笔记：

内容：三部分

包括（动态代理+配置aop+注解aop）

## 第一部分动态代理：

jdk 动态代理 + CGlib动态代理

##### 》》》》jdk动态代理如下

**public** **class** JDKproxy **implements** InvocationHandler {

// 我们创建的是（获取的是：目标对象）

**private** Object targetObject ;

**public** Object creageObject(Object targetObject){ //UserService us = new UserServiceImpl();

**this**.targetObject = targetObject;

//return Proxy.newProxyInstance(loader类加载器, interfaces接口, h 当前对象)

**return** Proxy.*newProxyInstance*(

// 获取目标对象的类加载器

targetObject.getClass().getClassLoader(),

// 获取当前目标对象的接口

targetObject.getClass().getInterfaces(),

// 当前对象

**this**);

}

@Override

**public** Object invoke(Object proxy, Method method, Object[] args)

**throws** Throwable {

Object invoke = **null** ;

**if**(method.getName().contains("delete")){

MethodOutSub ms = **new** MethodOutSub();

// 前置 通知

ms.sortDemo3();

/invoke=method.invoke(obj目标对象, args参数类型)

// invoke=method.invoke(targetObject, args);//targetObject就是目标对象

**//当代理对象执行方法的时候才去调用此方法**

invoke = method.invoke(targetObject, args);

// 后置 通知

ms.sortDemo3();

}

**return** invoke;

}

#### 》》》**CGlib动态代理如下**

**public** **class** CGlibProxy **implements** MethodInterceptor {

**private** Object objectTarget ;

**public** Object createObjectProxy(Object objectTarget ){

**this**.objectTarget = objectTarget;

// 这个类 就是用来造代理类的

Enhancer e = **new** Enhancer();

e.setClassLoader(**this**.objectTarget.getClass().getClassLoader());

e.setSuperclass(**this**.objectTarget.getClass());

e.setCallback(**this**);

**return** e.create(); //完成代理类的创建

}

@Override

**public** Object intercept(Object arg0, Method method, Object[] arg2,

MethodProxy arg3) **throws** Throwable {

/\*

\* 无论什么代理，必须要运行目标对象中的方法

\* 但是运行的时候这个对象怎么获取到呢？method.invoke(obj, arg2); 》》obj？？？

\*

\* 1:private Object objectTarget ;

\* 2:怎么给ObjectTarget赋值呢？

\* 两种方式：1：构造器 2：自定义一个方法

\* 我们选择2

\* 所以我就构建了一个方法createObjectProxy()

\*

\* UserService us = new UserService();

\*

\* 目标对象 ：也就是上面的us

\*

\*/

/\*

\* 反射： public void show(){}

\* public String show(){ return "xiaofang";}

\*/

System.*out*.println("-----前置通知----");

Object invoke = method.invoke(objectTarget, arg2);

**return** invoke;

}

## 第二部分：配置前置、后置、异常、环绕通知

<!-- 通知 （方法） -->

<bean id=*"security"* class=*"com.offcn.test004.Security"*></bean>

<!--目标对象 -->

<bean id=*"userService"* class=*"com.offcn.test004.UserServiceImpl"*></bean>

<!-- config表示，配置切面环境 -->

<aop:config>

<!--切面 ：某些类中的方法

切面表达式：execution(\* com.offcn.test004.UserServiceImpl.deleteUser(..)

怎么来的？

execution(public void com.offcn.test004.UserServiceImpl.deleteUser(String))

execution(public com.offcn.test004.UserServiceImpl.deleteUser(String))

execution(\* void com.offcn.test004.UserServiceImpl.deleteUser(String))

execution(public void com.offcn.test004.UserServiceImpl.deleteUser(..)) //..代表任意参数

execution(public void com.offcn.test004.UserServiceImpl.\*(..)) //..代表任意参数

-->

<aop:pointcut expression=*"execution(\* com.offcn.test004.UserServiceImpl.\*(..))"* id=*"pointcut001"*/>

<!--配置通知 -->

<aop:aspect ref=*"security"* id=*"p001"*>

<!-- 给切面pointcut001 配置前置通知after : 给userServiceImpl中的某个方法 添加一个方法after写到前面 -->

后置通知 <aop:after method="after" pointcut-ref="pointcut001" />

前置通知 <aop:before method="before" pointcut-ref="point001"></aop:before>

异常通知

<aop:after-throwing method="excptiondemo" pointcut-ref="point001" throwing="valueinfo">

</aop:after-throwing>

环绕通知

<**aop:around method="arounddemo" pointcut-ref="point001"**></**aop:around**>

</aop:aspect>

## 第三部分：注解aop配置（三个步骤）

第一步：  
*<!--开启aop注解-->*<**aop:aspectj-autoproxy**></**aop:aspectj-autoproxy**>

第二步：

@Aspect  
**public class** Security

第三步：

*/\*简化版\*/*@Before(value = **"execution(\* com.offcn.zhujie.UserServiceImpl.deleteUser(..))"**)  
**public void** show(){  
 System.***out***.println(**"======我这个通知要加到com.offcn.zhujie.UserServiceImpl.deleteUser方法上去"**);  
}

*/\*复杂版  
 \*  
 \* 首先：造一个空方法（方法体内没有数据）  
 \* 在方法上面打上注解@Pointcut 后面加上切面表达式  
 \* \*/* @Pointcut(value =**"execution(\* com.offcn.zhujie.UserServiceImpl.saveUser(..))"**)  
 **public void** ss(){  
 }  
  
 @Pointcut(value =**"execution(\* com.offcn.zhujie.UserServiceImpl.updateUser(..))"**)  
 **public void** sm(){  
 }  
  
*// 这是 一个通知* @After(value = **"ss()||sm() "**)  
 **public void** show002(){  
 System.***out***.println(**"这是复杂类型的注解"**);  
 }  
  
 @Around(value = **"sm()"**)  
 **public void** around(ProceedingJoinPoint joinPoint){  
  
 System.***out***.println(**"这是环绕通知前---执行中"**);  
  
 */\*快捷键 ctrl + alt +T \*/* **try** {  
 joinPoint.proceed();  
 } **catch** (Throwable throwable) {  
 throwable.printStackTrace();  
 }  
  
 System.***out***.println(**"这是环绕通知后---执行中"**);  
 }  
  
 @AfterThrowing(value = **"ss()"**,throwing = **"info"**)  
 **public void** throwdemo(Throwable info){  
  
*// System.out.println(info);* System.***out***.println(**"发生了异常，我来执行"**);  
  
 }