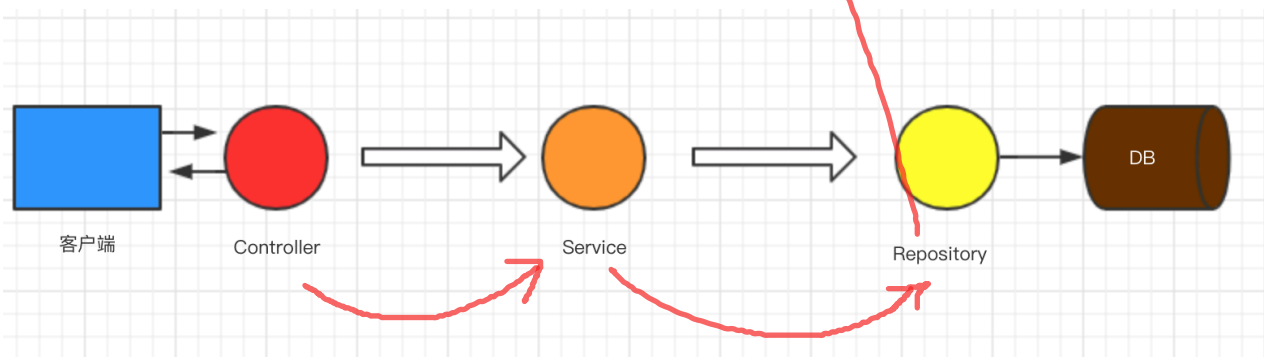
基于注解我们有三点要注意：第一要扫包，第二添加component注解，注意不能给接口加要给接口的实现类去加，第三点相关关联通过autowire来实现

实际开发的使用

实际开发中我们会将程序分为三层：

- Controller
- Service
- Repository（DAO）

关系 Controller --》Service --〉Repository



@Component 注解是将标注的类加载到 IoC 容器中，实际开发中可以根据业务需求分别使用 @Controller、@Service、@Repository 注解来标注控制层类、业务层类、持久层类。

Controller

都可以使用component但是使用这三个语意更加清楚

```
package com.southwind.controller;

import com.southwind.service.MyService;
import lombok.Setter;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;

@Setter
@Controller
public class MyController {

    @Autowired
    private MyService myService;

    /**
     * 模拟客户端请求
     */
    public String service(Double score){
        return myService.doService(score);
    }
}
```

Lombok的setter注解，因为mycontroller里面又myservice 当给myservice赋值时候这里没有setter就给myservice赋不了值

Service

```java
package com.southwind.service;

public interface MyService {
    public String doService(Double score);
}

package com.southwind.service.impl;

import com.southwind.repository.MyRepository;
import com.southwind.service.MyService;
import lombok.Setter;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Setter
@Service
public class MyServiceImpl implements MyService {

    @Autowired
    private MyRepository myRepository;

    @Override
    public String doService(Double score) {
        return myRepository.doRepository(score);
    }
}
```

Repository

```java
package com.southwind.repository;

public interface MyRepository {
    public String doRepository(Double score);
}

package com.southwind.repository.impl;

import com.southwind.repository.MyRepository;
import org.springframework.stereotype.Repository;

@Repository
public class MyRepositoryImpl implements MyRepository {
    @Override
    public String doRepository(Double score) {
        String result = "";
        if(score < 60){
            result = "不及格";
        }
        if(score >= 60 && score < 80){
```

```
            result = "合格";
        }
        if(score >= 80){
            result = "优秀";
        }
        return result;
    }
}
```

spring.xml

```
<context:component-scan base-package="com.southwind"></context:component-scan>
```

