# Spring框架整合JUnit单元测试

**1. 为了简化了JUnit的测试，使用Spring框架也可以整合测试**

**2. 具体步骤**

要求：必须先有JUnit的环境（即已经导入了JUnit4的开发环境）！！

步骤一：在程序中引入:spring-test.jar

步骤二：在具体的测试类上添加注解

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration("classpath:configs/applicationContext.xml")

public class SpringDemo1 {

@Resource(name="userService")

private UserService userService;

@Test

public void demo2(){

userService.save();

}

}

# AOP

## 什么是AOP

1、在软件业，AOP为Aspect Oriented Programming的缩写，意为：面向切面编程，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。AOP是OOP（面向对象编程）的延续，是软件开发中的一个热点，也是Spring框架中的一个重要内容，是函数式编程的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

1. AOP采取**横向抽取**机制，取代了传统**纵向继承**体系重复性代码
2. 经典应用：事务管理、性能监视、安全检查、缓存 、日志等
3. Spring AOP使用纯Java实现，不需要专门的编译过程和类加载器，在运行期通过代理方式向目标类织入增强代码对程序进行增强（不修改源码的情况下）
4. AspectJ是一个基于Java语言的**AOP框架**，Spring2.0开始，Spring AOP引入对Aspect的支持，AspectJ扩展了Java语言，提供了一个专门的编译器，在编译时提供横向代码的织入

## AOP的底层实现:

1. Srping框架的AOP技术底层也是采用的代理技术，代理的方式提供了两种

1. 基于JDK的动态代理

必须是面向接口的，只有实现了具体接口的类才能生成代理对象

2. 基于CGLIB动态代理

对于没有实现了接口的类，也可以产生代理，产生这个类的子类的方式

2. Spring的传统AOP中根据类是否实现接口，来采用不同的代理方式

1. 如果实现类接口，使用JDK动态代理完成AOP

2. 如果没有实现接口，采用CGLIB动态代理完成AOP

# 动态代理

## 代理模式

什么是代理模式及其作用

Proxy Pattern（即：代理模式），23种常用的面向对象软件的设计模式之一

代理模式的定义：为其他对象提供一种代理以控制对这个对象的访问。在某些情况下，一个对象不适合或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。

**优点:**

(1).职责清晰

真实的角色就是实现实际的业务逻辑，不用关心其他非本职责的事务，通过后期的代理完成一件完成事务，附带的结果就是编程简洁清晰。

(2).代理对象可以在客户端和目标对象之间起到中介的作用，这样起到了的作用和保护了目标对象的作用。

(3).高扩展性

**结构**

一个是真正的你要访问的对象(目标类)，另一个是代理对象,真正对象与代理

对象实现同一个接口,先访问代理类再访问真正要访问的对象。

代理模式简单案例

## JDK动态代理

Java.lang.reflect.Proxy类可以直接生成一个代理对象

**Proxy.newProxyInstance():**产生代理类的实例。仅能代理实现至少一个接口的类

ClassLoader：类加载器。固定写法，和被代理类使用相同的类加载器即可。

Class[] interface：代理类要实现的接口。固定写法，和被代理类使用相同的接口即可。

InvocationHandler：策略（方案）设计模式的应用。如何代理？

**InvocationHandler中的invoke方法：**调用代理类的任何方法，此方法都会执行

Object proxy:代理对象本身的引用。一般用不着。

Method method:当前调用的方法。

Object[] args:当前方法用到的参数

1.目标类：接口 + 实现类

2.工厂类：编写工厂生成代理

4.测试

**1目标类**

|  |
| --- |
| **public** **interface** UserService {    **public** **void** addUser();  **public** **void** updateUser();  **public** **void** deleteUser();  } |

**2、切面类**

|  |
| --- |
| **public** **class** MyAspect {    **public** **void** before(){  System.*out*.println(" 前");  }    **public** **void** after(){  System.*out*.println("后");  }  } |

