Spring_day2

我的第一个笔记本 春天 项目管理框架 作者:小陈

1. 业务层存在问题

```
package staticproxy;

public class UserServiceImpl implements UserSerivce {

    public void save() {
        try {
            System.out.println("开启事务");
            System.out.println("处理业务逻辑,调用DAO~~");
            System.out.println("提交事务");
        } catch (Exception e) {
            System.out.println("回滚事务");
            e.printStackTrace();
        }
    }

    public void delete() {
        try {
            System.out.println("开启事务");
            System.out.println("处理业务逻辑,调用DAO~~");
            System.out.println("提交事务");
        } catch (Exception e) {
            System.out.println("回滚事务");
            e.printStackTrace();
        }
    }

    public void update() {
        try {
            System.out.println("开启事务");
            System.out.println("处理业务逻辑,调用DAO~~");
            System.out.println("提交事务");
        } catch (Exception e) {
            System.out.println("回滚事务");
            e.printStackTrace();
        }
    }

    public void query() {
        try {
            System.out.println("开启事务");
            System.out.println("处理业务逻辑,调用DAO~~");
            System.out.println("提交事务");
        } catch (Exception e) {
            System.out.println("回滚事务");
            e.printStackTrace();
        }
    }
```

> **问题:**从上图中可以看出,现有业务层中控制事务代码出现了**大量的冗余**,如何解决现有业务层出现的冗余问题?

2. 静态代理

> **目标类(target):**被代理类称之为目标类
> **开发代理的原则:** 和目标类功能一致且实现相同的接口

1. 开发静态代理类

```java
package staticproxy;


//开发静态代理类  代理类和目标类实现相同的接口
public class UserServiceStaticProxy implements UserSerivce {

    private UserSerivce userSerivce;//依赖真正的目标类
    public void setUserSerivce(UserSerivce userSerivce) {
        this.userSerivce = userSerivce;
    }

    public void save() {
        try {
            System.out.println("开启事务");
            //?
            userSerivce.save();
            System.out.println("提交事务");
        } catch (Exception e) {
            System.out.println("回滚事务");
            e.printStackTrace();
        }
    }

    public void delete() {
        try {
            System.out.println("开启事务");
            //?
            userSerivce.delete();
            System.out.println("提交事务");
        } catch (Exception e) {
            System.out.println("回滚事务");
            e.printStackTrace();
        }
    }

    public void update() {
        try {
            System.out.println("开启事务");
            //?
            userSerivce.update();
            System.out.println("提交事务");
        } catch (Exception e) {
            System.out.println("回滚事务");
            e.printStackTrace();
        }
    }

    public void query() {
        try {
            System.out.println("开启事务");
            //?
            userSerivce.query();
            System.out.println("提交事务");
        } catch (Exception e) {
            System.out.println("回滚事务");
            e.printStackTrace();
        }
    }

}
```

2. 配置静态代理类

```xml
<!-- 管理静态代理类 -->
<bean id="userServiceStaticProxy" class="staticproxy.UserServiceStaticProxy">
    <!-- 注入目标类 -->
    <property name="userSerivce" ref="userSerivce"/>
</bean>

<!--  管理业务层组件类 -->
<bean id="userSerivce" class="staticproxy.UserServiceImpl"></bean>
```

3. 通过调用静态代理类完成功能

```java
//启动工厂
ApplicationContext context = new ClassPathXmlApplicationContext("staticproxy/spring.xml");

UserSerivce bean = (UserSerivce) context.getBean("userServiceStaticProxy");

bean.save();
```

> **问题**:往往在开发我们书写的不仅仅是一个业务层,两个业务层,而我们的业务层会有很多,**如果为每一个业务层开发一个静态代理类,不仅没有减轻工作量,甚至让我们的工作量多了一倍不止**怎么解决以上这个问题呢?能不能为我们现有的业务层在运行过程中**动态创建代理类**,通过**动态代理类**去解决我们现有业务层中业务代码冗余的问题 .

