04Spring AOP介绍与使用

AOP：Aspect Oriented Programming 面向切面编程

OOP：Object Oriented Programming 面向对象编程

面向切面编程：基于OOP基础之上新的编程思想，OOP面向的主要对象是类，而AOP面向的主要对象是切面，在处理日志、安全管理、事务管理等方面有非常重要的作用。AOP是Spring中重要的核心点，虽然IOC容器没有依赖AOP，但是AOP提供了非常强大的功能，用来对IOC做补充。通俗点说的话就是在程序运行期间。

在不修改原有代码的情况下 增强**跟主要业务没有关系的公共功能代码**到 **之前写好的方法中的指定位置** 这种编程的方式叫AOP

1、AOP的概念

为什么要引入AOP?

```
1
2  @Service
3  public class RoleServiceImpl  implements RoleService {
4
5      public Role get(Integer id) {
6          System.out.println("查询Role");
7          return new Role();
8      }
9
10     public void add(Role role) {
11         System.out.println("添加Role");
12     }
13
14     public void delete(Integer id) {
15         // 日志
16         System.out.println("删除Role");
17
```

```
18      }
19
20      public void update(Role role) {
21          System.out.println("修改Role");
22      }
23  }
```

此代码非常简单，就是基础的三层CRUD的代码实现，此时如果需要添加日志功能应该怎么做呢，只能在每个方法中添加日志输出，同时如果需要修改的话会变得非常麻烦。

按照上述方式抽象之后，代码确实简单很多，但是大家应该已经发现在输出的信息中并不包含具体的方法名称，我们更多的是想要在程序运行过程中动态的获取方法的名称及参数、结果等相关信息，此时可以通过使用**代理**的方式来进行实现。

AOP的底层用的代理，代理是一种设计模式

- 静态代理

弊端：需要为每一个被代理的类创建一个“代理类”，虽然这种方式可以实现，但是成本太高

- 动态代理（AOP的底层是用的动态）
  - jdk动态代理： 必须保证被代理的类实现了接口，
  - cglib动态代理： 不需要接口，

CalculatorProxy.java

```
1
2   package cn.tulingxueyuan.proxy;
3
4   import cn.tulingxueyuan.inter.Calculator;
5
6   import java.lang.reflect.InvocationHandler;
7   import java.lang.reflect.InvocationTargetException;
```

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

/**
 * 帮助Calculator生成代理对象的类
 */
public class CalculatorProxy {

    /**
     *
     *  为传入的参数对象创建一个动态代理对象
     * @param calculator 被代理对象
     * @return
     */
    public static Calculator getProxy(final Calculator calculator){


        //被代理对象的类加载器
        ClassLoader loader = calculator.getClass().getClassLoader();
        //被代理对象的接口
        Class<?>[] interfaces = calculator.getClass().getInterfaces();
        //方法执行器，执行被代理对象的目标方法
        InvocationHandler h = new InvocationHandler() {
            /**
             *  执行目标方法
             * @param proxy 代理对象，给jdk使用，任何时候都不要操作此对象
             * @param method 当前将要执行的目标对象的方法
             * @param args 这个方法调用时外界传入的参数值
             * @return
```

```
38              * @throws Throwable
39              */
40             public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
41                 //利用反射执行目标方法,目标方法执行后的返回值
42 //                System.out.println("这是动态代理执行的方法");
43                 Object result = null;
44                 try {
45                     System.out.println(method.getName()+"方法开始执行，参数是："+ Arrays.asList(args));
46                     result = method.invoke(calculator, args);
47                     System.out.println(method.getName()+"方法执行完成，结果是："+ result);
48                 } catch (Exception e) {
49                     System.out.println(method.getName()+"方法出现异常："+ e.getMessage());
50                 } finally {
51                     System.out.println(method.getName()+"方法执行结束了.....");
52                 }
53                 //将结果返回回去
54                 return result;
55             }
56         };
57         Object proxy = Proxy.newProxyInstance(loader, interfaces, h);
58         return (Calculator) proxy;
59     }
60 }
```

我们可以看到这种方式更加灵活，而且不需要在业务方法中添加额外的代码，这才是常用的方式。如果想追求完美的同学，还可以使用上述的日志工具类来完善。

LogUtil.java