**3、工厂**

|  |
| --- |
| **public** **class** MyBeanFactory {    **public** **static** UserService createService(){  //1 目标类  **final** UserService userService = **new** UserServiceImpl();  //2切面类  **final** MyAspect myAspect = **new** MyAspect();  /\* 3 代理类：将目标类（切入点）和 切面类（通知） 结合 --> 切面  \* Proxy.newProxyInstance  \* 参数1：loader ，类加载器，动态代理类 运行时创建，任何类都需要类加载器将其加载到内存。  \* 一般情况：当前类.class.getClassLoader();  \* 目标类实例.getClass().get...  \* 参数2：Class[] interfaces 代理类需要实现的所有接口  \* 目标类实例.getClass().getInterfaces() ;注意：只能获得自己接口，不能获得父元素接口  \*  \* 参数3：InvocationHandler 处理类，接口，必须进行实现类，一般采用匿名内部  \* 提供 invoke 方法，代理类的每一个方法执行时，都将调用一次invoke  \* 参数：Object proxy ：代理对象  \* 参数：Method method : 代理对象当前执行的方法的描述对象（反射）  \* 执行方法名：method.getName()  \* 执行方法：method.invoke(对象，实际参数)  \* 参数：Object[] args :方法实际参数  \*  \*/  UserService proxService = (UserService)Proxy.*newProxyInstance*(  MyBeanFactory.**class**.getClassLoader(),  userService.getClass().getInterfaces(),  **new** InvocationHandler() {    @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {    //前执行  myAspect.before();    //执行目标类的方法  Object obj = method.invoke(userService, args);    //后执行  myAspect.after();    **return** obj;  }  });    **return** proxService;  }  } |

**4、测试**

|  |
| --- |
| @Test  **public** **void** demo01(){  UserService userService = MyBeanFactory.*createService*();  userService.addUser();  userService.updateUser();  userService.deleteUser();  } |

## CGLIB字节码增强

有spring的核心包就行啦

核心类：Enhancer

* 没有接口，只有实现类。
* 采用字节码增强框架 cglib，在运行时 创建目标类的子类，从而对目标类进行增强。

1. 获得代理类的核心类Enhancer对象
2. 设置父类（目标类），setSuperclass（）方法，底层是创建目标类的子类
3. 设置回调函数enhancer.setCallbac(new MethodInterceptor())
4. 创建代理对象 create()方法

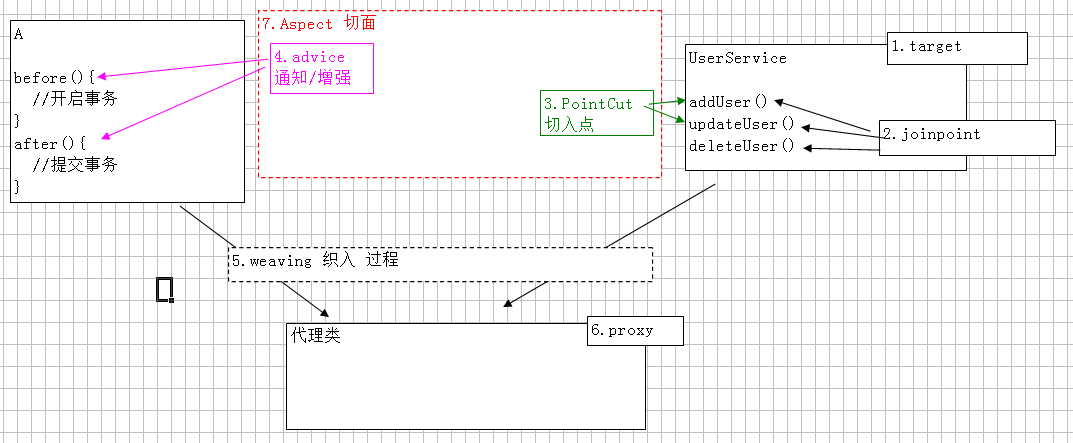
**工厂类**

|  |
| --- |
| **public** **class** MyBeanFactory {    **public** **static** UserServiceImpl createService(){  //1 目标类  **final** UserServiceImpl userService = **new** UserServiceImpl();  //2切面类  **final** MyAspect myAspect = **new** MyAspect();  // 3.代理类 ，采用cglib，底层创建目标类的子类  //3.1 核心类  Enhancer enhancer = **new** Enhancer();  //3.2 确定父类  enhancer.setSuperclass(userService.getClass());  /\* 3.3 设置回调函数 , MethodInterceptor接口 等效 jdk InvocationHandler接口  \* intercept() 等效 jdk invoke()  \* 参数1、参数2、参数3：以invoke一样  \* 参数4：methodProxy 方法的代理  \*  \*  \*/  enhancer.setCallback(**new** MethodInterceptor(){  @Override  **public** Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) **throws** Throwable {    //前  myAspect.before();    //执行目标类的方法  Object obj = method.invoke(userService, args);  // \* 执行代理类的父类 ，执行目标类 （目标类和代理类 父子关系）  methodProxy.invokeSuper(proxy, args);    //后  myAspect.after();    **return** obj;  }  });  //3.4 创建代理  UserServiceImpl proxService = (UserServiceImpl) enhancer.create();    **return** proxService;  }  } |

