Spring框架的AOP

**1. 增强整个类，未设置切点**

**1.1 设置增强处理器**

**1.1.1 MethodBeforeAdvice: 原方法被调用前增强**

*package* com.springAOP.beforeRun;

//import  
*import* org.springframework.aop.MethodBeforeAdvice;  
*import* java.lang.reflect.Method;

//必须实现下面接口  
*public class* MiPhBeforeRun *implements* MethodBeforeAdvice {  
 */\*私有域\*/* String *say*; */\*setter 方法\*/  
 public void* setSay(String say) {  
 *this*.*say* = say;  
 } *@Override  
 public void* before(Method method, Object[] objects, Object o) *throws* Throwable {  
 System.out.println(*say*);

}  
}

**注：method**参数是原方法，**objects**参数是原方法的参数列表，**o**参数是原方法所在的对象

**1.1.2 AfterReturningAdvice: 原方法返回值之后增强**

*package* com.springAOP.afterReturn;

*import* org.springframework.aop.AfterReturningAdvice;  
*import* java.lang.reflect.Method;  
*public class* MiPhAfterReturn *implements* AfterReturningAdvice {  
 */\*私有域\*/  
 private* String *say*;  
 */\*setter 方法\*/  
 public void* setSay(String say) {  
 *this*.*say* = say;  
 } *@Override  
 public void* afterReturning(Object o, Method method, Object[] objects, Object o1) *throws* Throwable {  
 System.out.println(say + “\t" + o1);  
 }  
}

**注：o**表示原方法的返回值，**method**表示原方法， **objects**表示原方法的参数列表， **o1**表示原方法所在的对象。

**1.1.3 ThrowAdvice: 原方法抛出异常之后增强**

*package* com.springAOP.afterThrow;

*import* org.springframework.aop.ThrowsAdvice;  
*public class* MiAfterThrow *implements* ThrowsAdvice {  
 *public void* otherException(ClassNotFoundException e) *throws* Throwable{  
 System.out.println("OtherException happened");  
 }  
  
 */\*通过映射定位这个方法，所以，方法名一定不能改变\*/  
 public void* afterThrowing(ClassNotFoundException e) *throws* Throwable {  
 System.out.println("IOException happened!");  
 }  
}

**注：**当被代理的对象的方法调用抛出异常之后，这个**类的名称为afterThrowing的方法**会被触发，注意这个方法**不是继承的方法，可以任意定义，唯一的要求就是方法名称正确**！原方法抛出的异常类型只要和这个方法参数的异常类型匹配，这个方法才会被触发调用！

**1.1.4: MethodInterceptor: 完全拦截原方法**

*package* com.springAOP.aroundAdvice;

*import* org.aopalliance.intercept.MethodInterceptor;  
*import* org.aopalliance.intercept.MethodInvocation;  
  
*public class* MiPhAroundMethod *implements* MethodInterceptor {  
 *@Override  
 public* Object invoke(MethodInvocation methodInvocation) *throws* Throwable {

//类似于BeforeMethodAdvice  
 System.out.println("Around: BeforeMethod");  
 //原方法被调用，可选操作，同时你还可以设置异常抛出并捕获实现类似ThrowAdvice  
 Object result = methodInvocation.proceed();  
 //原方法调用之后，相当于AfterReturningAdvice  
 System.out.println("Around: AfterReturn");  
 *return* result;  
 }  
}

注：需要注意此接口的名称。同时必须使用methodInvocation.proceed();语句实现方法回调；

**———————————————————————————————————————————————————————**

**1.2 配置增强处理器：**

**通过bean在xml中配置，于以其他类一般，如：**

*<!-- 增强 -->*

//配置上面自定义的增强”处理器”：即对业务具体的控制细节<bean *id*="mi\_BeforeRun" *class*="com.springAOP.beforeRun.MiPhBeforeRun"/>

**———————————————————————————————————————————————————————**

**1.3配置代理，连接增强处理：**

<bean *id*="proxy\_mi\_BeforeRun" *class*="org.springframework.aop.framework.ProxyFactoryBean">

<property *name*="target" *ref*="mi8\_MiPhones"/>  
 <property *name*="interceptorNames">  
 <list>  
 <value>mi\_BeforeRun</value>  
 </list>  
 </property>  
</bean>

