Spring

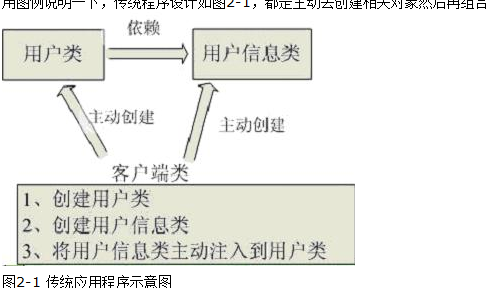
1. **Ioc**

Ioc—Inversion of Control，即“控制反转”，不是什么技术，而是一种设计思想。在Java开发中，Ioc意味着将你设计好的对象交给容器控制，而不是传统的在你的对 象内部直接控制。如何理解好Ioc呢？理解好Ioc的关键是要明确“谁控制谁，控制什么，为何是反转（有反转就应该有正转了），哪些方面反转了”，那我们 来深入分析一下：

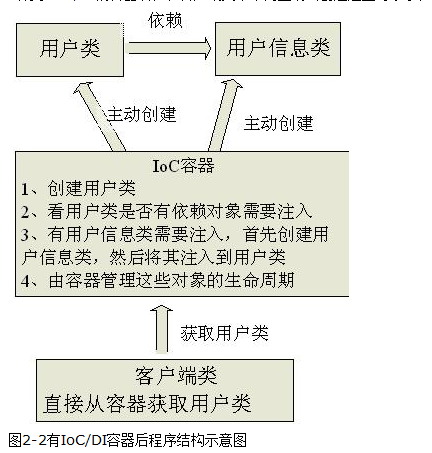
●**谁控制谁，控制什么：**传 统Java SE程序设计，我们直接在对象内部通过new进行创建对象，是程序主动去创建依赖对象；而IoC是有专门一个容器来创建这些对象，即由Ioc容器来控制对象的创建；谁控制谁？当然是IoC 容器控制了对象；控制什么？那就是主要控制了外部资源获取（不只是对象包括比如文件等）。

●**为何是反转，哪些方面反转了：**有反转就有正转，传统应用程序是由我们自己在对象中主动控制去直接获取依赖对象，也就是正转；而反转则是由容器来帮忙创建及注入依赖对象；为何是反转？因为由容器帮我们查找及注入依赖对象，对象只是被动的接受依赖对象，所以是反转；哪些方面反转了？依赖对象的获取被反转了。

用图例说明一下，传统程序设计如图2-1，都是主动去创建相关对象然后再组合起来：



当有了IoC/DI的容器后，在客户端类中不再主动去创建这些对象了，如图2-2所示:



**1.1.2  IoC能做什么**

IoC 不是一种技术，只是一种思想，一个重要的面向对象编程的法则，它能指导我们如何设计出松耦合、更优良的程序。传统应用程序都是由我们在类内部主动创建依赖对象，从而导致类与类之间高耦合，难于测试；有了IoC容器后，把创建和查找依赖对象的控制权交给了容器，由容器进行注入组合对象，所以对象与对象之间是松散耦合，这样也方便测试，利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活。

其实IoC对编程带来的最大改变不是从代码上，而是从思想上，发生了“主从换位”的变化。应用程序原本是老大，要获取什么资源都是主动出击，但是在IoC/DI思想中，应用程序就变成被动的了，被动的等待IoC容器来创建并注入它所需要的资源了。

IoC很好的体现了面向对象设计法则之一—— 好莱坞法则：“别找我们，我们找你”；即由IoC容器帮对象找相应的依赖对象并注入，而不是由对象主动去找。

**1.1.3  IoC和DI**

DI—Dependency Injection，即“依赖注入”：是组件之间依赖关系由容器在运行期决定，形象的说，即由容器动态的将某个依赖关系注入到组件之中。依赖注入的目的并非为软件系统带来更多功能，而是为了提升组件重用的频率，并为系统搭建一个灵活、可扩展的平台。通过依赖注入机制，我们只需要通过简单的配置，而无需任何代码就可指定目标需要的资源，完成自身的业务逻辑，而不需要关心具体的资源来自何处，由谁实现。

