实现PrivilegedAction接口的匿名内部类
            AccessController.doPrivileged((PrivilegedAction<Object>) () -> {
                invokeAwareMethods(beanName, bean);
                return null;
            }, getAccessControlContext());
        }
        else {
            //为Bean实例对象包装相关属性，如名称，类加载器，所属容器等信息
            invokeAwareMethods(beanName, bean);
        }
```

```
14
15          Object wrappedBean = bean;
16          //对BeanPostProcessor后置处理器的postProcessBeforeInitialization
17          //回调方法的调用，为Bean实例初始化前做一些处理
18          if (mbd == null || !mbd.isSynthetic()) {
19              wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, bean
20          }
21
22          //调用Bean实例对象初始化的方法，这个初始化方法是在Spring Bean定义配置
23          //文件中通过init-method属性指定的
24          try {
25              invokeInitMethods(beanName, wrappedBean, mbd);
26          }
27          catch (Throwable ex) {
28              throw new BeanCreationException(
29                      (mbd != null ? mbd.getResourceDescription() : null),
30                      beanName, "Invocation of init method failed", ex);
31          }
32          //对BeanPostProcessor后置处理器的postProcessAfterInitialization
33          //回调方法的调用，为Bean实例初始化之后做一些处理
34          if (mbd == null || !mbd.isSynthetic()) {
35              wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanN
36          }
37
38          return wrappedBean;
39      }
40
41      @Override
42      //调用BeanPostProcessor后置处理器实例对象初始化之前的处理方法
43      public Object applyBeanPostProcessorsBeforeInitialization(Object existingBean, Stri
44              throws BeansException {
45          Object result = existingBean;
46          //遍历容器为所创建的Bean添加的所有BeanPostProcessor后置处理器
47          for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
48              //调用Bean实例所有的后置处理中的初始化前处理方法，为Bean实例对象在
49              //初始化之前做一些自定义的处理操作
50              Object current = beanProcessor.postProcessBeforeInitialization(result, bean
51              if (current == null) {
52                  return result;
53              }
54              result = current;
55          }
56          return result;
```

```
    }

    @Override
    //调用BeanPostProcessor后置处理器实例对象初始化之后的处理方法
    public Object applyBeanPostProcessorsAfterInitialization(Object existingBean, Strin
            throws BeansException {

        Object result = existingBean;
        //遍历容器为所创建的Bean添加的所有BeanPostProcessor后置处理器
        for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
            //调用Bean实例所有的后置处理中的初始化后处理方法，为Bean实例对象在
            //初始化之后做一些自定义的处理操作
            Object current = beanProcessor.postProcessAfterInitialization(result, beanN
            if (current == null) {
                return result;
            }
            result = current;
        }
        return result;
    }
```

BeanPostProcessor是一个接口，其初始化前的操作方法和初始化后的操作方法均委托其实现子类来实现，在Spring中，BeanPostProcessor的实现子类非常的多，分别完成不同的操作，如：AOP面向切面编程的注册通知适配器、Bean对象的数据校验、Bean 继承属性、方法的合并等等，我们以最简单的AOP切面织入来简单了解其主要的功能。下面我们来分析其中一个创建AOP代理对象的子类AbstractAutoProxyCreator类。该类重写了postProcessAfterInitialization()方法。

2.2. 选择代理策略

进入postProcessAfterInitialization()方法，我们发现调到了一个非常核心的方法wrapIfNecessary()，其源码如下：

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

```
11
12      protected Object getCacheKey(Class<?> beanClass, @Nullable String beanName) {
13          if (StringUtils.hasLength(beanName)) {
14              return (FactoryBean.class.isAssignableFrom(beanClass) ?
15                      BeanFactory.FACTORY_BEAN_PREFIX + beanName : beanName);
16          }
17          else {
18              return beanClass;
19          }
20      }
21
22      protected Object wrapIfNecessary(Object bean, String beanName, Object cacheKey) {
23          if (StringUtils.hasLength(beanName) && this.targetSourcedBeans.contains(beanNam
24              return bean;
25          }
26          // 判断是否不应该代理这个bean
27          if (Boolean.FALSE.equals(this.advisedBeans.get(cacheKey))) {
28              return bean;
29          }
30          // 判断是否是一些isInfrastructureClass或者是否应该跳过这个bean
31          // 所谓isInfrastructureClass 就是指Advice/Pointcut/Advisor 等接口的实现类
32          // shouldSkip 默认实现为返回false，由于是protected方法，子类可以覆盖
33          if (isInfrastructureClass(bean.getClass()) || shouldSkip(bean.getClass(), beanN
34              this.advisedBeans.put(cacheKey, Boolean.FALSE);
35              return bean;
36          }
37
38          // 获取这个bean的advice
39          // Create proxy if we have advice.
40          Object[] specificInterceptors = getAdvicesAndAdvisorsForBean(bean.getClass(), b
41          if (specificInterceptors != DO_NOT_PROXY) {
42              this.advisedBeans.put(cacheKey, Boolean.TRUE);
43              Object proxy = createProxy(
44                      bean.getClass(), beanName, specificInterceptors, new SingletonTarge
45              this.proxyTypes.put(cacheKey, proxy.getClass());
46              return proxy;
47          }
48
49          this.advisedBeans.put(cacheKey, Boolean.FALSE);
50          return bean;
51      }
52
53      ...
```