```
1 package cn.tulingxueyuan.util;
2
```

```
import java.lang.reflect.Method;
import java.util.Arrays;

public class LogUtil {

    public static void start(Method method, Object ... objects){
//        System.out.println("XXX方法开始执行，使用的参数是："+ Arrays.asList(objects));
        System.out.println(method.getName()+"方法开始执行，参数是："+ Arrays.asList(objects));
    }

    public static void stop(Method method,Object ... objects){
//        System.out.println("XXX方法执行结束，结果是："+ Arrays.asList(objects));
        System.out.println(method.getName()+"方法开始执行，参数是："+ Arrays.asList(objects));

    }

    public static void logException(Method method,Exception e){
        System.out.println(method.getName()+"方法出现异常："+ e.getMessage());
    }

    public static void end(Method method){
        System.out.println(method.getName()+"方法执行结束了.....");
    }
}
```

CalculatorProxy.java

```
package cn.tulingxueyuan.proxy;

import cn.tulingxueyuan.inter.Calculator;
import cn.tulingxueyuan.util.LogUtil;
```

```
5
6  import java.lang.reflect.InvocationHandler;
7  import java.lang.reflect.InvocationTargetException;
8  import java.lang.reflect.Method;
9  import java.lang.reflect.Proxy;
10 import java.util.Arrays;
11
12 /**
13  * 帮助Calculator生成代理对象的类
14  */
15 public class CalculatorProxy {
16
17     /**
18      *
19      *  为传入的参数对象创建一个动态代理对象
20      * @param calculator 被代理对象
21      * @return
22      */
23     public static Calculator getProxy(final Calculator calculator){
24
25
26         //被代理对象的类加载器
27         ClassLoader loader = calculator.getClass().getClassLoader();
28         //被代理对象的接口
29         Class<?>[] interfaces = calculator.getClass().getInterfaces();
30         //方法执行器，执行被代理对象的目标方法
31         InvocationHandler h = new InvocationHandler() {
32             /**
33              *  执行目标方法
34              * @param proxy 代理对象，给jdk使用，任何时候都不要操作此对象
```

