Spring的AOP

AOP概述

- ✓ 编程语言最终极的目标就是能以更自然，更灵活的方式模拟世界，从原始机器语言道过程语言在岛面向对象的语言，编程语言一步步地用更自然，更强大的方式描述软件。
- ✓ AOP是软件开发饲养发展到一定阶段的产物，但AOP的出现并不是要完全代替OOP，而仅仅是作为OOP的有益补充。虽然AOP作为一项编程技术已经有多年的历史，但一直长时间停顿在学术领域，直到近几年，AOP才作为一项真正的实用技术在应用领域开疆扩土。
- ✓ 需要指出的是AOP的应用场合是受限的，它一般只适合用于那些具有横切逻辑的应用场合：如性能监测，访问控制，事物管理以及日志记录（虽然很多讲解日志记录的例子用于AOP的讲解，但很多人认为很难用AOP编写实用日志。）
- ✓ 不过，这丝毫不影响AOP作为一种新的软件开发思想在软件开发领域所占的地位。

AOP到底是什么

- ✓ AOP 是Aspect Oriented Programing 的简称，最初被翻译为“面向方面编程”，这个翻译向来为人所诟病，但是由于先入为主的效应，受众广泛，所以这个翻译依然被很多人使用。但我们更倾向于用“面向切面编程”的译法，因为他表达更加准确。

- ✓ AOP是一个软件层面的高度抽象，在现实世界中很难找到贴切的对应物，所以没有办法也只能同其他的讲解一样通过代码来帮助大家理解AOP的概念。

什么是代理模式及JDK动态代理

- ✓ **代理模式**：为目标对象提供一种代理以控制对这个对象的访问模式。

- ✓ 举个简单例子：一个客户想卖房子，这个过程需要了解房产市场，同买方协商，交易成功后还需要办理过户，而且他还一下找不到买家，这时他找到了房屋中介，希望房屋中介暂时代理他完成上面的操作并找到好买家卖个好价钱。这时卖房子客户就成为了代理的目标对象，而房屋介成为了代理对象。在卖房子过程中卖方无需关注上述繁琐的操作只需房屋中介找到买家后出售房屋即可，在出售房屋时候房屋中介就用代理的方式访问了目标对象--卖房者。在卖房子过程中了解市场，协商买方，办理过户这些操作都由房屋中介来完成，卖方只需要知道我的房子卖了多少钱这个业务逻辑了，这就是的代理模式。

代理模式有三个角色

- ✓ 代理模式一般涉及到的角色有：
- ✓ **抽象角色**：声明真实对象和代理对象的共同接口；
- ✓ **代理角色**：代理对象角色内部含有对真实对象的引用，从而可以操作真实对象，同时代理对象提供与真实对象相同的接口以便在任何时刻都能代替真实对象。同时，代理对象可以在执行真实对象操作时，附加其他的操作，相当于对真实对象进行封装。
- ✓ **真实角色**：代理角色所代表的真实对象，是我们最终要引用的对象。

JDK动态代理示例-1

UserService接口：

```
public interface UserService {

    void addUser(String username,String password);

    void deleteUser(int id);

    void updateUser(int id,String username,String
    password);

    String getUser(int id);

}
```

UserServiceImpl实现类：

```
public class UserServiceImpl implements UserService {

    public void addUser(String username, String password)
    {
        System.out.println("增加用户");
    }

    public void updateUser(int id, String username, String
    password) {
        System.out.println("更新用户");
    }

    public void deleteUser(int id) {
        System.out.println("删除用户");
    }

    public String getUser(int id) {
        System.out.println("返回用户");
        return "用户信息：100，张三，123456";
    }
}
```

JDK动态代理示例-2

```
public class LogHandler implements InvocationHandler {
	private Object targetObject;// 被代理的目标对象

	// 自定义方法，传入被代理的对象返回代理类对象
	public Object getProxy(Object targetObject) {
		this.targetObject = targetObject;

		//用Proxy类的静态方法newProxyInstance返回代理对象实例
		//需要传入三个参数：被代理对象的类加载器，被代理对象的接口，一个实现了InvocationHandler接口的类
	的对象实例
		return Proxy.newProxyInstance(targetObject.getClass().getClassLoader(),
					targetObject.getClass().getInterfaces(), this);
	}

	//覆盖invoke方法：此方法会在调用被代理类的方法的时候执行
	public Object invoke(Object proxy, Method method, Object[] args)
				throws Throwable {
		//日志操作
		this.log();
		//传入参数调用被代理类的原始方法，得到返回值
		Object returnValue = method.invoke(targetObject, args);
		//返回原始方法返回值
		return returnValue;		}

	// 日志操作
	private void log() {
		System.out.println("写入日志...");
	}
}
```