注：*class*="org.springframework.aop.framework.ProxyFactoryBean"表示所使用的代理对象。

第一个属性指定元对象，即需要被增强的对象。

第二个属性指定需要为原对象添加的一些增强处理器。

通过这个代理类，可以获得原对象的接口，具体指定哪个接口，可以在getBean()方法中指明。

注：这种一个ProxyFactoryBean只能为一个bean资源增强

**2. 为上面的增强设置切点，实现实用的代理**

**所谓切点就是对寻在指定类中的某些方法进行增强，要指定当然是需要指定的条件，而这个条件就是所谓的切点.**

**2.1: 通过方法名称映射指定切点：你可以只需要修改xml配置资源之间的关系，就在上面原有的增强和原有class的基础上：**

**2.1.1: 指定切点pointCut：即方法名称**

*<!--PointCut\_ByName-->*

<bean *id*="open\_PointCut" *class*="org.springframework.aop.support.NameMatchMethodPointcut">  
 <property *name*="mappedName" *value*="open"/>  
</bean>

注：上面除了id属性和value属性值自定义外，一般其他的可以不用更改。

open是方法的名称。

**2.1.2：连接增强与切点：advisor连接点，让增强只针对特定的切点：组成连接点**

*<!--Advisor-->*

<bean *id*="open\_Advisor" *class*="org.springframework.aop.support.DefaultPointcutAdvisor">  
 <property *name*="pointcut" *ref*="open\_PointCut"/>  
 <property *name*="advice" *ref*="open\_BeforeMethodAdvice"/>  
</bean>

注：可以将上面两步2.1.1 和 2.1.2结合在一起配置：如下

<bean *id*="open\_Advisor" *class*="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">  
 <property *name*="mappedName" *value*="close"/>  
 <property *name*="advice" *ref*="open\_BeforeMethodAdvice"/>  
</bean>

**2.1.3：将连接点传给代理**

*<!--切点代理器-->*

<bean *id*="pCProxy\_miBeforeRun" *class*="org.springframework.aop.framework.ProxyFactoryBean">  
 <property *name*="target" *ref*="mi8\_MiPhones"/>  
 <property *name*="interceptorNames">  
 <list><value>open\_Advisor</value></list>  
 </property>  
</bean>

**3.通过**BeanNameAutoProxyCreator **自动创建proxy**

<bean *id*="auto\_proxy" *class*="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">

<property *name*="beanNames">  
 <list>  
 <value>mi8\_MiPhones</value>  
 </list>  
 </property>

<property *name*="interceptorNames">  
 <list>  
 <value>open\_Advisor</value>  
 </list>  
 </property>  
</bean>

注：<property *name*="beanNames">和<property *name*="interceptorNames">一般是不用更改的，他们分别针对特定的资源；

beanNames: 指的是将要增强的类，即对应的值都是表明这些bean要由这个代理自动代理

interceptorNames：指的是连接点

注：自动代理有两个好处

其一：能够同时为多个bean（资源）进行一致的代理。原因是默认代理使用这种<property *name*="target" *ref*="mi8\_MiPhones"/>方式指定需要代理的资源，而自动代理使用

<property *name*="beanNames"><list><value>mi8\_MiPhones</value></list></property>可以同时通过<value>指定多个资源。

其二：可以直接通过资源本身的bean的id属性读取这个资源

**4.通过**DefaultAdvisorAutoProxyCreator**自动创建proxy**

**对于含有Advisor中连接点指定的方法名称的所有bean资源，哪些方法都会被自动代理。**

<bean *id*="auto\_proxy" *class*="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

**注**：**对于所有的Advisor上指定的方法名称,如果某个bean含有这样的方法，就会自动创建代理！**

**4.通过AspectJ框架自动实现Spring AOP**

**注解：**

**1. @Aspect：被注解的类被定义为AspectJ框架**

**2. @Before：切点指定的方法被执行前，这个方法先被执行**

**3. @AfterReturning：切点指定的方法返回之后，这个方法被执行**

**4. @AfterThrowing：切点指定的方法抛出异常之后，这个方法被执行**

**5. @Around：拦截切点指定的方法**

**6. @PointCut：声明切点标签，以便通过将切点和增强方法解耦合**

**spring项目中需要引入的jar包为：**

**4.1 java框架**

*package* com.SpringAOP\_AspectJ.aspectJ;

*import* org.aspectj.lang.JoinPoint;  
*import* org.aspectj.lang.ProceedingJoinPoint;  
*import* org.aspectj.lang.annotation.\*;  
*import* java.util.Arrays;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
*@Aspect  
public class* PhoneAspectJ {  
 *@Before*("execution(public \* com.phone.Phone.runProgram(String))")  
 *public void* logBefore(JoinPoint joinPoint) {  
 System.out.println("\nAspectJ--BeforeMethodAdvice");  
 }

**注：**@before注解类似于BeforeMethodAdvice接口中的方法，括号中的值其中：public \* com.phone.Phone.runProgram(String)) 表示切点，即对类：com.phone.Phone名中名称为：runProgram，参数类型为：String返回类型为：任意类型的方法设置增强（即代理）