理解DI的关键是：“谁依赖谁，为什么需要依赖，谁注入谁，注入了什么”，那我们来深入分析一下：

●**谁依赖于谁：**当然是应用程序依赖于IoC容器；

●**为什么需要依赖：**应用程序需要IoC容器来提供对象需要的外部资源；

●**谁注入谁：**很明显是IoC容器注入应用程序某个对象，应用程序依赖的对象；

**●注入了什么：**就是注入某个对象所需要的外部资源（包括对象、资源、常量数据）。

IoC 和DI由什么关系呢？其实它们是同一个概念的不同角度描述，由于控制反转概念比较含糊（可能只是理解为容器控制对象这一个层面，很难让人想到谁来维护对象关系），所以2004年大师级人物Martin Fowler又给出了一个新的名字：“依赖注入”，相对IoC 而言，**“依赖注入”明确描述了“被注入对象依赖IoC容器配置依赖对象”。**

在平时的java应用开发中，我们要实现某一个功能或者说是完成某个业务逻辑时至少需要两个或以上的对象来协作完成，在没有使用Spring的时候，每个对象在需要使用他的合作对象时，自己均要使用像new object() 这样的语法来将合作对象创建出来，这个合作对象是由自己主动创建出来的，创建合作对象的主动权在自己手上，自己需要哪个合作对象，就主动去创建，创建合作对象的主动权和创建时机是由自己把控的，而这样就会使得对象间的耦合度高了，A对象需要使用合作对象B来共同完成一件事，A要使用B，那么A就对B产生了依赖，也就是A和B之间存在一种耦合关系，并且是紧密耦合在一起，而使用了Spring之后就不一样了，创建合作对象B的工作是由Spring来做的，Spring创建好B对象，然后存储到一个容器里面，当A对象需要使用B对象时，Spring就从存放对象的那个容器里面取出A要使用的那个B对象，然后交给A对象使用，至于Spring是如何创建那个对象，以及什么时候创建好对象的，A对象不需要关心这些细节问题(你是什么时候生的，怎么生出来的我可不关心，能帮我干活就行)，A得到Spring给我们的对象之后，两个人一起协作完成要完成的工作即可。

　　所以**控制反转IoC(Inversion of Control)是说创建对象的控制权进行转移，以前创建对象的主动权和创建时机是由自己把控的，而现在这种权力转移到第三方**，比如转移交给了IoC容器，它就是一个专门用来创建对象的工厂，你要什么对象，它就给你什么对象，有了 IoC容器，依赖关系就变了，原先的依赖关系就没了，它们都依赖IoC容器了，通过IoC容器来建立它们之间的关系。

　　这是我对Spring的IoC**(控制反转)**的理解。DI**(依赖注入)**其实就是IOC的另外一种说法，DI是由Martin Fowler 在2004年初的一篇论文中首次提出的。他总结：**控制的什么被反转了？就是：获得依赖对象的方式反转了。**

IOC即inverse of control 控制反转，以前对象之间的引用是通过new来调用实现，有了Spring IOC,我们可以把对象之间的引用交给他来管理，这样就把控制权交给了Spring，所以就叫做控制反转。

Spring IOC的实现用到了设计模式：简单工厂，他也是从简单工厂进化而来的，下面我们看看Spring的IOC是如何进化来的。

简单工厂模式实现：

package org;

//抽象接口

interface Fruit{

public void eat();

}

//实现类A

class Apple implements Fruit{

public void eat(){

System.out.println("吃苹果。");

}

}

//实现类B

class Orange implements Fruit{

public void eat(){

System.out.println("吃橘子");

}

}

//工厂类

class Factory{

public static Fruit getInstance(String className){

Fruit f=null;

if(className.equals("apple")){

f=new Apple();

}

if(className.endsWith("orange")){

f=new Orange();

}

return f;

}

}

public class FactoryDemo02 {

public static void main(String args[]){

Fruit f=Factory.getInstance("apple");

f.eat();

}

}

反射+简单工厂

但是工厂类如果这样写的话，就有一个问题，如果增加了水果，比如香蕉，那么在工厂类里面也要进行相关的修改了，这样不合理，而java的反射机制可以解决这个问题

package org1;

interface Fruit {

public void eat();