Spring IoC 底层实现

核心技术点：XML 解析 + 反射

具体的思路：

1、根据需求编写 XML 文件，配置需要创建的 bean。

2、编写程序读取 XML 文件，获取 bean 相关信息，类、属性、id。

3、根据第 2 步获取到的信息，结合反射机制动态创建对象，同时完成属性的赋值。

4、将创建好的 bean 存入 Map 集合，设置 key - value 映射，key 就是 bean 中 id 值，value 就是 bean 对象。

5、提供方法从 Map 中通过 id 获取到对应的 value。

```
package com.southwind.ioc;

import org.dom4j.Document;
import org.dom4j.DocumentException;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.NoSuchBeanDefinitionException;
import org.springframework.beans.factory.ObjectProvider;
import org.springframework.beans.factory.config.AutowireCapableBeanFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.MessageSourceResolvable;
import org.springframework.context.NoSuchMessageException;
import org.springframework.core.ResolvableType;
import org.springframework.core.env.Environment;
import org.springframework.core.io.Resource;

import java.io.IOException;
```

```
import java.lang.annotation.Annotation;
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Locale;
import java.util.Map;


public class MyClassPathXmlApplicationContext implements ApplicationContext {

    private Map<String,Object> iocMap;

    public MyClassPathXmlApplicationContext(String path) {
        iocMap = new HashMap<>();
        //解析 XML
        parseXML(path);
    }

    public void parseXML(String path){
        SAXReader saxReader = new SAXReader();
        try {
            Document document = saxReader.read("src/main/resources/"+path);
            Element root = document.getRootElement();
            Iterator<Element> rootIter = root.elementIterator();
            while(rootIter.hasNext()){
                Element bean = rootIter.next();
                String idStr = bean.attributeValue("id");
                String className = bean.attributeValue("class");
                //反射动态创建对象
                Class clazz = Class.forName(className);
                Constructor constructor = clazz.getConstructor();
                Object object = constructor.newInstance();
                //给属性赋值
                Iterator<Element> beanIter = bean.elementIterator();
                while(beanIter.hasNext()){
                    Element property = beanIter.next();
                    String propertyName = property.attributeValue("name");
                    String propertyValue = property.attributeValue("value");
                    //获取setter方法
                    //num-setNum,brand-setBrand
                    String methodName =
"set"+propertyName.substring(0,1).toUpperCase()+propertyName.substring(1);
                    //获取属性
                    Field field = clazz.getDeclaredField(propertyName);
                    Method method =
clazz.getMethod(methodName,field.getType());
```

```
                    Object value = propertyValue;
                    //类型转换
                    switch (field.getType().getName()){
                        case "java.lang.Integer":
                            value = Integer.parseInt(propertyValue);
                            break;
                        case "java.lang.Double":
                            value = Double.parseDouble(propertyValue);
                            break;
                    }
                    //调用方法
                    method.invoke(object,value);
                }
                //存入 Map
                iocMap.put(idStr,object);
            }
        } catch (DocumentException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e){
            e.printStackTrace();
        } catch (NoSuchMethodException e){
            e.printStackTrace();
        } catch (InstantiationException e){
            e.printStackTrace();
        } catch (IllegalAccessException e){
            e.printStackTrace();
        } catch (InvocationTargetException e){
            e.printStackTrace();
        } catch (NoSuchFieldException e){
            e.printStackTrace();
        }

    }

    @Override
    public Object getBean(String s) throws BeansException {
        return iocMap.get(s);
    }
}
```

