第一代aop是静态的 生成.class文件后与原有的系统结合在一起

第二胎aop是动态的,运行期间通过动态代理产生代理对象

jdk接口代理,cblib类继承 动态生成字节码的class文件

package sp.test.aop;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.annotation.AfterReturning;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

import org.aspectj.lang.annotation.DeclareParents;

import org.aspectj.lang.annotation.Pointcut;

import org.springframework.stereotype.Component;

@Component

@Aspect

public class MyAspect {

@Pointcut("execution (\* sp.test..\*.show(..)) ")

public void pc1() {

}

// 会匹配Singer1的所有方法

@Pointcut("within(sp.test.aop.Singer1)")

public void pc2() {

}

// 当目标对象是Singer1的时候匹配上

@Pointcut("target(sp.test.aop.Singer1)")

public void pc3() {

}

// 当代理对象是一个Singer1的时候匹配上

// 现在是无法匹配上的 因为是基于接口的代理

// 而Singer1是一个类!

@Pointcut("this(sp.test.aop.Singer1)")

public void pc4() {

}

// 当代理对象是一个Performer的时候匹配所有方法

@Pointcut("this(sp.test.aop.Performer)")

public void pc5() {

}

// 会动态匹配参数是否是一个String

// 并且参数有且只有一个String

@Pointcut("args(String)")

public void pc6() {

}

// 同上,但是匹配第一个是String,第二个任意的方法

@Pointcut("args(String,\*)")

public void pc7() {

}

// 同上,但是匹配第一个是String,其他任意的方法

@Pointcut("args(String,..)")

public void pc8() {

}

// 匹配所有打了MyAnno1的类的所有方法

@Pointcut("@within(MyAnno1)")

public void pc9() {

}

// 在Spring中 @target和@within 没太大区别

// 只是@within是静态匹配 target是动态匹配

// 匹配所有打了MyAnno1的的目标对象

@Pointcut("@target(MyAnno1)")

public void pc10() {

}

// 当参数被打上MyAnno1的时候

// 要注意有些Annotation没有继承,如果传进来的是一个代理

// 那这个代理是没有相应的annotation的

@Pointcut("@args(sp.test.aop.MyAnno1,..)")

public void pc11() {

}

// 会匹配所有打上MyAnno1的方法

@Pointcut("@annotation(MyAnno1)")

public void pc12() {

}

@Before("pc12()")

public void before() {

System.out.println( "前置" );

}

// /会匹配aop包下所有类的所有方法

//+有且只有第一个参数的动态类型是String(即在方法上可以声明Object,但实际是个String)

//+把第一个参数的作为参数s调用after函数

//第一个参数jp可以没有

//args(String)会动态匹配

//如果你写的是execution (\* sp.test.aop.\*.\*(String)) 则是静态匹配String!

@AfterReturning("execution (\* sp.test.aop.\*.\*(..)) && args(String) && args(s) && this(Performer) &&this(othis)")

public void after(JoinPoint jp, String s,Performer othis) {

System.out.println( "后置" + s +othis);

}

//如果将上面一个写成切点,再利用的话 是下面这样

@Pointcut("execution (\* sp.test.aop.\*.\*(..)) && args(String) && args(s) && this(Performer) &&this(othis)")

public void pc13(String s,Performer othis) {

}

//这里参数名字可以不一样

@AfterReturning("pc13(s1,s2)")

public void after2(JoinPoint jp,String s1,Performer s2){

System.out.println("切qieqie");

}

//其他的也有类似的用法 target(Performer) && target(t)

//表示给Singer1这个类添加接口及其实现

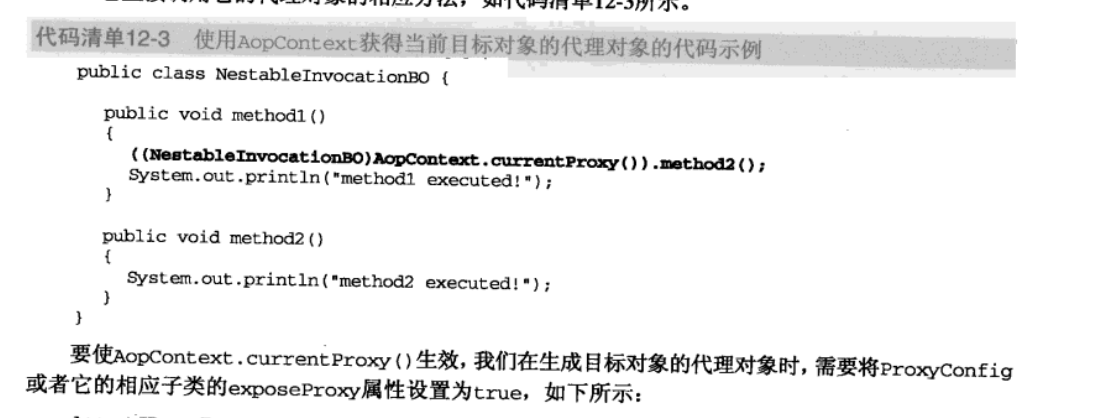
//可以使用通配符

@DeclareParents(value="sp.test.aop.Singer1",defaultImpl=MyInt1Impl.class)

private MyInt1 myin1;

}





在自身的方法m1里调m2将不会触发代理

