Spring01 回顾：

1. 前天讲到Spring哪几个模块？

2. 简述Spring IOC容器

a. 创建对象

🡪 构造函数

🡪 工厂

b. DI 依赖注入

🡪 set方法注入

\* 普通字段注入值 、集合属性

\* 内部bean注入

\* p 名称空间

\* 自动装配

\* 注解

3. Spring 如何与Struts2整合？

目标：

1.代理模式

2. Aop编程

\* 手动实现aop编程

\* spring Aop 注解方式

\* Spring Aop XML 配置

\* 切入点表达式语法详解

3. Spring 对jdbc模块的支持

# 1. 代理模式

## 概念

Proxy, 表示代理！ 提供了对目标对象另外的访问方式，即通过代理访问**目标对象**！

Java中代理：

静态代理

动态代理

Jdk 代理

Cglib 代理

Spring默认支持

## 静态代理

特点：

1. 目标对象必须要实现接口

2. 代理对象，要实现与目标对象一样的接口

需求：

IUserDao.java dao接口

UserDao.java dao实现

Void save(); // 保存实现

UserDaoProxy.java dao代理类，对UserDao中功能进行扩展！

App

缺点：

1. 代理对象，需要依赖目标对象的接口！

如果接口功能变化，目标对象变化，会引入代理对象的变化!

2. 对每一个目标对象，都要分别写一个代理类，麻烦！

(代理工厂)

## 动态代理

动态代理：

1. 通常说的动态代理，就是指jdk代理!

因为是通过jdk的api在运行时期，动态的生成代理对象的！

2. 目标对象一定要实现接口, 代理对象不用实现接口!

JDK 生成代理对象的Api

|-- Proxy

|  |  |
| --- | --- |
| static [Object](mk:@MSITStore:E:\API\JDK_API_1_6_zh_CN.CHM::/java/lang/Object.html) | [**newProxyInstance**](mk:@MSITStore:E:\API\JDK_API_1_6_zh_CN.CHM::/java/lang/reflect/Proxy.html#newProxyInstance(java.lang.ClassLoader, java.lang.Class[], java.lang.reflect.InvocationHandler))([ClassLoader](mk:@MSITStore:E:\API\JDK_API_1_6_zh_CN.CHM::/java/lang/ClassLoader.html) loader, [Class](mk:@MSITStore:E:\API\JDK_API_1_6_zh_CN.CHM::/java/lang/Class.html)<?>[] interfaces, [InvocationHandler](mk:@MSITStore:E:\API\JDK_API_1_6_zh_CN.CHM::/java/lang/reflect/InvocationHandler.html) h) |

参数loader :

当前目标对象使用的类加载器！

参数interfaces :

当前目标对象实现的接口

参数 h:

接口类型，事件处理器.

当执行目标对象方法的时候，会触发事件； 把当前执行的方法(method对象)，传入事件处理器方法参数中, 这样就可以根据业务逻辑，判断是否执行目标对象方法或扩展功能！

## Cglib代理

也叫”子类代理”

当目标对象没有实现接口，就不能使用jdk提供的代理，可以以子类的方式实现！

在运行时期动态在内存中构建一个子类对象的方法，从而对目标对象扩展，这种就是cglib代理！

Spring也支持cglib代理，核心包中已经包含此功能！

* **JDK的动态代理有一个限制，就是使用动态代理的对象必须实现一个或多个接口。如果想代理没有实现接口的类，就可以使用CGLIB实现。**
* **CGLIB是一个强大的高性能的代码生成包，它可以在运行期扩展Java类与实现Java接口。它广泛的被许多AOP的框架使用，例如Spring AOP和dynaop，为他们提供方法的interception（拦截）。**
* **CGLIB包的底层是通过使用一个小而快的字节码处理框架ASM，来转换字节码并生成新的类。不鼓励直接使用ASM，因为它要求你必须对JVM内部结构包括class文件的格式和指令集都很熟悉。**

如：

public class UserDao { }

public class $Cglib\_my\_class extends UserDao {}

总结：

1. 目标对象可以不实现接口

2. 目标类不能为final， 如果为final报错

3. 方法如果为final/static, 不会被代理拦截！ 会直接执行目标对象方法 ！

## 代理总结

在SpringAop编程中，

如果目标对象有实现接口，spring使用jdk提供的代理生成代理对象！

如果目标对象没有实现接口，使用cglib代理！

如果目标没有实现接口、且为final , 不能进行aop编程，报错！不能生成代理！

# 2. Aop编程

|  |
| --- |
| **public void add(User user) {**  **Session session = null;**  **Transaction trans = null;**  **try {**  **session = HibernateSessionFactoryUtils.getSession(); 【关注点代码】**  **trans = session.beginTransaction(); 【关注点代码】**    **session.save(user); // 业务**    **trans.commit(); 【关注点代码】**  **} catch (Exception e) {**  **e.printStackTrace();**  **if(trans != null){**  **trans.rollback();**  **}**  **} finally{**  **HibernateSessionFactoryUtils.closeSession(session);**  **}**  **}** |

Aop 编程，

主要是分离业务代码与关注点代码！

关注点代码，写一次，在执行业务代码时候**动态植入关注点代码**！

## 自己实现aop编程

## aop编程, 注解方式

概念：

Aop编程：

关注点代码与业务代码分离！(jdk/cglib代理)

关注点：

重复执行的代码, 也叫关注点代码!

切面：

**关注点代码形成的类，就叫做切面**

springAop编程，也叫面向切面编程！

Aop: Aspect Object Programming 面向切面编程!