JDK动态代理示例-3

```
public class Test {
    public static void main(String[] args) {
        //创建一个日志处理类
        LogHandler handler = new LogHandler();

        //返回一个代理类对象实例，这个实例和被代理类的接口类型是一致的，JDK动态代理只对接口类型有效
        UserService service = (UserService) handler.getProxy(new UserServiceImpl());

        //调用的所有方法都会有日志功能
        service.addUser("张三", "123456");
        service.deleteUser(100);
        service.updateUser(100,"李四","999888");
        System.out.println(service.getUser(100));
    }
}
```

代理模式小结

- ✓ 我们虽然使用JDK的动态代理完成了代理模式，但是这种实现方式有几个明显需要改进的地方：

1.目标类的所有方法都添加到代理逻辑，而有时，这并不是我们所期望的，我们可能只希望对业务类中的某些特定方法添加代理业务。

2.我们通过硬编码的方式指定了织入代理业务的织入点，即在目标类业务方法的开始和结束前织入代码。

3.我们手工编写代理实例的创建过程，为不同创建代理时，需要分别编写相应的创建代码，无法做到通用。

- ✓ 以上三个问题在ＡＯＰ中占有重要地位，应为Spring AOP的主要工作就是围绕以上三点展开：Spring AOP 通过Pointcut（切点）指定在那些类的那些方法上添加横切逻辑，通过Advice（增强）描述横切逻辑和方法的具体织入点（方法前，方法后，方法的两端等），此外Spring还通过Advisor(切面)将Pointcut和Advice两者组装起来，有了Advisor的信息，Spring就可以利用JDK或CGLib的动态代理技术采用统一的方式为目标Bean创建织入切面的代理对象了。

AOP术语介绍

- ✓ **连接点（Joinpoint）：**程序执行的某特定位置：如类开始初始化前，类初始化后，类某个方法调用前，调用后，方法抛出异常后。一个类或一段程序拥有一些具有边界性质的特定点，这些代码中的特定点就称为“连接点”。
- ✓ Spring仅支持方法的连接点，即仅能在方法调用前，方法调用后，方法抛出异常时以及方法调用前后这些程序执行点织入增强，从某种程度上说AOP是一个黑客（因为它要想目前类中嵌入额外的代码逻辑），连接点就是AOP向目标类打入楔子的时候的点。

AOP术语介绍

- ✓ **切点（Pointcut）：**每个程序都拥有多个连接点，如一个拥有两个方法的类，这两个方法都是连接点，即连接点是程序类中客观存在的事物，但在这为数不多的连接点中，如何定位到某个感兴趣的连接点上呢？就是到底给哪个方法增强一下呢
- ✓ AOP通过“切点”定位特定的连接点，通过数据库查询的感念来理解切点和连接点的关系在合适不过了：连接点相当于数据库中的记录，而切点相当于查询条件。切点和连接点不是一对一的关系，一个切点可以匹配多个连接点。
- ✓ 在Spring中，切点通过org.springframework.aop.Pointcut接口进行描述，他使用类和方法作为连接的查询条件，Spring AOP的规则解析引擎负责解析切点所设定的查询条件，找到对应的连接点---其实确切的说，应该是执行点而非连接点，因为连接点是方法执行前，执行后等包括方位信息的具体程序执行点，而切点只定位到某个方法上，所以如果希望定位到具体的连接点上，还需要提供方位信息。
- ✓ 说白了，切点就是一个过滤条件，确定给哪些类的哪些方法织入增强

AOP术语介绍

- ✓ **增强（Advice）：** 增强是织入到目标类连接点上的一段程序代码，例如日志处理程序
- ✓ **目标对象(traget):**增强逻辑的织入目标类,如果没有AOP,目标业务类需要自己实现所有逻辑,在AOP帮助下,目标类只需实现非横切逻辑的程序逻辑,而日志处理,性能监视等横切逻辑可以使用AOP动态织入到特定的连接点上.
- ✓ **引介(Introdction)：** 引介是一种特殊的增强，他为类添加一些属性和方法，这样既使一个业务类原本没有实现某个接口，通过AOP的引介功能，我们可以动态地位该业务类添加接口的实例逻辑，让业务类成为这个接口的实现类。说白了就是假如一个类有方法1和方法2，用引介可以动态给类添加一个方法3。

AOP术语介绍

- ✓ **织入（Weaving）：**织入是将增强添加对目标类具体连接点上的过程，AOP像一台织布机，将目标类，增强或者引介通过AOP这台织布机天衣无缝的编制在一起，我们不能不说“织入”这个词太精辟了，根据不同的实现技术AOP有三种织入技术：

