AOP

AOP是Spring框架除了IoC之外的另外一个核心机制，AOP：Aspect Oriented Programming，意为面向切面编程。Java是面向对象编程（OOP），指将所有的一切都看做是对象，通过对象与对象之间的相互作用关系来完成业务逻辑的一种方式。

AOP是对OOP的一个补充，在程序运行时，动态地将代码切入到类的指定方法、指定位置上的编程思想，将不同的方法的同一个位置抽象成一个切面对象，对该切面对象进行编程就是AOP。

AOP的优点：

- 降低模块之间的耦合度。
- 使系统更容易扩展。
- 更好的代码复用。
- 非业务代码更加集中，不分散，便于统一管理。
- 业务代码更加简洁纯粹，不参杂其他代码的影响。

如何实现AOP?

使用动态代理的方式来实现。

我们希望CalImpl只进行业务运算，不进行打印日志的工作，那么就需要有一个对象来替代CalImpl进行打印日志的工作，这就是代理对象。

代理对象首先应该具备CalImpl的所有功能，并且在此基础上，扩展出打印日志的功能。

bind方法是MyInvocationHandler类提供给外部调用的方法，传入委托对象，bind方法会返回一个代理对象。

bind方法完成了两项工作：

- 将外部传进来的委托对象保存到成员变量中，因为业务方法调用时需要用到委托对象，其实还是让委托对象自己调用自己的方法，动态代理只是为其提供了可扩展的空间。
- 通过Proxy.newProxyInstance方法创建一个代理对象，newProxyInstance方法的参数：
    - 我们知道对象时JVM根据运行时类来创建的，此时需要动态创建一个代理对象，可以使用委托对象的运行时类来获取类加载器，然后将动态生成的代理类加载到内存中，通过该类来创建代理对象。
    - 同时代理对象需要具备委托对象的所有功能，即需要拥有委托对象的所有接口，所以传入委托对象的接口信息，object.getClass().getInterfaces()。
    - this指当前的MyInvocationHandler对象。
- invoke方法：method对象指代委托对象的所有方法，args用来表示委托对象方法的参数信息。method.invoke(this.object,args)是通过反射机制来调用委托对象的业务方法。

所以在method.invoke(this.object,args)代码的前后添加打印日志的信息，就等于在委托对象的业务方法前后添加打印日志的信息，并且完成了分类，业务方法在委托对象中，打印日志信息在代理对象中。

Cal

```
package com.southwind.util;
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);
}
```

CalImpl

```
package com.southwind.util;
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;
    }
}
```

MyInvocationHandler

```
package com.southwind.util;
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 {
        System.out.println(method.getName()+"的参数是: "+
Arrays.toString(args));
        Object result = method.invoke(this.object,args);
        System.out.println(method.getName()+"的结果是: "+result);
        return result;
    }
}
```

Test

```
package com.southwind.test;
import com.southwind.util.Cal;
import com.southwind.util.CalImpl;
import com.southwind.util.MyInvocationHandler;
public class Test {
    public static void main(String[] args) {
        //创建委托对象
        Cal cal = new CalImpl();
        //获取动态代理对象
        MyInvocationHandler myInvocationHandler = new
MyInvocationHandler();
        Cal cal1 = (Cal) myInvocationHandler.bind(cal);
        cal1.add(10,3);
        cal1.sub(10,3);
        cal1.mul(10,3);
        cal1.div(10,3);
    }
}
```

以上就是通过动态代理实现AOP的过程，我们在使用Spring框架的AOP时，可以用更加简便、容易理解的方式来完成AOP，Spring框架对AOP的过程进行了封装，让开发者可以更加简便地使用AOP进行开发。

在Spring框架中，我们不需要创建动态代理类，只需要创建一个切面类，由该切面类产生的对象就是切面对象，可以将非业务代码写入到切面对象中，再切入到业务方法中，Spring框架底层会自动根据切面类以及目标类生成一个代理对象。

- 创建切面类 LoggerAspect

```
package com.southwind.aop;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.*;
import org.springframework.stereotype.Component;
import java.util.Arrays;
@Aspect
@Component
public class LoggerAspect {

    @Before("execution(public int com.southwind.util.CalImpl.*(..))")
    public void before(JoinPoint joinPoint){
        //获取方法名
        String name = joinPoint.getSignature().getName();
        //获取参数列表
        String args = Arrays.toString(joinPoint.getArgs());
        System.out.println(name+"的参数是：+args);
    }

    @After("execution(public int com.southwind.util.CalImpl.*(..))")
    public void after(JoinPoint joinPoint){
        //获取方法名
        String name = joinPoint.getSignature().getName();
        System.out.println(name+"方法结束");
    }

    @AfterReturning(value = "execution(public int
com.southwind.util.CalImpl.*(..))",returning = "result")
    public void afterReturning(JoinPoint joinPoint,Object result){
        //获取方法名
        String name = joinPoint.getSignature().getName();
        System.out.println(name+"方法的结果是：+result);
    }

    @AfterThrowing(value = "execution(public int
com.southwind.util.CalImpl.*(..))",throwing = "exception")
    public void afterThrowing(JoinPoint joinPoint,Exception exception){
        //获取方法名
        String name = joinPoint.getSignature().getName();
        System.out.println(name+"方法抛出异常：+exception);
    }
}
```

LoggerAspect类定义处添加了两个注解：

- @Aspect：表示该类是切面类。
- @Component：将该类注入到IoC容器中。

- @Before：表示切面方法执行的时机是业务方法执行之前。
- @After：表示切面方法执行的时机是业务方法执行之后。
- @AfterReturning：表示切面方法执行的时机是业务方法return之后。
- @AfterThrowing：表示切面方法执行的时机是业务方法抛出异常之后。
- execution(public int com.southwind.util.CalImpl.*(..))：表示切入点是com.southwind.util包下CalImpl类中的所有方法，即CalImpl所有方法在执行时会优先执行切面方法。

目标类也需要添加@Component注解

```
package com.southwind.util;
import org.springframework.stereotype.Component;
@Component
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;
    }
}
```

spring.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"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
    http://www.springframework.org/schema/aop
```

```
    http://www.springframework.org/schema/aop/spring-aop-4.0.xsd
    http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd
">

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

    <!-- 使Aspect注解生效，为委托类自动生成代理对象 -->
    <aop:aspectj-autoproxy></aop:aspectj-autoproxy>

</beans>
```

- 将com.southwind包下的所有添加了@Component注解的类扫描到IoC容器中。
- 添加aop:aspectj-autoproxy注解，Spring容器会结合切面类和目标类自动生成动态代理对象，Spring框架的AOP底层远离就是通过动态代理的方式完成面向切面编程。

切面：横切关注点被模块化的特殊对象。

CalImpl所有方法中需要加入日志的部分，抽象成一个切面对象LoggerAspect。

通知：切面对象完成的工作。

LoogerAspect对象打印日志的操作。

目标：被通知的对象，即被横切的对象。

CalImpl对象。

代理：切面、通知、目标混合之后的对象。

连接点：程序要执行的某个特定位置。

切面方法要插入业务方法的具体位置。

切点：AOP通过切点定位到连接点。