火龙果 · 整理
uml.org.cn

Spring AOP

火龙果 · 整理
uml.org.cn

讲解 目录

1. Spring AOP的使用
2. AOP的相关概念
3. AOP和OOP的区别
4. AOP的应用场景
5. 动手实现一个AOP
6. AOP的实现原理及源码解析

Spring AOP的使用

Service：业务方法

```
public class Service {
    public void print(){
        System.out.println("----业务方法开始");
    }
}
```

```
public class TestAspect {
    public void doAfter(JoinPoint jp) {
        System.out.println("log Ending method: " + jp.getTarget().getClass().getName() + "." + jp.getSignature().getName());
    }
    public Object doAround(ProceedingJoinPoint pjp) throws Throwable {
        long time = System.currentTimeMillis();
        Object retVal = pjp.proceed();
        time = System.currentTimeMillis() - time;
        System.out.println("process time: " + time + " ms");
        return retVal;
    }
    public void doBefore(JoinPoint jp) {
        System.out.println("log Begining method: " + jp.getTarget().getClass().getName() + "." + jp.getSignature().getName());
    }
    public void doThrowing(JoinPoint jp, Throwable ex) {
        System.out.println("method " + jp.getTarget().getClass().getName() + "." + jp.getSignature().getName() + " throw exception");
        System.out.println(ex.getMessage());
    }
}
```

TestAspect：
增加业务方法的额外功能，也叫做通知（Advice）

Spring AOP的使用

applicationContext.xml配置

```
<!--业务方法-->
<bean id ="service" class="service.Service" />
<!--额外功能-->
<bean id="aspectBean" class="TestAspect" />
<aop:config>
    <!--定义切面-->
    <aop:aspect id="TestAspect" ref="aspectBean">
        <!--配置com.spring.service包下所有类或接口的所有方法-->
        <aop:pointcut id="businessService" expression="execution(* service.*.*(..))" />
        <aop:before pointcut-ref="businessService" method="doBefore"/>
        <aop:after pointcut-ref="businessService" method="doAfter"/>
        <aop:around pointcut-ref="businessService" method="doAround"/>
        <aop:after-throwing pointcut-ref="businessService" method="doThrowing" throwing="ex"/>
    </aop:aspect>
</aop:config>
```

测试结果

```
log Begining method: service.Service.print
1111111
process time: 27 ms
log Ending method: service.Service.print
```

测试类：TestAop

```
public class TestAop {
    @Test
    public void testAop() {
        ApplicationContext ac = new ClassPathXmlApplicationContext( configLocation: "applicationContext.xml");
        Service service = (Service)ac.getBean( s: "service");
        service.print();
    }
}
```

火龙果 · 整理
uml.org.cn

AOP 定义

AOP定义：

AOP(Aspect Oriented Programing), 面向切面编程,通过动态搭建代理类来解决项目中代码冗余的问题。

为什么使用AOP：

- 用了aop能让你少写很多代码
- 为了更清晰的逻辑，可以让你的业务逻辑去关注自己本身的业务，而不去想一些其他的事情，这些其他的事情包括：安全，事物，日志等

Spring AOP 核心概念

- 通知(Advice)

  就是你想要的功能，也就是上面说的 安全，事物，日志等。你给先定义好把，然后在想用的地方用一下

- 连接点（JoinPoint）

  就是spring允许你使用通知的地方,和方法有关的前前后后（抛出异常），都是连接点

- 切入点（Pointcut）

  一个类里，有15个方法,有几十个连接点，只想让其中的几个，在调用这几个方法之前，之后或者抛出异常时干点什么，那么就用切点来定义这几个方法，让切点来筛选连接点，选中那几个你想要的方法

- 切面（Aspect）

  切面是通知和切入点的结合。通知说明了干什么和什么时候干,而切入点说明了在哪干（指定到底是哪个方法），这就是一个完整的切面定义。

- 切面（Aspect）

  切面是通知和切入点的结合。通知说明了干什么和什么时候干,而切入点说明了在哪干（指定到底是哪个方法），这就是一个完整的切面定义。

- 引入（introduction）

  就是spring允许你使用通知的地方,和方法有关的前前后后（抛出异常），都是连接点允许我们向现有的类添加新方法属性。就是把切面（也就是新方法属性：通知定义的）用到目标类中

- 目标（target）

  引入中所提到的目标类，也就是要被通知的对象，也就是真正的业务逻辑

- 代理(proxy)

  怎么实现整套aop机制的，都是通过代理

- 织入(weaving)