  1）编译期织入，这要求使用特殊的Java编译器。
  2）类加载期织入，这要求使用特殊的类装载器
  3）动态代理织入，在运行期为目标类添加增强生成子类的方式。
- ✓ **代理（Proxy）：**一个被AOP织入增强后，就产生了一个结果类，它是融合了原类和增强逻辑的代理类，根据不同的代理方式，代理类即可能是和原类具有相同接口的类，也可能就是原类的子类，所以我们可以采用调用原类相同的方式调用代理类。
- ✓ **切面（Aspect）：**切面由切点和增强（引介）组成，它既包括了横切逻辑的定义，也包括了连接点的定位，Spring AOP就是负责实施切面的框架，它将切面所定义的横切逻辑织入到切面所指定的连接点中。
- ✓ AOP的工作重心在于如何将增强应用于目标类对象的连接点上，这里首先包括两个工作,第一：如何通过切点和增强定位到连接点上，第二：如何在增强中编写切面的代码。

使用Spring进行面向切面（AOP）编程

- ✓ Spring提供了两种切面声明方式，实际工作中我们可以选用其中一种：
  - ➢ 基于注解方式声明切面
  - ➢ 基于XML配置方式声明切面

配置文件中添加AOP命名空间

要进行AOP编程，首先我们要在spring的配置文件中引入aop命名空间：

```
<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-2.5.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">
    ……
</beans>
```

基于注解方式声明切面——切面类

```
package com.spring;
import org.aspectj.lang.annotation.*;
/**
 *加入切面注解
 */
@Aspect
public class LogHandler {
	/**
	 * 定义Pointcut切点，切点的名称就是方法名m1，此方法不能有返回值和参数，
	 * 也不需要具体的实现，该方法只是一个标识
	 * Pointcut注解的内容是一个execution表达式，描述满足条件的方法
	 *  "execution(* add*(..))" 中第一个*表示有无返回值的方法全都匹配，
	 *                        add*表示add开头的所有方法
	 *                        (..)表示有没有参数都匹配
	 */
	@Pointcut("execution(* com.spring.UserServiceImpl.*(..))")
	private void m1(){}
	/**
	 * 定义Advice，标识在哪个切入点织入增强方法,@Before就是方法执行前
	 */
	@Before("m1()")
	private void log() {
		System.out.println("写入日志...");
	}
}
```

基于注解方式声明切面——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:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

    <!-- 设置AspectJ的注解支持：在Spring中集成了AspectJ， AspectJ是一个面向切面的框架 -->
    <aop:aspectj-autoproxy/>
    <!-- 配置相关的Bean -->
    <bean id="logHandler" class="com.spring.LogHandler"/>
    <bean id="userServiceImpl" class="com.spring.UserServiceImpl"/>
</beans>
```

基于XML配置方式声明切面——切面类

```
package com.spring;
public class LogHandler {
    private void log() {
        System.out.println("写入日志...");
    }
}
```

基于XML配置方式声明切面——配置文件

```
<!-- 配置相关的Bean类 -->
<bean id="logHandler" class="com.spring.LogHandler" />
<bean id="userServiceImpl" class="com.spring.UserServiceImpl" />
<!-- 配置AOP -->
<aop:config>
    <!-- 设置切面的类的Bean引用 -->
    <aop:aspect id="aspect1" ref="logHandler">
        <!-- 设置切点，命名为m1，表达式条件设置为所有add开头的方法 -->
        <aop:pointcut id="m1" expression="execution(* add*(..))"/>
        <!-- 设置增强，在方法执行之前执行log增强方法，引用m1切点-->
        <aop:before method="log" pointcut-ref="m1"/>
    </aop:aspect>
</aop:config>
```

JoinPoint类型

- ✓ 此类型封装了执行的原始方法的信息，例如方法的名称，参数列表等
- ✓ 示例：调用增强方法的时候同时打印输出被调用的方法的名称和参数值

```
private void log(JoinPoint joinPoint) {
        System.out.println("写入日志...");
        //输出方法名称
        System.out.println("调用方法名：" +joinPoint.getSignature().getName());
        //输出参数值
        Object[] args = joinPoint.getArgs();
        for (int i = 0;i < args.length ;i ++)
                System.out.println("参数" +  i + "=" + args[i]);
}
```

在Spring中实现AOP的两种机制

- ✓ Spring AOP 使用了两种代理机制
  - ➢ 基于JDK动态代理
  - ➢ 是基于CGLib的字节码生成
- ✓ 之所以使用两种机制，很大程度上是因为JDK本身只提供了接口代理，而不支持类代理，如果一个类没有实现接口，那么JDK动态代理技术就无能为力了，这时候Spring会选择使用CGLib实现AOP，因为CGLib是不需要类实现接口的，使用哪种代理技术Spring会灵活切换的。