即类似于pointCut,这个完整的@before注解类似于advisor。下面以此类推。

首先，注解的方法是任意的，即其定义是没有限制的，只需要注意，这个方法的形参最多只能有一个JoinPoint类型的形参，而这个形参是对原方法的而外控制。即他包装了原方法的一些信息。当然，这个@before注解有不然没有参数，不然只有一个JoinPoint类型的参数。例如下面没有参数  
 *@Before*("execution(public \* com.phone.Phone.close())")  
 *public void* log1Before() {  
 System.out.println("Bey...");  
 }

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 *@AfterReturning*("execution(public \* com.phone.Phone.runProgram()", returning = "result")  
 *public void* logAfter(JoinPoint joinPoint, String[] result) *throws* Throwable {  
 System.out.println("After：" + Arrays.*toString*(result));  
 }

**注：**这个注解与@Before不同的是。它可以比@Before多一个类型的参数，如上面：result，首先此参数的名称必须在注解时returning = "result"，然后在方法形参上声明名称一样的参数，此时定义的形参类型就是切点方法的返回类型或者该返回类型会强制转化此定义的类型。当然这个形参的值便是切点方法的返回值了。

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 *@Around*("execution(public \* com.phone.Phone.runProgram()")  
 *public* String[] logAround(ProceedingJoinPoint joinPoint) *throws* Throwable {  
 System.out.println("Around:");  
 String[] strings = *new String*[]{"jjf1", "jj2"};  
 *//return (String[]) joinPoint.proceed(joinPoint.getArgs());  
 return* strings;  
 }

**注：**由于@Around注解是拦截整个方法，因此切点方法是否被真正执行必须得靠这个注解的方法控制，所以如果想对切点方法更多的控制或者回调原方法，必须在定义这个注解方法声明类型为ProceedingJoinPoint的参数，可以通过下面两种方式实现回调：

joinPoint.proceed();  
joinPoint.proceed(*new String*[]{"微信"});

第一种是按照默认的方式回调，即按照切点方法的原参数回调切点方法，第二种是按照给定的参数调用切点方法，这种方式的方法形式参数为：

Object[]  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 *@AfterThrowing*(""execution(public \* com.phone.Phone.throwException(..))")  
 *public void* afterThrow(JoinPoint joinPoint) {  
 System.out.println("afterThrow:" + joinPoint.getSignature().getName());  
 }  
}

*@AfterThrowing*(value = "execution(public \* com.phone.Phone.throwException(..))", throwing = "mythrow")  
*public void* afterThrow(JoinPoint joinPoint, ClassNotFoundException mythrow) {  
 System.out.println("afterThrow:" + joinPoint.getSignature().getName());  
}

**注：**第一个注解，其没有指定throwing,所以也不用在方法形参上定义某个异常类型相对应的变量，所以这种方式是对所有类型的异常都进行抛出后的增强，而第二个注解，指定了 throwing 和形参上类型相同的变量mythrow，其作用是当切点方法抛出mythrow变量变量指定的方法时，才会被增强.

**4.2 在xml文件上的配置**

**使用上面的AspectJ框架，需要在xml中如下配置。**

*<!--声明启动AspectJ框架功能。以便能够扫描工程项目中的所有的 @AspectJ 注解-->*

<aop:aspectj-autoproxy/>

*<!--需要增强的资源 配置-->*<bean *id*="mi\_Phones" *class*="com.phone.MiPhones">  
 <property *name*="model" *value*="Mi-9 CPU-855 storage-128G"/>  
</bean>  
  
*<!--AspectJ框架 配置-->*<bean *id*="phoneAspectJ" *class*="com.SpringAOP\_AspectJ.aspectJ.PhoneAspectJ"/>  
<bean *id*="phoneAspectJ1" *class*="com.SpringAOP\_AspectJ.aspectJ.PhoneAspectJ1"/>

注：完成上面即能实现所需业务

**总结：**

Spring AOP 框架能够将业务解耦合，将不同模块的功能解耦和，如日志系统，性能控制，类似于事件监听，但比事件监听程序更加简洁，更加完善。

**需要注意的是，不是一旦切点方法被增强之后，增强的方法就会触发，例如，如果类内部之间的方法调用即使方法之间都有增强，也不会被增强触发，必须有经过代理的包装之后，通过特定的方式调用才能触发如：接口。但这其实更加复杂，这关系到你选择什么样的模式建立代理。**

**需要注意环绕增强Around的使用，可以认为其优先级最高，一旦方法被调用，就会被所有Around类型的增强拦截，即所有的Before增强或者After增强目前都没有机会织入，只有等到Around增强的方法回调切点方法，这些增强才一定会被织入，如果Around拦截器没有回调，则这些增强也一定不会被织入。可以为一个切点设置任意多的增强，但程序的正确性不能依赖于这些增强的执行顺序。**