```
54
55      protected Object createProxy(Class<?> beanClass, @Nullable String beanName,
56              @Nullable Object[] specificInterceptors, TargetSource targetSource) {
57
58          if (this.beanFactory instanceof ConfigurableListableBeanFactory) {
59              AutoProxyUtils.exposeTargetClass((ConfigurableListableBeanFactory) this.bea
60          }
61
62          ProxyFactory proxyFactory = new ProxyFactory();
63          proxyFactory.copyFrom(this);
64
65          if (!proxyFactory.isProxyTargetClass()) {
66              if (shouldProxyTargetClass(beanClass, beanName)) {
67                  proxyFactory.setProxyTargetClass(true);
68              }
69              else {
70                  evaluateProxyInterfaces(beanClass, proxyFactory);
71              }
72          }
73
74          Advisor[] advisors = buildAdvisors(beanName, specificInterceptors);
75          proxyFactory.addAdvisors(advisors);
76          proxyFactory.setTargetSource(targetSource);
77          customizeProxyFactory(proxyFactory);
78
79          proxyFactory.setFrozen(this.freezeProxy);
80          if (advisorsPreFiltered()) {
81              proxyFactory.setPreFiltered(true);
82          }
83
84          return proxyFactory.getProxy(getProxyClassLoader());
85      }
```

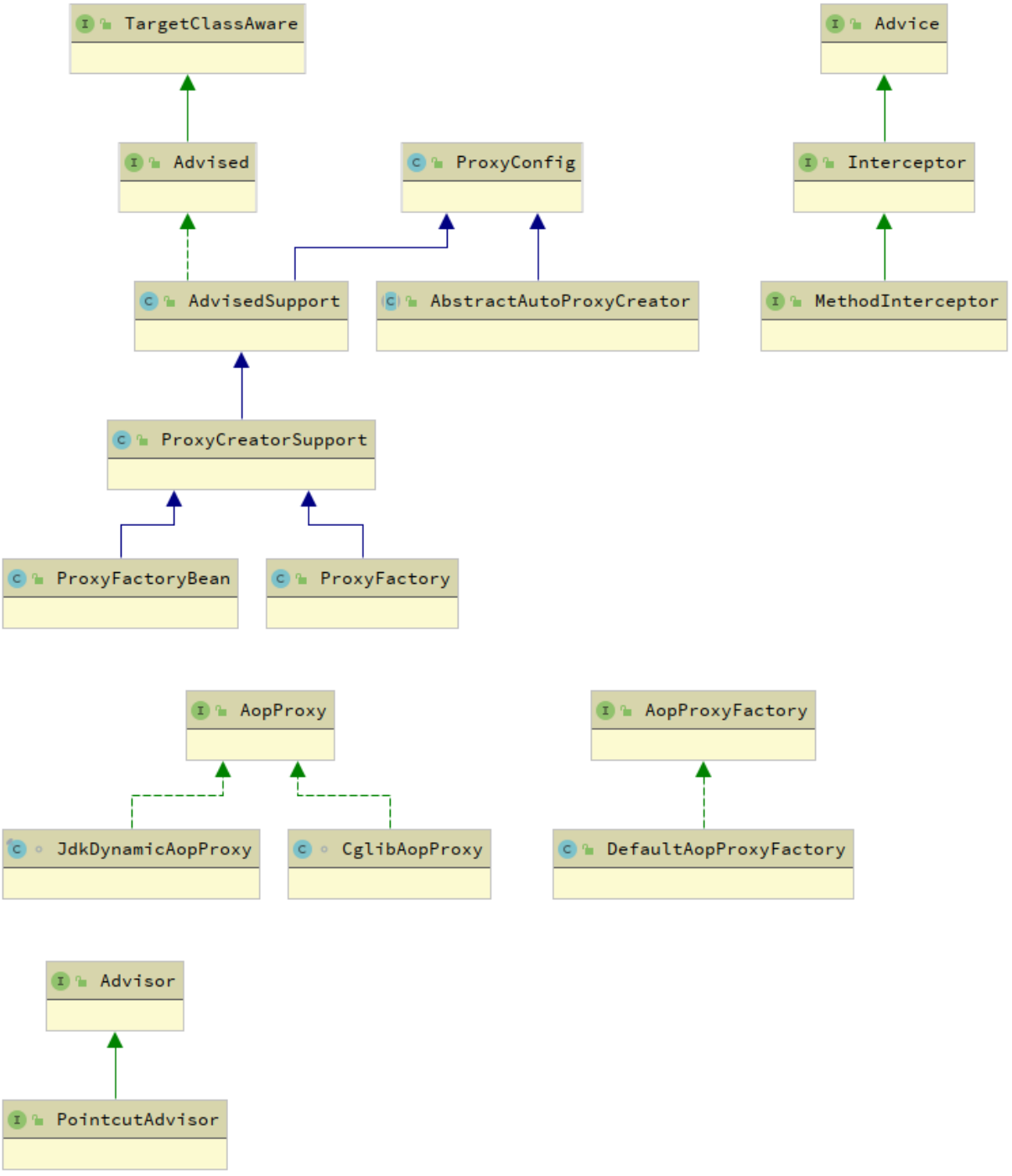
整个过程跟下来，我发现最终调用的是proxyFactory.getProxy()方法。到这里我们大概能够猜到proxyFactory 有JDK和CGLib的，那么我们该如何选择呢？最终调用的是DefaultAopProxyFactory的createAopProxy()方法：