}

class Apple implements Fruit {

public void eat() {

System.out.println("吃苹果。");

}

}

class Orange implements Fruit {

public void eat() {

System.out.println("吃橘子");

}

}

class Factory {

public static Fruit getInstance(String className) {

Fruit f = null;

try {

f = (Fruit) Class.forName(className).newInstance();

} catch (Exception e) {

e.printStackTrace();

}

return f;

}

}

public class CopyOfFactoryDemo03 {

public static void main(String args[]) {

Fruit f = Factory.getInstance("org1.Apple");

f.eat();

}

}

反射+简单工厂+xml配置文件

利用java的反射机制，就能动态的实例化各种类了。 但是这个程序还是存在一个问题，就是主函数这里需要填入一个完整的类名称，不够方便，所以要增加配置文件来简化

package org3;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.util.Properties;

interface Fruit {

public void eat();

}

class Apple implements Fruit {

public void eat() {

System.out.println("吃苹果。");

}

}

class Orange implements Fruit {

public void eat() {

System.out.println("吃橘子");

}

}

class Factory {

public static Fruit getInstance(String className) {

Fruit f = null;

try {

f = (Fruit) Class.forName(className).newInstance();

} catch (Exception e) {

e.printStackTrace();

}

return f;

}

}

class PropertiesOperate{

private Properties pro=null;

private File file=new File("d:"+File.separator+"fruit.properties");

public PropertiesOperate(){

this.pro=new Properties();

if(file.exists()){

try {

pro.loadFromXML(new FileInputStream(file));

} catch (Exception e) {

e.printStackTrace();

}

}else{

this.save();

}

}

private void save(){

this.pro.setProperty("apple","org3.Apple");

this.pro.setProperty("orange", "org3.Orange");

try {

this.pro.storeToXML(new FileOutputStream(this.file),"Fruit");

} catch (Exception e) {

e.printStackTrace();

}

}

public Properties getProperties(){

return this.pro;

}

}

public class CopyOfFactoryDemo04 {

public static void main(String args[]) {

Properties pro=new PropertiesOperate().getProperties();

Fruit f= Factory.getInstance(pro.getProperty("apple"));

f.eat();

}

}

终极版本Spring IOC

加入配置文件问题就解决了，以后如果要增加新的水果类，都要在这个配置文件里面登记。这时我们可以说配置文件可以控制程序的执行，现在看起来有点像spring的ioc了。下面我们来看看Spring IOC是如何实现的。

package test2;

public class Person {

private String name;

private int age;

private Grade grade;

public String getName() {

return name;

}

public Grade getGrade() {

return grade;

}

public void setGrade(Grade grade) {

this.grade = grade;

}

public void setName(String name) {

this.name = name;

}

public void setAge(int age) {

this.age = age;

}

public int getAge() {

return age;

}

public int getTotleGrade() {

return grade.getEnglish()+grade.getMath();

}

}

package test2;

public class Grade {

private int math;

private int english;

public int getMath() {

return math;

}

public void setMath(int math) {

this.math = math;

}

public int getEnglish() {

return english;

}

public void setEnglish(int english) {

this.english = english;

}

}

Bean.xml配置文件（该文件只要放在test2包里面就好了）

//很多豆豆

//第一个豆豆，是一个Person类，id名字随便取，还要写上类的全名

//下面开始把这个类里面的所有属性列出来，并赋值，至于你说难道一定要赋值吗？我想可以，我刚学，不知道

小龙//这里的名字是通过程序里面的set来赋值的，不信你去掉程序里面相关的set，就出错了

23//这里有点特别，这个grade变量是一个对象，和一般的变量要区别对待

//这里指向了本配置文件里面一个名字叫Grade(即id=Grade)的bean

//同上

9959

Test类

package test2;

import org.springframework.beans.factory.BeanFactory;

import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;

import org.springframework.core.io.Resource;

import test.ExampleBean;

public class Test {

public static void main(String args[]){

Resource input = new ClassPathResource("test2/Bean.xml");//Bean.xml的路径

System.out.println("resource is:" + input);

BeanFactory factory = new XmlBeanFactory(input);//把input扔到工厂里面去，这个工厂就能为你提供实例了（我也不知道能不能这样说）