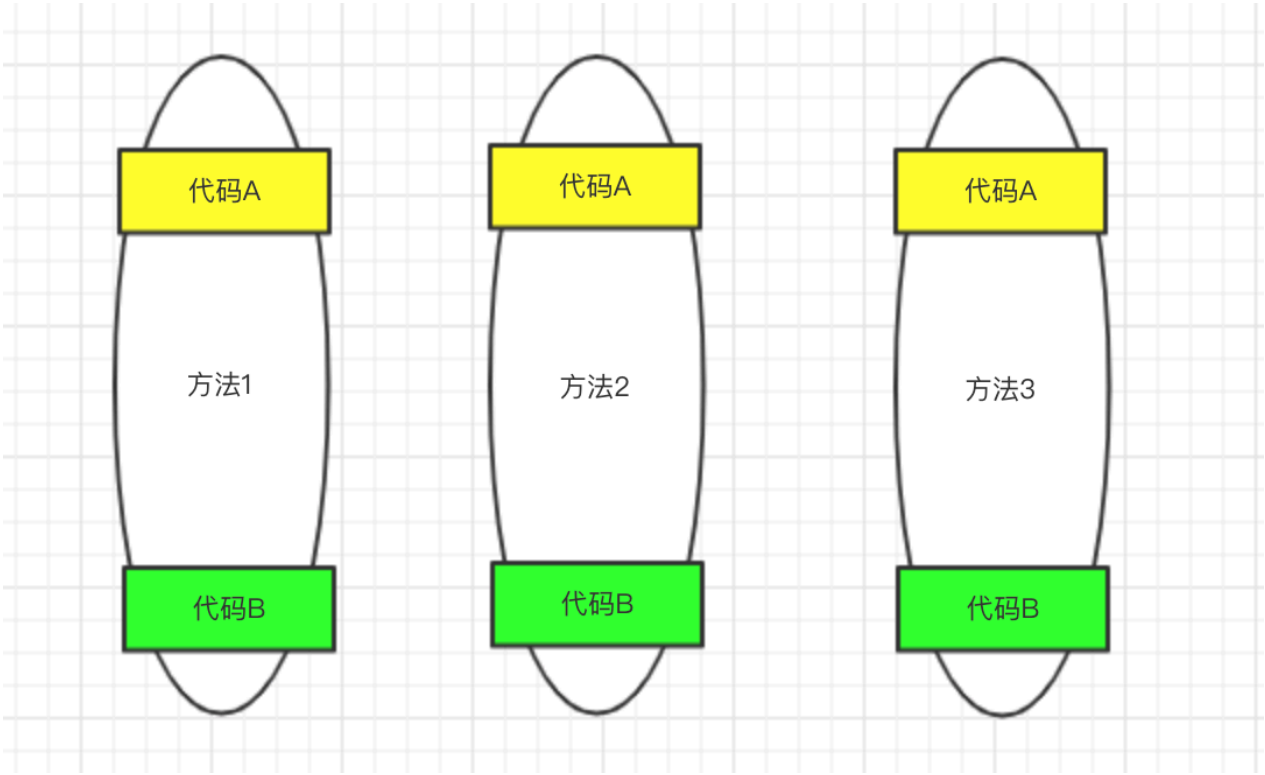
Spring AOP

AOP（Aspect Oriented Programming）面向切面编程。

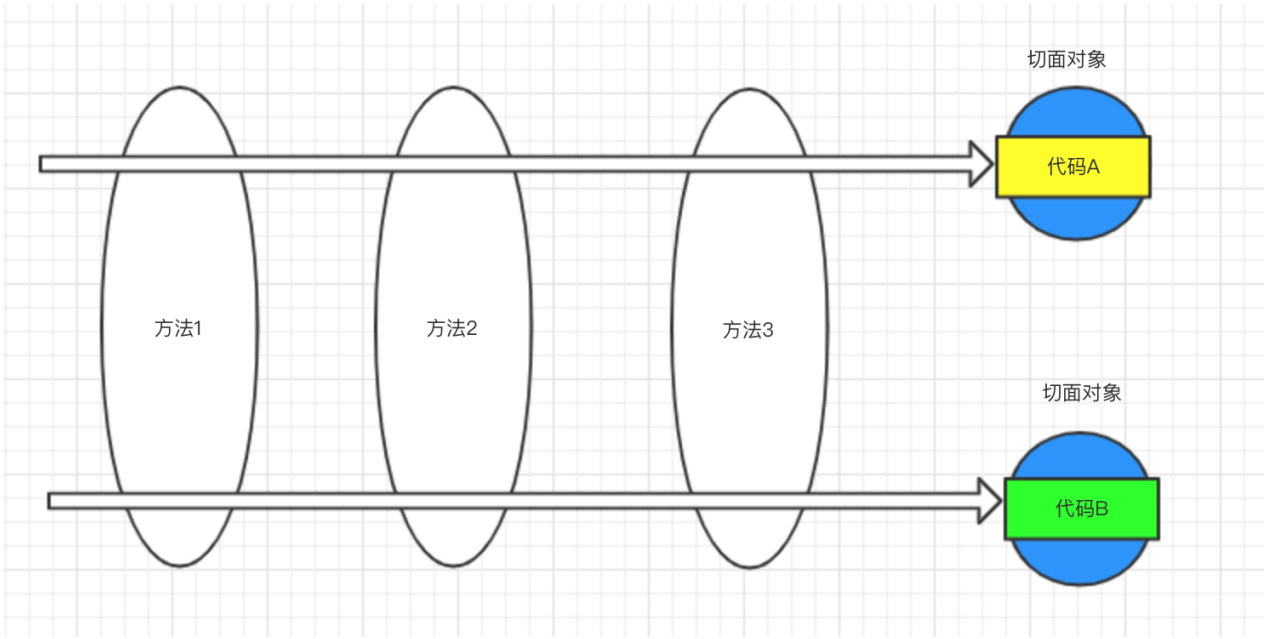
OOP（Object Oriented Programming）面向对象编程，用对象化的思想来完成程序。