```
1 ackage org.springframework.aop.framework;
2
3 import java.io.Serializable;
4 import java.lang.reflect.Proxy;
```

```
5
6  import org.springframework.aop.SpringProxy;
7
8  @SuppressWarnings("serial")
9  public class DefaultAopProxyFactory implements AopProxyFactory, Serializable {
10
11     @Override
12     public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {
13         if (config.isOptimize() || config.isProxyTargetClass() || hasNoUserSuppliedProx
14             Class<?> targetClass = config.getTargetClass();
15             if (targetClass == null) {
16                 throw new AopConfigException("TargetSource cannot determine target clas
17                         "Either an interface or a target is required for proxy creation
18             }
19             if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {
20                 return new JdkDynamicAopProxy(config);
21             }
22             return new ObjenesisCglibAopProxy(config);
23         }
24         else {
25             return new JdkDynamicAopProxy(config);
26         }
27     }
28
29     private boolean hasNoUserSuppliedProxyInterfaces(AdvisedSupport config) {
30         Class<?>[] ifcs = config.getProxiedInterfaces();
31         return (ifcs.length == 0 || (ifcs.length == 1 && SpringProxy.class.isAssignable
32     }
33
34 }
```

2.3. 调用代理方法

分析调用逻辑之前先上类图，看看Spring中主要的AOP组件：



上面我们已经了解到Spring提供了两种方式来生成代理方式有JDKProxy和CGLib。下面我们来研究一下Spring 如何使用JDK来生成代理对象，具体的生成代码放在JdkDynamicAopProxy这个类中，直接上相关代码：

```
    @Override
    public Object getProxy() {
        return getProxy(ClassUtils.getDefaultClassLoader());
    }

    /**
     * 获取代理类要实现的接口,除了Advised对象中配置的,还会加上SpringProxy, Advised(opaq
```

```
     * 检查上面得到的接口中有没有定义 equals或者hashcode的接口
     * 调用Proxy.newProxyInstance创建代理对象
     */
    @Override
    public Object getProxy(@Nullable ClassLoader classLoader) {
        if (logger.isDebugEnabled()) {
            logger.debug("Creating JDK dynamic proxy: target source is " + this.advised
        }
        Class<?>[] proxiedInterfaces = AopProxyUtils.completeProxiedInterfaces(this.adv
        findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);
        return Proxy.newProxyInstance(classLoader, proxiedInterfaces, this);
    }
```

通过注释我们应该已经看得非常明白代理对象的生成过程，此处不再赘述。下面的问题是，代理对象生成了，那切面是如何织入的？

我们知道 InvocationHandler是JDK动态代理的核心，生成的代理对象的方法调用都会委托到InvocationHandler.invoke()方法。而从JdkDynamicAopProxy的源码我们可以看到这个类其实也实现了InvocationHandler，下面我们分析Spring AOP是如何织入切面的，直接上源码看invoke()方法：