  把切面应用到目标对象来创建新的代理对象的过程

AOP和OOP的区别

- AOP是面向切面编程，OOP是面向对象编程

- AOP适合定义横向关系，而OOP适合定义纵向关系

  OOP引入封装、继承、多态等概念来建立一种对象层次结构，用于模拟公共行为的一个集合。不过OOP允许开发者定义纵向的关系，但并不适合定义横向的关系，而AOP技术恰恰相反，它利用一种称为"横切"的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，即切面。所谓"切面"，简单说就是那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来

AOP的应用场景

- Log 日志
- Authentication 权限
- Transactions 事务
- Form Validation表单验证

……

动手实现一个AOP

```java
public interface IHello {
    /**
     * 业务方法
     * @param str
     */
    void sayHello(String str);
}
```

```java
public class Hello implements IHello {
    @Override
    public void sayHello(String str) {
        System.out.println("hello "+str);
    }
}
```

```java
public class Logger {
    public static void start(){
        System.out.println(new Date()+ " say hello start...");
    }
    public static void end(){
        System.out.println(new Date()+ " say hello end");
    }
}
```

```java
public class ProxyHello implements IHello {
    private IHello hello;
    public ProxyHello(IHello hello) {
        super();
        this.hello = hello;
    }
    @Override
    public void sayHello(String str) {
        Logger.start();//添加特定的方法
        hello.sayHello(str);
        Logger.end();
    }
}
```

```java
public class Test {
    public static void main(String[] args) {
        IHello hello = new ProxyHello(new Hello());
        hello.sayHello( str: "明天");
    }
}
```

```
Wed Jul 26 11:14:51 CST 2017 say hello start...
hello 明天
Wed Jul 26 11:14:51 CST 2017 say hello end
```

动手实现一个AOP

思考1：

1).如果我们像Hello这样的类很多,那么,我们是不是要去写很多个HelloProxy这样的类呢？

2).如何解决？

动手实现一个AOP

```java
public class DynaProxyHello implements InvocationHandler {
    private Object target;//目标对象
    /**
     * 通过反射来实例化目标对象
     * @param object
     * @return
     */
    public Object bind(Object object){
        this.target = object;
        return Proxy.newProxyInstance(this.target.getClass().getClassLoader(), this.target.getClass().getInterfaces(), h: this);
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        Object result = null;
        Logger.start();//添加额外的方法
        //通过反射机制来运行目标对象的方法
        result = method.invoke(this.target, args);
        Logger.end();
        return result;
    }
}
```

```java
public class DynaTest {
    public static void main(String[] args) {
        IHello hello = (IHello) new DynaProxyHello().bind(new Hello());
        hello.sayHello( str: "明天");
    }
}
```

动手实现一个AOP

问题：

1) 额外功能(记录日志)类和代理类耦合性比较高

火龙果 · 整理
uml.org.cn

Spring AOP实现原理

- 手动使用ProxyFactoryBean来创建Proxy对象
- ProxyFactoryBean如何加载配置
- ProxyFactoryBean如何获取代理对象
- 获取到的拦截器链式如何执行的

Spring AOP实现原理

手动使用ProxyFactoryBean来创建Proxy对象,在执行业务方法的各个阶段，插入额外功能

```
public static void main(String[] args) {
    //1.针对不同的时期类型，提供不同的Advice
    Advice beforeAdvice = new TicketServiceBeforeAdvice();
    Advice afterReturningAdvice = new TicketServiceAfterReturningAdvice();
    Advice aroundAdvice = new TicketServiceAroundAdvice();
    Advice throwsAdvice = new TicketServiceThrowsAdvice();
    RailwayStation railwayStation = new RailwayStation();
    //2.创建ProxyFactoryBean,用以创建指定对象的Proxy对象
    ProxyFactoryBean proxyFactoryBean = new ProxyFactoryBean();
    //3.设置Proxy的接口
    proxyFactoryBean.setInterfaces(TicketService.class);
    //4. 设置RealSubject
    proxyFactoryBean.setTarget(railwayStation);
    //5.使用JDK基于接口实现机制的动态代理生成Proxy代理对象，如果想使用CGLIB，需要将这个flag设置成true
    proxyFactoryBean.setProxyTargetClass(true);
    //6. 添加不同的Advice
    proxyFactoryBean.addAdvice(afterReturningAdvice);
    proxyFactoryBean.addAdvice(aroundAdvice);
    proxyFactoryBean.addAdvice(throwsAdvice);
    proxyFactoryBean.addAdvice(beforeAdvice);
    proxyFactoryBean.setProxyTargetClass(false);
    //7通过ProxyFactoryBean生成Proxy对象
    TicketService ticketService = (TicketService) proxyFactoryBean.getObject();
    ticketService.sellTicket();
}
```