# AOP的开发中的相关术语

|  |
| --- |
| 1. Joinpoint(连接点) -- 所谓连接点是指那些被拦截到的点。在spring中,这些点指的是方法,因为spring只支持方法类型的连接点  2. Pointcut(切入点) -- 所谓切入点是指我们要对哪些Joinpoint进行拦截的定义  3. Advice(通知/增强) -- 所谓通知是指拦截到Joinpoint之后所要做的事情就是通知.通知分为前置通知,后置通知,异常通知,最终通知,环绕通知(切面要完成的功能)  4. Introduction(引介) -- 引介是一种特殊的通知在不修改类代码的前提下, Introduction可以在运行期为类动态地添加一些方法或Field  5. Target(目标对象) -- 代理的目标对象  6. Weaving(织入) -- 是指把增强应用到目标对象来创建新的代理对象的过程  7. Proxy（代理） -- 一个类被AOP织入增强后，就产生一个结果代理类  8. Aspect(切面) -- 是切入点和通知的结合，以后咱们自己来编写和配置的 |

看图理解：



# AOP联盟通知类型

|  |
| --- |
| * AOP联盟为通知Advice定义了org.aopalliance.aop.Advice * Spring按照通知Advice在目标类方法的连接点位置，可以分为5类   + 前置通知 org.springframework.aop.MethodBeforeAdvice     - 在目标方法执行前实施增强   + 后置通知 org.springframework.aop.AfterReturningAdvice     - 在目标方法执行后实施增强   + 环绕通知 org.aopalliance.intercept.MethodInterceptor     - 在目标方法执行前后实施增强   + 异常抛出通知 org.springframework.aop.ThrowsAdvice     - 在方法抛出异常后实施增强   + 引介通知 org.springframework.aop.IntroductionInterceptor     - 在目标类中添加一些新的方法和属性 |

|  |
| --- |
| 环绕通知，必须手动执行目标方法  try{  //前置通知  //执行目标方法  //后置通知  } catch(){  //抛出异常通知  } |

# spring aop编程：【掌握】

导包：spring的传统AOP的开发的包

\* spring-aop-4.2.4.RELEASE.jar

\* com.springsource.org.aopalliance-1.0.0.jar

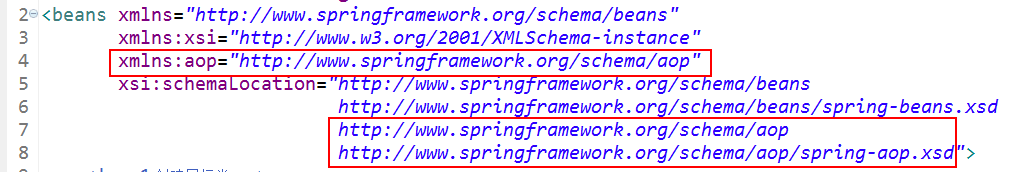
\* aspectJ的开发包

\* com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar

\* spring-aspects-4.2.4.RELEASE.jar

* 从spring容器获得目标类，如果配置aop，spring将自动生成代理。
* 要确定目标类，aspectj 切入点表达式，导入jar包

### spring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: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/aop*  *http://www.springframework.org/schema/aop/spring-aop.xsd"*>  <!-- 1 创建目标类 -->  <bean id=*"userServiceId"* class=*"com.qf.c\_spring\_aop.UserServiceImpl"*></bean>  <!-- 2 创建切面类（通知） -->  <bean id=*"myAspectId"* class=*"com.qf.c\_spring\_aop.MyAspect"*></bean>  <!-- 3 aop编程  3.1 导入命名空间  3.2 使用 <aop:config>进行配置  proxy-target-class="true" 声明时使用cglib代理  <aop:pointcut> 切入点 ，从目标对象获得具体方法  <aop:advisor> 特殊的切面，只有一个通知 和 一个切入点  advice-ref 通知引用  pointcut-ref 切入点引用  3.3 切入点表达式  execution(\* com.qf.c\_spring\_aop.\*.\*(..))  选择方法 返回值任意 包 类名任意 方法名任意 参数任意    -->  <aop:config proxy-target-class=*"true"*>  <aop:pointcut expression=*"execution(\* com.qf.c\_spring\_aop.\*.\*(..))"* id=*"myPointCut"*/>  <aop:advisor advice-ref=*"myAspectId"* pointcut-ref=*"myPointCut"*/>  </aop:config>  </beans> |