```
    @Override
    @Nullable
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        MethodInvocation invocation;
        Object oldProxy = null;
        boolean setProxyContext = false;

        TargetSource targetSource = this.advised.targetSource;
        Object target = null;

        try {
            //eqauls()方法，具目标对象未实现此方法
            if (!this.equalsDefined && AopUtils.isEqualsMethod(method)) {
                // The target does not implement the equals(Object) method itself.
                return equals(args[0]);
            }
            //hashCode()方法，具目标对象未实现此方法
            else if (!this.hashCodeDefined && AopUtils.isHashCodeMethod(method)) {
                // The target does not implement the hashCode() method itself.
                return hashCode();
            }
```

```
            else if (method.getDeclaringClass() == DecoratingProxy.class) {
                // There is only getDecoratedClass() declared -> dispatch to proxy conf
                return AopProxyUtils.ultimateTargetClass(this.advised);
            }
            //Advised接口或者其父接口中定义的方法,直接反射调用,不应用通知
            else if (!this.advised.opaque && method.getDeclaringClass().isInterface() &
                    method.getDeclaringClass().isAssignableFrom(Advised.class)) {
                // Service invocations on ProxyConfig with the proxy config...
                return AopUtils.invokeJoinpointUsingReflection(this.advised, method, ar
            }

            Object retVal;

            if (this.advised.exposeProxy) {
                // Make invocation available if necessary.
                oldProxy = AopContext.setCurrentProxy(proxy);
                setProxyContext = true;
            }

            // Get as late as possible to minimize the time we "own" the target,
            // in case it comes from a pool.
            //获得目标对象的类
            target = targetSource.getTarget();
            Class<?> targetClass = (target != null ? target.getClass() : null);

            // Get the interception chain for this method.
            //获取可以应用到此方法上的Interceptor列表
            List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvi

            // Check whether we have any advice. If we don't, we can fallback on direct
            // reflective invocation of the target, and avoid creating a MethodInvocati
            //如果没有可以应用到此方法的通知(Interceptor)，此直接反射调用 method.invoke
            if (chain.isEmpty()) {
                // We can skip creating a MethodInvocation: just invoke the target dire
                // Note that the final invoker must be an InvokerInterceptor so we know
                // nothing but a reflective operation on the target, and no hot swappin
                Object[] argsToUse = AopProxyUtils.adaptArgumentsIfNecessary(method, ar
                retVal = AopUtils.invokeJoinpointUsingReflection(target, method, argsTo
            }
            else {
                // We need to create a method invocation...
                //创建MethodInvocation
                invocation = new ReflectiveMethodInvocation(proxy, target, method, args
```

```
                // Proceed to the joinpoint through the interceptor chain.
                retVal = invocation.proceed();
            }

            // Massage return value if necessary.
            Class<?> returnType = method.getReturnType();
            if (retVal != null && retVal == target &&
                    returnType != Object.class && returnType.isInstance(proxy) &&
                    !RawTargetAccess.class.isAssignableFrom(method.getDeclaringClass())
                // Special case: it returned "this" and the return type of the method
                // is type-compatible. Note that we can't help if the target sets
                // a reference to itself in another returned object.
                retVal = proxy;
            }
            else if (retVal == null && returnType != Void.TYPE && returnType.isPrimitiv
                throw new AopInvocationException(
                        "Null return value from advice does not match primitive return
            }
            return retVal;
        }
        finally {
            if (target != null && !targetSource.isStatic()) {
                // Must have come from TargetSource.
                targetSource.releaseTarget(target);
            }
            if (setProxyContext) {
                // Restore old proxy.
                AopContext.setCurrentProxy(oldProxy);
            }
        }
    }
```

主要实现思路可以简述为：首先获取应用到此方法上的通知链（Interceptor Chain）。如果有通知，则应用通知，并执行JoinPoint；如果没有通知，则直接反射执行JoinPoint。而这里的关键是通知链是如何获取的以及它又是如何执行的呢？现在来逐一分析。首先，从上面的代码可以看到，通知链是通过Advised.getInterceptorsAndDynamicInterceptionAdvice）这个方法来获取的，我们来看下这个方法的实现逻辑：

```
    public List<Object> getInterceptorsAndDynamicInterceptionAdvice(Method method, @Nul
        MethodCacheKey cacheKey = new MethodCacheKey(method);
        List<Object> cached = this.methodCache.get(cacheKey);
```