Spring AOP实现原理

ProxyFactoryBean，是Spring AOP的核心，我们为ProxyFactoryBean提供了如下信息

- Adivce列表
- 真正的实例对象引用ticketService
- 告诉ProxyFactoryBean使用基于接口实现的JDK动态代理机制实现proxy

我们要从ProxyFactoryBean中获得什么？ ----代理对象

-----ProxyFactoryBean倒像是一个桥梁，准备了必要的环境（比如将配置文件上的配置加载到属性上），真正的代理对象靠AopProxy生成。

Spring AOP实现原理

加载配置： 创建并调用AopProxy返回代理对象。

```
public Object getObject() throws BeansException {
        this.initializeAdvisorChain();
        if(this.isSingleton()) {
                return this.getSingletonInstance();
        } else {
                if(this.targetName == null) {
                        this.logger.warn("Using non-singleton");
                }
                return this.newPrototypeInstance();
        }
}
```

```
initializeAdvisorChain(){
    根据interceptorNames拿到每一个interceptor（即advice）的名字
    根据名字通过ioc拿到advice实例，并将其加入到advisors成员中
}
```

Spring AOP实现原理

```
// 如何拿到
class AdvisedSupport{
   private List<Advisor> advisors;
   getInterceptorsAndDynamicInterceptionAdvice(method, targetClass){
     AdvisedSupport.MethodCacheKey cacheKey = new AdvisedSupport.MethodCacheKey(method);
      List cached = (List)this.methodCache.get(cacheKey);
      if(cached == null) {
         cached = this.advisorChainFactory.getInterceptorsAndDynamicInterceptionAdvice(this, method,
targetClass);
         this.methodCache.put(cacheKey, cached);
      }
      return cached;
   }
}
```

1.先从缓存里看看
2.如果缓存中没有
3.遍历advisors（依据配置加载的所有advisors（advisor=advice + pointcut）），通过当前advisor中的pointcut的matches方法判断advisor是否使用这个method，使用则将其转换为Interceptor，加入到chain中
4.返回chain并加入缓存

Spring AOP实现原理

ProxyFactoryBean如何获取代理对象

```
protected Object getProxy(AopProxy aopProxy) {
    //创建并调用AopProxy返回代理对象
    return aopProxy.getProxy(this.proxyClassLoader);

}
```

```
public interface AopProxy {
    Object getProxy();
    Object getProxy(ClassLoader classLoader);
}
class JdkDynamicAopProxy implements AopProxy, InvocationHandler{
    private final AdvisedSupport advised;   // 包含大量配置信息
    public Object getProxy(ClassLoader classLoader) {
        Class[] proxiedInterfaces = AopProxyUtils.completeProxiedInterfaces(this.advised);
        findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);
        // 返回代理对象
        return Proxy.newProxyInstance(classLoader, proxiedInterfaces, this);
    }
    // InvocationHandler接口方法实现
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        1. 根据method拿到对应的拦截器链
        2. 执行拦截器链
    }
}
```

Spring AOP实现原理

拦截链的执行

```
// 链的执行
class ReflectiveMethodInvocation implements ProxyMethodInvocation{
    // 目标方法、参数和类型
    protected final Object target,Method method,Object[] arguments,Class targetClass;
    // 当前拦截器的索引
    private int currentInterceptorIndex = -1;
    protected final List interceptorsAndDynamicMethodMatchers;// 拦截器链（已经从advice转化
为了interceptor（适配器模式））
    public Object proceed() throws Throwable {
        // 如果执行到链的最后，则直接执行目标方法
        // 获取当前interceptor
        Object interceptorOrInterceptionAdvice =

this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptorIndex);
        if (interceptorOrInterceptionAdvice符合特定类型) {
            // 执行特定逻辑
        }else {
            // 执行拦截器
            return ((MethodInterceptor) interceptorOrInterceptionAdvice).invoke(this);
        }
    }
}
```

Spring AOP实现原理

aop拦截器链的执行逻辑如下

1) 执行所有的前置通知，如果碰到后置通知，则方法入栈（递归调用）。
2) 执行目标方法
3) 执行后置通知（原来压栈的方法出栈）
4) 异常通知（与后置通知类似（都是在方法的后边执行嘛），不过，貌似一个方法的异常通知只能有一个）

火龙果 · 整理
uml.org.cn

The End