```
             * @param method 当前将要执行的目标对象的方法
             * @param args 这个方法调用时外界传入的参数值
             * @return
             * @throws Throwable
             */
            public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                //利用反射执行目标方法,目标方法执行后的返回值
//                System.out.println("这是动态代理执行的方法");
                Object result = null;
                try {
                    LogUtil.start(method,args);
                    result = method.invoke(calculator, args);
                    LogUtil.stop(method,args);
                } catch (Exception e) {
                    LogUtil.logException(method,e);
                } finally {
                    LogUtil.end(method);
                }
                //将结果返回回去
                return result;
            }
        };
        Object proxy = Proxy.newProxyInstance(loader, interfaces, h);
        return (Calculator) proxy;
    }
}
```

很多同学看到上述代码之后可能感觉已经非常完美了，但是要说明的是，这种动态代理的实现方式调用的是jdk的基本实现，如果需要代理的目标对象没有实现任何接口，那么是无法为他创建代理对象的，这也是致命的缺陷。而在Spring中我们可以不编写上述如此复杂的代码，只需要利用AOP，就能够轻轻松松实现上述功能，当然，Spring AOP的底层实现也依赖的是动态代理。

AOP的核心概念及术语

- 切面（Aspect）：指关注点模块化，这个关注点可能会横切多个对象。事务管理是企业级Java应用中有关横切关注点的例子。 在Spring AOP中，切面可以使用通用类基于模式的方式（schema-based approach）或者在普通类中以`@Aspect`注解（@AspectJ 注解方式）来实现。
- 连接点（Join point）：在程序执行过程中某个特定的点，例如某个方法调用的时间点或者处理异常的时间点。在Spring AOP中，一个连接点总是代表一个方法的执行。
- 通知（Advice）：在切面的某个特定的连接点上执行的动作。通知有多种类型，包括“around”, “before” and “after”等等。通知的类型将在后面的章节进行讨论。 许多AOP框架，包括Spring在内，都是以拦截器做通知模型的，并维护着一个以连接点为中心的拦截器链。
- 切点（Pointcut）：匹配连接点的断言。通知和切点表达式相关联，并在满足这个切点的连接点上运行（例如，当执行某个特定名称的方法时）。切点表达式如何和连接点匹配是AOP的核心：Spring默认使用AspectJ切点语义。
- 引入（Introduction）：声明额外的方法或者某个类型的字段。Spring允许引入新的接口（以及一个对应的实现）到任何被通知的对象上。例如，可以使用引入来使bean实现 `IsModified`接口， 以便简化缓存机制（在AspectJ社区，引入也被称为内部类型声明（inter））。
- 目标对象（Target object）：被一个或者多个切面所通知的对象。也被称作被通知（advised）对象。既然Spring AOP是通过运行时代理实现的，那么这个对象永远是一个被代理（proxied）的对象。
- AOP代理（AOP proxy）:AOP框架创建的对象，用来实现切面契约（aspect contract）（包括通知方法执行等功能）。在Spring中，AOP代理可以是JDK动态代理或CGLIB代理。
- 织入（Weaving）：把切面连接到其它的应用程序类型或者对象上，并创建一个被被通知的对象的过程。这个过程可以在编译时（例如使用AspectJ编译器）、类加载时或运行时中完成。 Spring和其他纯Java AOP框架一样，是在运行时完成织入的。

AOP的通知类型

- 前置通知（Before advice）：在连接点之前运行但无法阻止执行流程进入连接点的通知（除非它引发异常）。
- 后置返回通知（After returning advice）:在连接点正常完成后执行的通知（例如，当方法没有抛出任何异常并正常返回时）。
- 后置异常通知（After throwing advice）：在方法抛出异常退出时执行的通知。
- 后置通知（总会执行）（After (finally) advice）：当连接点退出的时候执行的通知（无论是正常返回还是异常退出）。
- 环绕通知（Around Advice）:环绕连接点的通知，例如方法调用。这是最强大的一种通知类型，。环绕通知可以在方法调用前后完成自定义的行为。它可以选择是否继续执行连接点或直接返回自定义的返回值又或抛出异常将执行结束。

AOP的应用场景

- 日志管理
- 权限认证
- 安全检查
- 事务控制

2、Spring AOP的简单配置

在上述代码中我们是通过动态代理的方式实现日志功能的，但是比较麻烦，现在我们将要使用spring aop的功能实现此需求，其实通俗点说的话，就是把LogUtil的工具类换成另外一种实现方式。

1、在ioc的基础上添加pom依赖

```
1
2         <dependency>
3             <groupId>org.aspectj</groupId>
4             <artifactId>aspectjweaver</artifactId>
5             <version>1.9.5</version>
6         </dependency>
7
8         <dependency>
9             <groupId>org.springframework</groupId>
10            <artifactId>spring-aspects</artifactId>
11            <version>5.2.3.RELEASE</version>
```

2、编写配置

- 将目标类和切面类加入到IOC容器中，在对应的类上添加组件注解
  - 给LogUtil添加@Component注解
  - 给MyCalculator添加@Service注解
  - 添加自动扫描的配置

```
1 <!--别忘了添加context命名空间-->
2 <context:component-scan base-package="cn.tulingxueyuan"></context:component-scan>
```

- 设置程序中的切面类
  - 在LogUtil.java中添加@Aspect注解
- 设置切面类中的方法是什么时候在哪里执行
  - 在增强模块的类上面标记

- 声明为切面
- 将切面交给spring去管理

```
@Aspect
@Component
```

```
package cn.tulingxueyuan.util;

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

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

@Component
@Aspect
public class LogUtil {

    /*
    设置下面方法在什么时候运行
        @Before:在目标方法之前运行：前置通知
        @After:在目标方法之后运行：后置通知
        @AfterReturning:在目标方法正常返回之后：返回通知
        @AfterThrowing:在目标方法抛出异常后开始运行：异常通知
        @Around:环绕：环绕通知