AOP 是对 OOP 的一个补充，是在另外一个维度上抽象出对象。

具体是指程序运行时动态地将非业务代码切入到业务代码中，从而实现程序的解耦合，将非业务代码抽象成一个对象，对对象编程就是面向切面编程。



上述形式的代码维护性差，没有代码复用性，使用 AOP 进行优化，如下图所示。



AOP 的优点：

- 可以降低模块之间的耦合性
- 提高代码的复用性
- 提高代码的维护性
- 集中管理非业务代码，便于维护
- 业务代码不受非业务代码的影响，逻辑更加清晰

通过一个例子来理解 AOP。

1、创建一个计算器接口 Cal

```
package com.southwind.aop;

public interface Cal {
    public int add(int num1,int num2);
    public int sub(int num1,int num2);
    public int mul(int num1,int num2);
    public int div(int num1,int num2);
}
```

2、创建接口的实现类 CalImpl

```
package com.southwind.aop.impl;

import com.southwind.aop.Cal;

public class CalImpl implements Cal {
    @Override
    public int add(int num1, int num2) {
        int result = num1 + num2;
        return result;
    }

    @Override
    public int sub(int num1, int num2) {
        int result = num1 - num2;
        return result;
    }

    @Override
    public int mul(int num1, int num2) {
        int result = num1 * num2;
        return result;
    }

    @Override
    public int div(int num1, int num2) {
        int result = num1 / num2;
        return result;
    }
}
```

日志打印

- 在每个方法开始位置输出参数信息。
- 在每个方法结束位置输出结果信息。

AOP是一种编程思想不是一个具体的技术我们需要用具体技术实现它

对于计算器来讲，加减乘除就是业务代码，日志打印就是非业务代码。

AOP 如何实现？使用动态代理的方式来实现。

代理首先应该具备 CalImpl 的所有功能，并在此基础上，扩展出打印日志的功能。

1、删除 CalImpl 方法中所有打印日志的代码，只保留业务代码。

程序运行才创建的类

2、创建 MyInvocationHandler 类，实现 InvocationHandler 接口，生成动态代理类。

动态代理类，需要动态生成，需要获取到委托类的接口信息，根据这些接口信息动态生成一个代理类，然后再由 ClassLoader 将动态生成的代理类加载到 JVM。

要加载前提是要有这个类，那么就需要动态创建代理类，代理需要和委托一样的功能，功能看接口，代理类就需要实现和委托类一样的接口就需要获取委托类的接口

```
package com.southwind.aop;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.Arrays;

public class MyInvocationHandler implements InvocationHandler {
    //委托对象
    private Object object = null;

    //返回代理对象
    public Object bind(Object object){
        this.object = object;
        return Proxy.newProxyInstance(
                object.getClass().getClassLoader(),
                object.getClass().getInterfaces(),
                this);
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        //实现业务代码和非业务代码的解耦合
        System.out.println(method.getName()+"方法的参数是"+
Arrays.toString(args));
        Object result = method.invoke(this.object,args);
        System.out.println(method.getName()+"方法的结果是"+ result);
        return result;
    }
}
```

程序通过InvocationHandler来创建动态代理类

完成业务代码

返回代理对象

类加载器

接口

```
package com.southwind.aop;

import com.southwind.aop.impl.CalImpl;

public class Test {
```