```
        if (cached == null) {
            cached = this.advisorChainFactory.getInterceptorsAndDynamicInterceptionAdvi
                    this, method, targetClass);
            this.methodCache.put(cacheKey, cached);
        }
        return cached;
    }
```

通过上面的源码我们可以看到，实际获取通知的实现逻辑其实是由AdvisorChainFactory的getInterceptorsAndDynamicInterceptionAdvice()方法来完成的，且获取到的结果会被缓存。下面来分析 getInterceptorsAndDynamicInterceptionAdvice()方法的实现：

```
    /**
     * 从提供的配置实例config中获取advisor列表,遍历处理这些advisor.如果是IntroductionAd
     * 则判断此Advisor能否应用到目标类targetClass上.如果是PointcutAdvisor,则判断
     * 此Advisor能否应用到目标方法method上.将满足条件的Advisor通过AdvisorAdaptor转化成I
     */
    @Override
    public List<Object> getInterceptorsAndDynamicInterceptionAdvice(
            Advised config, Method method, @Nullable Class<?> targetClass) {

        // This is somewhat tricky... We have to process introductions first,
        // but we need to preserve order in the ultimate list.
        List<Object> interceptorList = new ArrayList<>(config.getAdvisors().length);
        Class<?> actualClass = (targetClass != null ? targetClass : method.getDeclaring
        //查看是否包含IntroductionAdvisor
        boolean hasIntroductions = hasMatchingIntroductions(config, actualClass);
        //这里实际上注册一系列AdvisorAdapter,用于将Advisor转化成MethodInterceptor
        AdvisorAdapterRegistry registry = GlobalAdvisorAdapterRegistry.getInstance();

        for (Advisor advisor : config.getAdvisors()) {
            if (advisor instanceof PointcutAdvisor) {
                // Add it conditionally.
                PointcutAdvisor pointcutAdvisor = (PointcutAdvisor) advisor;
                if (config.isPreFiltered() || pointcutAdvisor.getPointcut().getClassFil
                    //这个地方这两个方法的位置可以互换下
                    //将Advisor转化成Interceptor
                    MethodInterceptor[] interceptors = registry.getInterceptors(advisor
                    //检查当前advisor的pointcut是否可以匹配当前方法
                    MethodMatcher mm = pointcutAdvisor.getPointcut().getMethodMatcher()
                    if (MethodMatchers.matches(mm, method, actualClass, hasIntroduction
```

```
                        if (mm.isRuntime()) {
                            // Creating a new object instance in the getInterceptors()
                            // isn't a problem as we normally cache created chains.
                            for (MethodInterceptor interceptor : interceptors) {
                                interceptorList.add(new InterceptorAndDynamicMethodMatc
                            }
                        }
                        else {
                            interceptorList.addAll(Arrays.asList(interceptors));
                        }
                    }
                }
            }
            else if (advisor instanceof IntroductionAdvisor) {
                IntroductionAdvisor ia = (IntroductionAdvisor) advisor;
                if (config.isPreFiltered() || ia.getClassFilter().matches(actualClass))
                    Interceptor[] interceptors = registry.getInterceptors(advisor);
                    interceptorList.addAll(Arrays.asList(interceptors));
                }
            }
            else {
                Interceptor[] interceptors = registry.getInterceptors(advisor);
                interceptorList.addAll(Arrays.asList(interceptors));
            }
        }

        return interceptorList;
    }
```

这个方法执行完成后，Advised中配置能够应用到连接点（JoinPoint）或者目标类（Target Object）的Advisor 全部被转化成了MethodInterceptor，接下来我们再看下得到的拦截器链是怎么起作用的。

```
            // Check whether we have any advice. If we don't, we can fallback on direct
            // reflective invocation of the target, and avoid creating a MethodInvocati
            //如果没有可以应用到此方法的通知(Interceptor)，此直接反射调用 method.invoke
            if (chain.isEmpty()) {
                // We can skip creating a MethodInvocation: just invoke the target dire
                // Note that the final invoker must be an InvokerInterceptor so we know
                // nothing but a reflective operation on the target, and no hot swappin
                Object[] argsToUse = AopProxyUtils.adaptArgumentsIfNecessary(method, ar
```