### 测试

|  |
| --- |
| @Test  **public** **void** demo01(){  String xmlPath = "com/qf/c\_spring\_aop/beans.xml";  ApplicationContext applicationContext = **new** ClassPathXmlApplicationContext(xmlPath);    //获得目标类  UserService userService = (UserService) applicationContext.getBean("userServiceId");  userService.addUser();  userService.updateUser();  userService.deleteUser();  } |

# AspectJ

## 介绍

* AspectJ是一个基于Java语言的AOP框架
* Spring2.0以后新增了对AspectJ切点表达式支持
* @AspectJ 是AspectJ1.5新增功能，通过JDK5注解技术，允许直接在Bean类中定义切面

新版本Spring框架，建议使用AspectJ方式来开发AOP

* 主要用途：自定义开发

## 切入点表达式【掌握】

1.execution() 用于描述方法 【掌握】

语法：execution(修饰符 返回值 包.类.方法名(参数) throws异常)

修饰符，一般省略

public 公共方法

\* 任意

返回值，不能省略

void 返回没有值

String 返回值字符串

\* 任意

包，[省略]

com.qf.crm 固定包

com.qf.crm.\*.service crm包下面子包任意 （例如：com.qf.crm.service）

com.qf.crm.. crm包下面的所有子包（含自己）

com.qf.crm.\*.service.. crm包下面任意子包，固定目录service，service目录任意包

类，[省略]

UserServiceImpl 指定类

\*Impl 以Impl结尾

User\* 以User开头

\* 任意

方法名，不能省略

addUser 固定方法

add\* 以add开头

\*Do 以Do结尾

\* 任意

(参数)

() 无参

(int) 一个整型

(int ,int) 两个

(..) 参数任意

throws ,可省略，一般不写。

**execution(\* com.qf.crm.\*.service..\*.\*(..))**

## AspectJ 通知类型

* aop联盟定义通知类型，具有特性接口，必须实现，从而确定方法名称。
* aspectj 通知类型，只定义类型名称。和方法格式。
* 个数：6种，知道5种，掌握1中。

before:前置通知(应用：各种校验)

在方法执行前执行，如果通知抛出异常，阻止方法运行

afterReturning:后置通知(应用：常规数据处理)

方法正常返回后执行，如果方法中抛出异常，通知无法执行

必须在方法执行后才执行，所以可以获得方法的返回值。

around:环绕通知(应用：十分强大，可以做任何事情)

方法执行前后分别执行，可以阻止方法的执行

必须手动执行目标方法

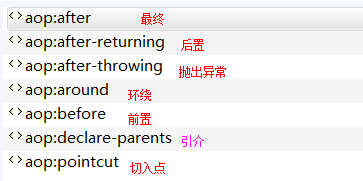
afterThrowing:抛出异常通知(应用：包装异常信息)

方法抛出异常后执行，如果方法没有抛出异常，无法执行

after:最终通知(应用：清理现场)

方法执行完毕后执行，无论方法中是否出现异常

|  |
| --- |
| 环绕  try{  //前置：before  //手动执行目标方法  //后置：afterRetruning  } catch(){  //抛出异常 afterThrowing  } finally{  //最终 after  } |



## 基于xml

### 1. 步骤一：创建JavaWEB项目，引入具体的开发的jar包

\* 先引入Spring框架开发的基本开发包

\* 再引入Spring框架的AOP的开发包

\* spring的传统AOP的开发的包

\* spring-aop-4.2.4.RELEASE.jar

\* com.springsource.org.aopalliance-1.0.0.jar

\* aspectJ的开发包

\* com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar

\* spring-aspects-4.2.4.RELEASE.jar

### 2. 步骤二：创建Spring的配置文件，引入具体的AOP的schema约束

|  |
| --- |
| <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  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/aop http://www.springframework.org/schema/aop/spring-aop.xsd"> |