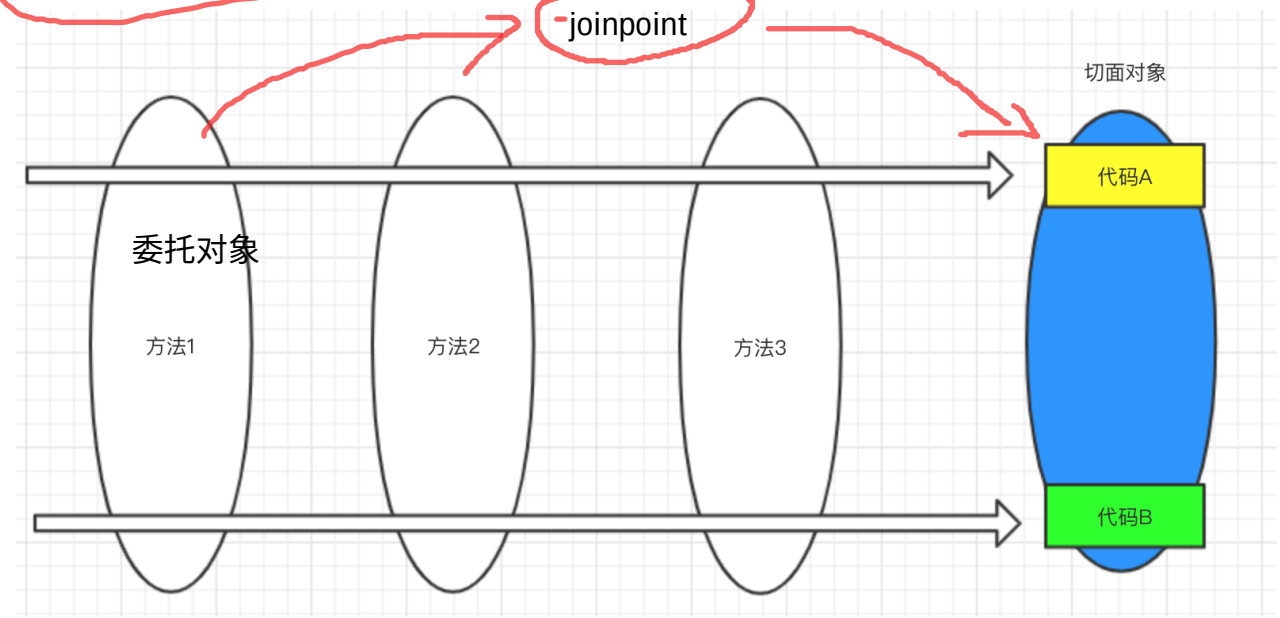
```
    public static void main(String[] args) {
        //实例化委托对象
        Cal cal = new CalImpl();
        //获取代理对象
        MyInvocationHandler myInvocationHandler = new MyInvocationHandler();
        Cal proxy = (Cal) myInvocationHandler.bind(cal);
        proxy.add(10,3);
        proxy.sub(10,3);
        proxy.mul(10,3);
        proxy.div(10,3);
    }
}
```

上述代码通过动态代理机制实现了业务代码和非业务代码的解耦合，这是 Spring AOP 的底层实现机制，真正在使用 Spring AOP 进行开发时，不需要这么复杂，可以用更好理解的方式来完成开发。

Spring AOP 的开发步骤

上面是实现原理

joinpoint



1、创建切面类。

```
package com.southwind.aop;

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

import java.util.Arrays;

@Component
@Aspect
public class LoggerAspect {

    @Before("execution(public int com.southwind.aop.impl.CalImpl.*(..))")
    public void before(JoinPoint joinPoint){
```

maven中添加aop依赖，aspect
aspect注解才生效

这个注解一加表示切面的方法会在add方法之前执行

切面和业务代码通过位置来关联所以在方法执行前操作

joinpoint连接委托对象方法和切面对象方法，通过这个可以拿到委托对象方法名和参数列表

```
        String name = joinPoint.getSignature().getName();
        String args = Arrays.toString(joinPoint.getArgs());
        System.out.println(name+"方法的参数是"+args);
    }

    @After("execution(public int com.southwind.aop.impl.CalImpl.*(..))")
    public void after(JoinPoint joinPoint){
        String name = joinPoint.getSignature().getName();
        System.out.println(name+"方法执行完毕");
    }

    @AfterReturning(value = "execution(public int
com.southwind.aop.impl.CalImpl.*(..))",returning = "result")
    public void afterReturn(JoinPoint joinPoint,Object result){
        String name = joinPoint.getSignature().getName();
        System.out.println(name+"方法的结果是"+result);
    }

    @AfterThrowing(value = "execution(public int
com.southwind.aop.impl.CalImpl.*(..))",throwing = "ex")
    public void afterThrowing(JoinPoint joinPoint,Exception ex){
        String name = joinPoint.getSignature().getName();
        System.out.println(name+"方法抛出异常"+ex);
    }
}
```