```
                retVal = AopUtils.invokeJoinpointUsingReflection(target, method, argsTo
            }
            else {
                // We need to create a method invocation...
                //创建MethodInvocation
                invocation = new ReflectiveMethodInvocation(proxy, target, method, args
                // Proceed to the joinpoint through the interceptor chain.
                retVal = invocation.proceed();
            }
```

从这段代码可以看出，如果得到的拦截器链为空，则直接反射调用目标方法，否则创建MethodInvocation，调用其proceed()方法，触发拦截器链的执行，来看下具体代码：

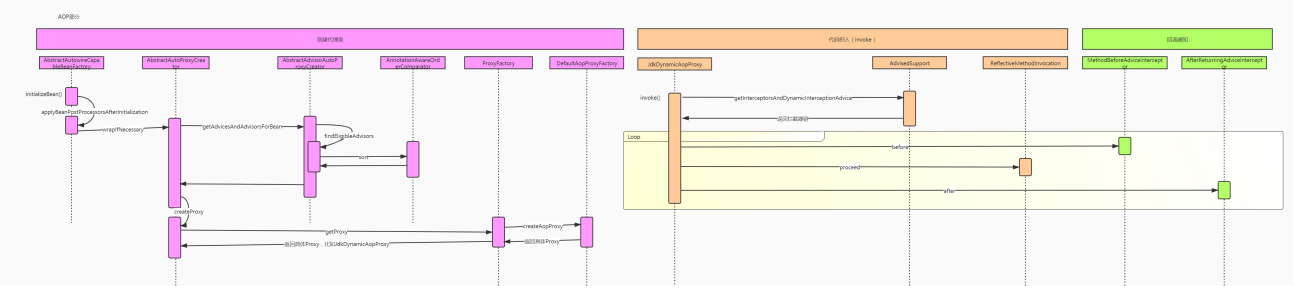
```
    @Override
    @Nullable
    public Object proceed() throws Throwable {
        //    We start with an index of -1 and increment early.
        //如果Interceptor执行完了，则执行joinPoint
        if (this.currentInterceptorIndex == this.interceptorsAndDynamicMethodMatchers.s
            return invokeJoinpoint();
        }

        Object interceptorOrInterceptionAdvice =
                this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptor
        //如果要动态匹配joinPoint
        if (interceptorOrInterceptionAdvice instanceof InterceptorAndDynamicMethodMatch
            // Evaluate dynamic method matcher here: static part will already have
            // been evaluated and found to match.
            InterceptorAndDynamicMethodMatcher dm =
                    (InterceptorAndDynamicMethodMatcher) interceptorOrInterceptionAdvic
            //动态匹配：运行时参数是否满足匹配条件
            if (dm.methodMatcher.matches(this.method, this.targetClass, this.arguments)
                return dm.interceptor.invoke(this);
            }
            else {
                // Dynamic matching failed.
                // Skip this interceptor and invoke the next in the chain.
                //动态匹配失败时,略过当前Intercetpor,调用下一个Interceptor
                return proceed();
            }
        }
```

```
29          else {
30              // It's an interceptor, so we just invoke it: The pointcut will have
31              // been evaluated statically before this object was constructed.
32              //执行当前Intercetpor
33              return ((MethodInterceptor) interceptorOrInterceptionAdvice).invoke(this);
34          }
35      }
```

至此，通知链就完美地形成了。我们再往下来看invokeJoinpointUsingReflection()方法，其实就是反射调用：

```
1       public static Object invokeJoinpointUsingReflection(@Nullable Object target, Method
2               throws Throwable {
3
4           // Use reflection to invoke the method.
5           try {
6               ReflectionUtils.makeAccessible(method);
7               return method.invoke(target, args);
8           }
9           catch (InvocationTargetException ex) {
10              // Invoked method threw a checked exception.
11              // We must rethrow it. The client won't see the interceptor.
12              throw ex.getTargetException();
13          }
14          catch (IllegalArgumentException ex) {
15              throw new AopInvocationException("AOP configuration seems to be invalid: tr
16                      method + "] on target [" + target + "]", ex);
17          }
18          catch (IllegalAccessException ex) {
19              throw new AopInvocationException("Could not access method [" + method + "]"
20          }
21      }
```

Spring AOP源码就分析到这儿，相信小伙伴们应该有了基本思路，下面时序图来一波。



2.4. 触发通知

在为AopProxy代理对象配置拦截器的实现中，有一个取得拦截器的配置过程，这个过程是由DefaultAdvisorChainFactory实现的，这个工厂类负责生成拦截器链，在它的getInterceptorsAndDynamicInterceptionAdvice方法中，有一个适配器和注册过程，通过配置Spring 预先设计好的拦截器，Spring 加入了它对AOP实现的处理。