        当编写完注解之后还需要设置在哪些方法上执行，使用表达式
        execution(访问修饰符  返回值类型  方法全称)
     */
```

```
24          // 前置通知
25          @Before("execution(* cn.tulingxueyuan.service..*.*(..))")
26          public static void before(){
27             /* System.out.println(method.getName()+"方法运行前，参数是"+
28                      (args==null?"": Arrays.asList(args).toString()));*/
29
30              System.out.println("方法前");
31          }
32
33          // 后置通知
34          @After("execution(* cn.tulingxueyuan.service..*.*(..))")
35          public static void after(){
36             /* System.out.println(method.getName() +"方法运行后，参数是"+
37                      (args==null?"": Arrays.asList(args).toString()));*/
38              System.out.println("方法后");
39          }
40
41          // 后置异常通知
42          @AfterThrowing("execution(* cn.tulingxueyuan.service..*.*(..))")
43          public static void afterException(){
44             // System.out.println("方法报错了:"+ex.getMessage());
45
46              System.out.println("方法异常");
47          }
48
49          // 后置返回通知
50          @AfterReturning("execution(* cn.tulingxueyuan.service..*.*(..))")
51          public static void afterEnd(){
52              //System.out.println("方法结束，返回值是:"+returnValue);
53              System.out.println("方法返回");
```

```
54
55          }
56  }
```

- 开启基于注解的aop的功能

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xmlns:context="http://www.springframework.org/schema/context"
5         xmlns:aop="http://www.springframework.org/schema/aop"
6         xsi:schemaLocation="http://www.springframework.org/schema/beans
7         http://www.springframework.org/schema/beans/spring-beans.xsd
8         http://www.springframework.org/schema/context
9         http://www.springframework.org/schema/context/spring-context.xsd
10        http://www.springframework.org/schema/aop
11        https://www.springframework.org/schema/aop/spring-aop.xsd
12 ">
13     <context:component-scan base-package="cn.tulingxueyuan"></context:component-scan>
14     <aop:aspectj-autoproxy></aop:aspectj-autoproxy>
15 </beans>
```

3、测试

MyTest.java

```
1   import cn.tulingxueyuan.inter.Calculator;
2   import org.springframework.context.ApplicationContext;
3   import org.springframework.context.support.ClassPathXmlApplicationContext;
4   public class MyTest {
5       public static void main(String[] args){
6           ApplicationContext context = new ClassPathXmlApplicationContext("aop.xml");
```

```
        Calculator bean = context.getBean(Calculator.class);
        bean.add(1,1);
    }
}
```

spring AOP的动态代理方式是jdk自带的方式，容器中保存的组件是代理对象com.sun.proxy.$Proxy对象

4、通过cglib来创建代理对象

MyCalculator.java

```
package cn.tulingxueyuan.inter;

import org.springframework.stereotype.Service;

@Service
public class MyCalculator {
    public int add(int i, int j) {
        int result = i + j;
        return result;
    }

    public int sub(int i, int j) {
        int result = i - j;
        return result;
    }

    public int mult(int i, int j) {
        int result = i * j;
        return result;
    }

    public int div(int i, int j) {
```

```
        int result = i / j;
        return result;
    }
}
```

MyTest.java

```
public class MyTest {
    public static void main(String[] args){
        ApplicationContext context = new ClassPathXmlApplicationContext("aop.xml");
        MyCalculator bean = context.getBean(MyCalculator.class);
        bean.add(1,1);
        System.out.println(bean);
        System.out.println(bean.getClass());
    }
}
```

可以通过cglib的方式来创建代理对象，此时不需要实现任何接口，代理对象是class cn.tulingxueyuan.inter.MyCalculator$$EnhancerBySpringCGLIB$$1f93b605类型

综上所述：在spring容器中，如果有接口，那么会使用jdk自带的动态代理，如果没有接口，那么会使用cglib的动态代理。动态代理的实现原理，后续会详细讲。

面试题：

- 什么是AOP
- JDK动态代理和CGLIB动态代理的区别
- 解释一下Spring AOP里面的几个名词
- 搭建基于Spring的AOP