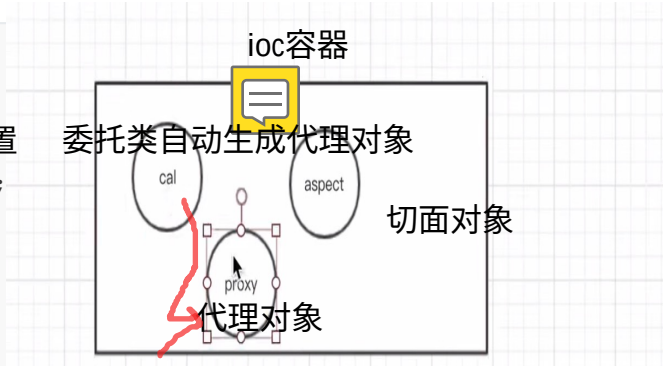
- @Component，将切面类加载到 IoC 容器中。
- @Aspect，表示该类是一个切面类。
- @Before，表示方法的执行时机是在业务方法之前，execution 表达式表示切入点是 CalImpl 类中的 add 方法。
- @After，表示方法的执行时机是在业务方法结束之后，execution 表达式表示切入点是 CalImpl 类中的 add 方法。
- @AfterReturning，表示方法的执行时机是在业务方法返回结果之后，execution 表达式表示切入点是 CalImpl 类中的 add 方法，returning 是将业务方法的返回值与切面类方法的形参进行绑定。
- @AfterThrowing，表示方法的执行时机是在业务方法抛出异常之后，execution 表达式表示切入点是 CalImpl 类中的 add 方法，throwing 是将业务方法的异常与切面类方法的形参进行绑定。

2、委托类也需要添加 @Component

```
package com.southwind.aop.impl;

import com.southwind.aop.Cal;
import org.springframework.stereotype.Component;

@Component
public class CalImpl implements Cal {
    @Override
    public int add(int num1, int num2) {
        int result = num1 + num2;
```



业务代码写在委托对象，非业务代码写在切面对象

切面bean的作用是为委托对象生成一个代理对象，切面中定义了非业务代码以及什么时候去执行在什么位置执行，具体的实现是交给proxy，proxy再结合委托对象，实现业务代码与非业务代码解耦合，代理对象是动态生成的在spring.xml中

```
        return result;
    }

    @Override
    public int sub(int num1, int num2) {
        int result = num1 - num2;
        return result;
    }

    @Override
    public int mul(int num1, int num2) {
        int result = num1 * num2;
        return result;
    }

    @Override
    public int div(int num1, int num2) {
        int result = num1 / num2;
        return result;
    }
}
```

3、spring.xml　　既然是基于注解方式就需要配置个自动扫包

```
<?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"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-4.3.xsd
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
   http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd
">

    <!-- 自动扫包 -->
    <context:component-scan base-package="com.southwind.aop">
</context:component-scan>

    <!-- 为委托对象自动生成代理对象 -->
    <aop:aspectj-autoproxy></aop:aspectj-autoproxy>

</beans>
```

- aop:aspectj-autoproxy，Spring IoC 容器会结合切面对象和委托对象自动生成动态代理对象，AOP 底层就是通过动态代理机制来实现的。

AOP 的概念：

- 切面对象：根据切面抽象出来的对象，CalImpl 所有方法中需要加入日志的部分，抽象成一个切面类 LoggerAspect。
- 通知：切面对象具体执行的代码，即非业务代码，LoggerAspect 对象打印日志的代码。
- 目标：被横切的对象，即 CalImpl，将通知加入其中。
- 代理：切面对象、通知、目标混合之后的结果，即我们使用 JDK 动态代理机制创建的对象。
- 连接点：需要被横切的位置，即通知要插入业务代码的具体位置。

在Aop中，如果切入点匹配到方法，则aop会创建一个代理对象，来执行对应的切入点方法，和执行匹配到的方法。 如果匹配不成功的话，就不会创建代理对象，而没有匹配到的这个方法就由它本身所在的类创建对象（目标对象）去执行这个方法。