举例，哪些是切面？

事务，权限控制, 日志…

切入点表达式

拦截方法，给方法所在的类，生成代理对象！

Spring在初始化容器的时候，会根据切入点表达式的规则，会符合拦截规则的方法所在的类生成代理对象！

使用Aop开发步骤：

1. 引入aop 相关 jar文件

(aspectj 在spring之前，面向切面开发的公用组件)

aopalliance.jar 【spring-framework-2.5.6\lib\aopalliance】

aspectjrt.jar 【spring-framework-2.5.6\lib\aspectj】

aspectjweaver.jar 【spring-framework-2.5.6\lib\aspectj】

spring-aop-3.2.5.RELEASE.jar 【Spring3.2源码】

2. 引入aop名称空间

|  |
| --- |
| <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:**aop**=*"http://www.springframework.org/schema/aop"*  xsi:schemaLocation=*"*  *http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop.xsd"*>    <!-- 开启注解扫描 -->  <context:component-scan base-package=*"cn.itcast.e\_aop\_anno"*></context:component-scan>  <!-- 开启aop注解 -->  <aop:aspectj-autoproxy></aop:aspectj-autoproxy>    </beans> |

3. 开启aop注解

<aop:aspectj-autoproxy></aop:aspectj-autoproxy>

4. 使用Aop相关注解

@Aspect 指定一个类为切面类

（切面类也需要实例化）

（切面类中的方法，也叫做通知）

@Before 前置通知 【在执行目标对象方法之前执行】

@After 后置通知 【在执行目标对象方法之后执行】

@AfterReturning 返回后通知 【在执行目标对象方法结束后执行， 出现异常不执行】

@AfterThrowing 异常通知 【在执行目标对象方法出现异常时候执行】

@Around 环绕通知 【环绕目标方法执行】

@Pointcut 定义一个切入点表达式变量 （后面使用这个切入点表达式的时候，直接引用方法名即可）

Spring生成代理对象的过程？

1. 创建容器对象的时候， 根据“切入点表达式”拦截的类，生成代理对象；

2. 如果目标对象有实现接口，使用jdk代理！

3. 如果目标对象没有实现接口，使用cglib代理！

4. 从容器获取代理后的对象

5. 执行代理对象的方法，在运行时期，动态植入“切面”类中的“通知”!

## aop编程, XML配置方式

步骤：

1. 引入aop 相关jar文件

2. bean.xml 引入aop名称空间

3. Aop配置

|  |
| --- |
| <?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:context=*"http://www.springframework.org/schema/context"*  xmlns:aop=*"http://www.springframework.org/schema/aop"*  xsi:schemaLocation=*"*  *http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop.xsd"*>    <!-- dao实例加入容器 -->  <bean id=*"userDao"* class=*"cn.itcast.f\_aop\_xml.UserDao"*></bean>    <!-- 实例化切面类 -->  <bean id=*"aop"* class=*"cn.itcast.f\_aop\_xml.TransactionAop"*></bean>    <!-- Aop相关配置 -->  <aop:config>  <!-- 切入点表达式定义 -->  <aop:pointcut expression=*"execution(\* cn.itcast.f\_aop\_xml.UserDao.\*(..))"* id=*"pt"*/>    <!-- 切面配置 -->  <aop:aspect ref=*"aop"*>    <!-- 【环绕通知】 -->  <aop:around method=*"arroud"* pointcut-ref=*"pt"*/>    <!-- 【前置通知】 在目标方法之前执行 -->  <aop:before method=*"beginTransaction"* pointcut-ref=*"pt"* />    <!-- 【后置通知】 -->  <aop:after method=*"commit"* pointcut-ref=*"pt"*/>    <!-- 【返回后通知】 -->  <aop:after-returning method=*"afterReturing"* pointcut-ref=*"pt"*/>    <!-- 异常通知 -->  <aop:after-throwing method=*"afterThrowing"* pointcut-ref=*"pt"*/>    </aop:aspect>  </aop:config>    </beans> |

## 切入点表达式语法详解

切入点表达式：

拦截指定的类，生成代理对象！

execution(

modifiers-pattern? 拦截的方法的访问修饰符

ret-type-pattern 方法返回类型，必须指定

declaring-type-pattern? 拦截的方法所在的类

name-pattern(param-pattern)  **拦截的方法（以及方法的参数列表）**

throws-pattern?) 方法声明的异常

总结：

拦截，一定要指定到方法！

共性问题：

1. @Override报错

Jdk1.5 只支持父类方法的重写，不支持接口

Jdk1.6 修复这个问题

修改编译编译，改为1.6

2. xml配置没有提示

# 3. Spring 对jdbc模块的支持

学习了Spring的功能有：

Spring Core ioc容器

Spring Web 对web支持（与struts整合）

Spring Aop 面向切面编程

Spring Jdbc 对jdbc的支持

Spring 对象Jdbc的支持，

Spring 对jdbc的操作进行了简化！

提供了**JdbcTemplate**模板工具类，简化jdbc的操作！

如果想用spring jdbc功能，先引入jar文件：

spring-jdbc-3.2.5.RELEASE.jar 工具类包

spring-tx-3.2.5.RELEASE.jar 事务支持依赖包

连接池、数据库驱动包！

开发步骤：

1. 原始的jdbc操作代码

2. 对连接管理

3. 对jdbc操作进行简化

🡪 JdbcTemplate 工具类Api

🡪 JdbcTemplate与DataSource依赖关系

重点：

1. Aop, xml配置方式实现

2. Aop, 注解方式实现

-----------------------

3. 三种代理模式，练习

4. Spring 对jdbc支持