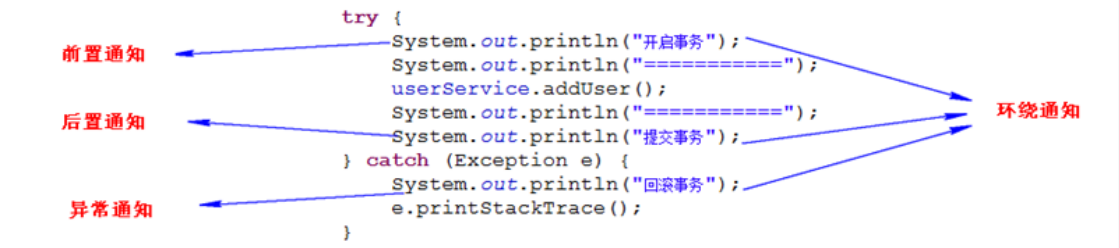
3. 动态代理

> 通过jdk提供的**Proxy**这个类,动态为现有的业务生成代理类
>
> > **参数一**:当前线程类加载器
> > **参数二:**生成代理类的接口类型
> > **参数三:**通过代理类对象调用方法时会优先进入参数三中的invoke方
> > **Proxy.newProxyInstance(loader, interfaces, h)**;//返回值就是动态代理对象

---

4. AOP 中的几个重要概念

- **通知(Advice):** 处理目标方法以外的操作都称之为通知
- **切入点(PointCut):** 要为哪些类中的哪些方法加入通知
- **切面(Aspect):** 通知 + 切入点



5. AOP的编程步骤

- 导入AOP依赖jar
- 引入AOP命名空间
- 开发通知类
- 配置AOP切面
- 测试

```
导包
    aopalliance-1.0.jar
    aspectjrt-1.7.4.jar
    aspectjweaver-1.7.4.jar
加入aop的命名空间
    <beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
                            http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
                            http://www.springframework.org/schema/aop
                            http://www.springframework.org/schema/aop/spring-aop-3.2.xsd ">
写类 实现spring 提供通知接口
    前置通知    MethodBeforeAdvice
    环绕通知    MethodInterceptor
    后置通知    AfterReturningAdvice
    异常通知    ThrowsAdvice

配置aop
    <!--声明通知-->
    <bean id="myTimeAdvice" class="zpark.service.MyTimeAdvice"></bean>
    <aop:config >
        <!-- 切入点 -->
        <aop:pointcut expression="execution(* zpark.service.DeptServiceImpl.save(..))" id="pc"/>
        <!--   通知+切入点 -->
        <aop:advisor advice-ref="myTimeAdvice" pointcut-ref="pc"/>
    </aop:config>
```

6. 前置通知

```
MethodBeforeAdvice  前置通知
类   implements MethodBeforeAdvice {
    /**
     * 参数一:目标类中当前调用的方法对象
     * 参数二:当前调用方法的参数
     * 参数三:目标对象
     */
    public void before(Method m, Object[] args, Object target)
            throws Throwable {
        System.out.println("日志通知类记录方法的名字为: "+m.getName());
    }
}
```

7. 环绕通知

```
MethodInterceptor  环绕通知
类   implements MethodInterceptor {
    public Object invoke(MethodInvocation mi) throws Throwable {
        System.out.println("记录方法运行时间通知:");
        long startTime = new Date().getTime();
        //执行方法调用
        Object proceed = mi.proceed(); //返回值为目标方法返回值,这个返回值一定要在通知类中返回
        long endTime = new Date().getTime();
        System.out.println("运行时间:  "+(endTime-startTime));
        return proceed;
    }
}
```

8. 后置通知

```
AfterReturningAdvice  后置通知
        implements AfterReturningAdvice{
            /**
             * 主体逻辑：此中的过程将在核心业务执行之后执行
             * ret:目标方法返回值 如果方法是void，则ret为null
             * m: 目标方法
             * param: 方法参数表
             * target: 目标
             */
            public void afterReturning(Object ret, Method m, Object[] param,
                    Object target) throws Throwable {
            }
        }
```

9. 异常通知

```
类implements ThrowsAdvice{
    //ex:目标抛出的异常
    public void afterThrowing(Exception ex){
        System.out.println("ex:"+ex.getMessage());
    }
}
```

10. execution表达式

```
execution(返回值   包.类.方法(参数表));
    execution(* com.zpark.service.UserServiceImpl.save(..))
        返回值:任意
        包:com.zpark.service
        类:UserServiceImpl
        方法:save
        参数:任意
    execution(* com.zpark.service.UserServiceImpl.*(..))
        返回值:任意
        包:com.zpark.service
        类:UserServiceImpl
        方法:任意
        参数:任意
    execution(* com.zpark.service.*.*(..))
        返回值:任意
        包:com.zpark.service
        类:任意
        方法:任意
        参数:任意
    execution(* com.zpark..*.*(..))
        返回值:任意
        包:com.zpark..
        类:任意
        参数表:任意
    execution(* *(..))
        返回值:任意
        包:任意
        类:任意
        参数表:任意
    原则:切入尽量的精准,避免不必要的切入
        execution(* com.zpark..UserService*.*User*(..))
            返回值:任意
            包:com.zpark及com.zpark子包
            类:以UserService开头的类
            方法:含有User的方法
            参数表:任意
```