```
    /**
     * 从提供的配置实例config中获取advisor列表,遍历处理这些advisor.如果是IntroductionAd
     * 则判断此Advisor能否应用到目标类targetClass上.如果是PointcutAdvisor,则判断
     * 此Advisor能否应用到目标方法method上.将满足条件的Advisor通过AdvisorAdaptor转化成I
     */
    @Override
    public List<Object> getInterceptorsAndDynamicInterceptionAdvice(
            Advised config, Method method, @Nullable Class<?> targetClass) {

        // This is somewhat tricky... We have to process introductions first,
        // but we need to preserve order in the ultimate list.
        List<Object> interceptorList = new ArrayList<>(config.getAdvisors().length);
        Class<?> actualClass = (targetClass != null ? targetClass : method.getDeclaring
        //查看是否包含IntroductionAdvisor
        boolean hasIntroductions = hasMatchingIntroductions(config, actualClass);
        //这里实际上注册一系列AdvisorAdapter,用于将Advisor转化成MethodInterceptor
        AdvisorAdapterRegistry registry = GlobalAdvisorAdapterRegistry.getInstance();

        ...

        return interceptorList;
    }
```

GlobalAdvisorAdapterRegistry 负责拦截器的适配和注册过程。

```
/*
 * Copyright 2002-2012 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
```

```
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.springframework.aop.framework.adapter;

/**
 * Singleton to publish a shared DefaultAdvisorAdapterRegistry instance.
 *
 * @author Rod Johnson
 * @author Juergen Hoeller
 * @author Phillip Webb
 * @see DefaultAdvisorAdapterRegistry
 */
public abstract class GlobalAdvisorAdapterRegistry {

    /**
     * Keep track of a single instance so we can return it to classes that request it.
     */
    private static AdvisorAdapterRegistry instance = new DefaultAdvisorAdapterRegistry(

    /**
     * Return the singleton {@link DefaultAdvisorAdapterRegistry} instance.
     */
    public static AdvisorAdapterRegistry getInstance() {
        return instance;
    }

    /**
     * Reset the singleton {@link DefaultAdvisorAdapterRegistry}, removing any
     * {@link AdvisorAdapterRegistry#registerAdvisorAdapter(AdvisorAdapter) registered}
     * adapters.
     */
    static void reset() {
        instance = new DefaultAdvisorAdapterRegistry();
    }

}
```

而GlobalAdvisorAdapterRegistry 起到了适配器和单例模式的作用，提供了一个DefaultAdvisorAdapterRegistry，它用来完成各种通知的适配和注册过程。

```
package org.springframework.aop.framework.adapter;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

import org.aopalliance.aop.Advice;
import org.aopalliance.intercept.MethodInterceptor;

import org.springframework.aop.Advisor;
import org.springframework.aop.support.DefaultPointcutAdvisor;

/**
 * Default implementation of the {@link AdvisorAdapterRegistry} interface.
 * Supports {@link org.aopalliance.intercept.MethodInterceptor},
 * {@link org.springframework.aop.MethodBeforeAdvice},
 * {@link org.springframework.aop.AfterReturningAdvice},
 * {@link org.springframework.aop.ThrowsAdvice}.
 *
 * @author Rod Johnson
 * @author Rob Harrop
 * @author Juergen Hoeller
 */
@SuppressWarnings("serial")
public class DefaultAdvisorAdapterRegistry implements AdvisorAdapterRegistry, Serializa

    private final List<AdvisorAdapter> adapters = new ArrayList<>(3);


    /**
     * Create a new DefaultAdvisorAdapterRegistry, registering well-known adapters.
     */
    public DefaultAdvisorAdapterRegistry() {
        registerAdvisorAdapter(new MethodBeforeAdviceAdapter());
        registerAdvisorAdapter(new AfterReturningAdviceAdapter());
        registerAdvisorAdapter(new ThrowsAdviceAdapter());
    }


```