### 步骤三：编写切面类

|  |
| --- |
| /\*\*  \* 切面类，含有多个通知  \*/  **public** **class** MyAspect {    **public** **void** myBefore(JoinPoint joinPoint){  System.*out*.println("前置通知 ： " + joinPoint.getSignature().getName());  }    **public** **void** myAfterReturning(JoinPoint joinPoint,Object ret){  System.*out*.println("后置通知 ： " + joinPoint.getSignature().getName() + " , -->" + ret);  }    **public** Object myAround(ProceedingJoinPoint joinPoint) **throws** Throwable{  System.*out*.println("前");  //手动执行目标方法  Object obj = joinPoint.proceed();    System.*out*.println("后");  **return** obj;  }    **public** **void** myAfterThrowing(JoinPoint joinPoint,Throwable e){  System.*out*.println("抛出异常通知 ： " + e.getMessage());  }    **public** **void** myAfter(JoinPoint joinPoint){  System.*out*.println("最终通知");  }  } |

### 步骤四：spring配置

**编写目标类，切面类 aop的配置**

|  |
| --- |
| <!-- 1 创建目标类 -->  <bean id=*"userServiceId"* class=*"com.qf.d\_aspect.a\_xml.UserServiceImpl"*></bean>  <!-- 2 创建切面类（通知） -->  <bean id=*"myAspectId"* class=*"com.qf.d\_aspect.a\_xml.MyAspect"*></bean>  <!-- 3 aop编程  <aop:aspect> 将切面类 声明“切面”，从而获得通知（方法）  ref 切面类引用  <aop:pointcut> 声明一个切入点，所有的通知都可以使用。  expression 切入点表达式  id 名称，用于其它通知引用  -->  <aop:config>  <aop:aspect ref=*"myAspectId"*>  <aop:pointcut expression=*"execution(\* com.qf.d\_aspect.a\_xml.UserServiceImpl.\*(..))"* id=*"myPointCut"*/>    <!-- 3.1 前置通知  <aop:before method="" pointcut="" pointcut-ref=""/>  method : 通知，及方法名  pointcut :切入点表达式，此表达式只能当前通知使用。  pointcut-ref ： 切入点引用，可以与其他通知共享切入点。  通知方法格式：public void myBefore(JoinPoint joinPoint){  参数1：org.aspectj.lang.JoinPoint 用于描述连接点（目标方法），获得目标方法名等  例如：  <aop:before method="myBefore" pointcut-ref="myPointCut"/>  -->    <!-- 3.2后置通知 ,目标方法后执行，获得返回值  <aop:after-returning method="" pointcut-ref="" returning=""/>  returning 通知方法第二个参数的名称  通知方法格式：public void myAfterReturning(JoinPoint joinPoint,Object ret){  参数1：连接点描述  参数2：类型Object，参数名 returning="ret" 配置的  例如：  <aop:after-returning method="myAfterReturning" pointcut-ref="myPointCut" returning="ret" />  -->    <!-- 3.3 环绕通知  <aop:around method="" pointcut-ref=""/>  通知方法格式：public Object myAround(ProceedingJoinPoint joinPoint) throws Throwable{  返回值类型：Object  方法名：任意  参数：org.aspectj.lang.ProceedingJoinPoint  抛出异常  执行目标方法：Object obj = joinPoint.proceed();  例如：  <aop:around method="myAround" pointcut-ref="myPointCut"/>  -->  <!-- 3.4 抛出异常  <aop:after-throwing method="" pointcut-ref="" throwing=""/>  throwing ：通知方法的第二个参数名称  通知方法格式：public void myAfterThrowing(JoinPoint joinPoint,Throwable e){  参数1：连接点描述对象  参数2：获得异常信息，类型Throwable ，参数名由throwing="e" 配置  例如：  <aop:after-throwing method="myAfterThrowing" pointcut-ref="myPointCut" throwing="e"/>  -->  <!-- 3.5 最终通知 -->  <aop:after method=*"myAfter"* pointcut-ref=*"myPointCut"*/>        </aop:aspect>  </aop:config> |

### 步骤五：完成测试

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.**class**)  @ContextConfiguration("classpath:com/qf/test/beans.xml")  **public** **class** TestAop {  @Resource(name="userDao")  **private** UserDao userDao;  @Test  **public** **void** run1(){  userDao.add();  }  } |