```
40      @Override
41      public Advisor wrap(Object adviceObject) throws UnknownAdviceTypeException {
42          if (adviceObject instanceof Advisor) {
43              return (Advisor) adviceObject;
44          }
45          if (!(adviceObject instanceof Advice)) {
46              throw new UnknownAdviceTypeException(adviceObject);
47          }
48          Advice advice = (Advice) adviceObject;
49          if (advice instanceof MethodInterceptor) {
50              // So well-known it doesn't even need an adapter.
51              return new DefaultPointcutAdvisor(advice);
52          }
53          for (AdvisorAdapter adapter : this.adapters) {
54              // Check that it is supported.
55              if (adapter.supportsAdvice(advice)) {
56                  return new DefaultPointcutAdvisor(advice);
57              }
58          }
59          throw new UnknownAdviceTypeException(advice);
60      }
61
62      @Override
63      public MethodInterceptor[] getInterceptors(Advisor advisor) throws UnknownAdviceTyp
64          List<MethodInterceptor> interceptors = new ArrayList<>(3);
65          Advice advice = advisor.getAdvice();
66          if (advice instanceof MethodInterceptor) {
67              interceptors.add((MethodInterceptor) advice);
68          }
69          for (AdvisorAdapter adapter : this.adapters) {
70              if (adapter.supportsAdvice(advice)) {
71                  interceptors.add(adapter.getInterceptor(advisor));
72              }
73          }
74          if (interceptors.isEmpty()) {
75              throw new UnknownAdviceTypeException(advisor.getAdvice());
76          }
77          return interceptors.toArray(new MethodInterceptor[interceptors.size()]);
78      }
79
80      @Override
81      public void registerAdvisorAdapter(AdvisorAdapter adapter) {
82          this.adapters.add(adapter);
```

```
    }

}
```

DefaultAdvisorAdapterRegistry 设置了一系列的配置，正是这些适配器的实现，为Spring AOP提供了编织能力。下面以MethodBeforeAdviceAdapter为例，看具体的实现：

```
class MethodBeforeAdviceAdapter implements AdvisorAdapter, Serializable {

    @Override
    public boolean supportsAdvice(Advice advice) {
        return (advice instanceof MethodBeforeAdvice);
    }

    @Override
    public MethodInterceptor getInterceptor(Advisor advisor) {
        MethodBeforeAdvice advice = (MethodBeforeAdvice) advisor.getAdvice();
        return new MethodBeforeAdviceInterceptor(advice);
    }

}
```

Spring AOP为了实现advice的织入，设计了特定的拦截器对这些功能进行了封装。我们接着看MethodBeforeAdviceInterceptor如何完成封装的？

```
public class MethodBeforeAdviceInterceptor implements MethodInterceptor, Serializable {

    private MethodBeforeAdvice advice;


    /**
     * Create a new MethodBeforeAdviceInterceptor for the given advice.
     * @param advice the MethodBeforeAdvice to wrap
     */
    public MethodBeforeAdviceInterceptor(MethodBeforeAdvice advice) {
        Assert.notNull(advice, "Advice must not be null");
        this.advice = advice;
    }

    @Override
```

```
    public Object invoke(MethodInvocation mi) throws Throwable {
        this.advice.before(mi.getMethod(), mi.getArguments(), mi.getThis() );
        return mi.proceed();
    }

}
```

可以看到，invoke方法中，首先触发了advice的before回调，然后才是proceed。

AfterReturningAdviceInterceptor的源码：

```
public class AfterReturningAdviceInterceptor implements MethodInterceptor, AfterAdvice,

    private final AfterReturningAdvice advice;


    /**
     * Create a new AfterReturningAdviceInterceptor for the given advice.
     * @param advice the AfterReturningAdvice to wrap
     */
    public AfterReturningAdviceInterceptor(AfterReturningAdvice advice) {
        Assert.notNull(advice, "Advice must not be null");
        this.advice = advice;
    }

    @Override
    public Object invoke(MethodInvocation mi) throws Throwable {
        Object retVal = mi.proceed();
        this.advice.afterReturning(retVal, mi.getMethod(), mi.getArguments(), mi.getThi
        return retVal;
    }

}
```

ThrowsAdviceInterceptor的源码：

```
    @Override
    public Object invoke(MethodInvocation mi) throws Throwable {
        try {
            return mi.proceed();
```

```
        }
        catch (Throwable ex) {
            Method handlerMethod = getExceptionHandler(ex);
            if (handlerMethod != null) {
                invokeHandlerMethod(mi, ex, handlerMethod);
            }
            throw ex;
        }
    }

    private void invokeHandlerMethod(MethodInvocation mi, Throwable ex, Method method)
        Object[] handlerArgs;
        if (method.getParameterCount() == 1) {
            handlerArgs = new Object[] { ex };
        }
        else {
            handlerArgs = new Object[] {mi.getMethod(), mi.getArguments(), mi.getThis()
        }
        try {
            method.invoke(this.throwsAdvice, handlerArgs);
        }
        catch (InvocationTargetException targetEx) {
            throw targetEx.getTargetException();
        }
    }
```

至此，我们知道了对目标对象的增强是通过拦截器实现的，最后还是上时序图：