Person person =(Person) factory.getBean("Person");//你要一个叫Person的东西，那好，工厂就去找“Person"给你

Grade grade=(Grade)factory.getBean("Grade");

System.out.println("姓名："+person.getName());//person可以调用里面相关的方法，就相当于new了一个Person一样

System.out.println("年龄："+person.getAge());

System.out.println("数学成绩："+grade.getMath());

System.out.println("英语成绩："+grade.getEnglish());

System.out.println("数学，英语总成绩："+person.getTotleGrade());

}

}

如此看来，你在对比一开始的那个水果的程序，你会发现，spring配置文件，还是一个工厂，只不过换种形式一样，他管理所有的类，新建的类要到工厂里面去登记，不然就不能被主程序用，这就是为什么说ioc就是工厂模式的升级版。至于配置文件的书写，就跟堆积木一样。

--------------------------------- 顺便提下，关于Spring读取配置文件的方法： applicationcontext--- FileSystemXmlApplicationContext---这个方法是从文件绝对路径加载配置文

ClassPathXmlApplicationContext---这个方法是从classpath下加载配置文件(适合于相对路径方式加载) XmlWebApplicationContext----专为web工程定制的方法，推荐Web项目中使用。

beanfactory--- ClassPathResource --- 从系统的类路径中加载

FileSystemResource --- 从文件系统加载，比如说自己指定配置文件的全路径

InputStreamResource --- 从输入流中加载

ServletContextResource --- 从Servlet 上下文环境中加载

UrlResource --- 从指定的Url加载

--------------------------------------------- BeanFactory和ApplicationContext的区别 ApplicationContext接口,它由BeanFactory接口派生而来，因而提供BeanFactory所有的功能。ApplicationContext以一种更向面向框架的方式工作以及对上下文进行分层和实现继承，ApplicationContext包还提供了以下的功能：

? MessageSource, 提供国际化的消息访问

? 资源访问，如URL和文件

? 事件传播

? 载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的web层

最主要的就是BeanFactory延迟加载，当使用到getBean的时候才会抛异常，而ApplicationContext在刚开始启动加载的时候就会抛出异常，这样有利于检查所依赖属性是否注入；所以通常情况下我们选择使用ApplicationContext。

1. **AOP**

 AllLogAdvice类代码变成了如下：

**Java代码  [收藏代码](javascript:void())**

1. **package** com.aop;
3. **import** org.apache.log4j.Logger;
4. **import** org.aspectj.lang.JoinPoint;
5. **import** org.aspectj.lang.ProceedingJoinPoint;
6. **import** org.aspectj.lang.annotation.After;
7. **import** org.aspectj.lang.annotation.AfterReturning;
8. **import** org.aspectj.lang.annotation.AfterThrowing;
9. **import** org.aspectj.lang.annotation.Around;
10. **import** org.aspectj.lang.annotation.Aspect;
11. **import** org.aspectj.lang.annotation.Before;
12. **import** org.aspectj.lang.annotation.Pointcut;
14. @Aspect
15. **public** **class** AllLogAdvice {
16. **private** Logger logger = Logger.getLogger(AllLogAdvice.**class**);
18. // @Pointcut("execution(\* com.test.spring.aop.pointcutexp..JoinPointObjP2.\*(..))")
19. // @Pointcut("within(com.test.spring.aop.pointcutexp..\*)")
20. // @Pointcut("this(com.test.spring.aop.pointcutexp.Intf)")
21. // @Pointcut("target(com.test.spring.aop.pointcutexp.Intf)")
22. // @Pointcut("@within(org.springframework.transaction.annotation.Transactional)")
23. // @Pointcut("@annotation(org.springframework.transaction.annotation.Transactional)")
24. // @Pointcut("args(String)")
25. @Pointcut("execution(\* com.service.UserService.add\*(..)) || execution(\* com.service.UserService.delete\*(..))")
26. **public** **void** pointcut1() {
27. // 定义一个pointcut，下面用Annotation标注的通知方法可以公用这个pointcut
28. }
30. // 前置通知
31. // 拦截参数为一个String类型的方法
32. @Before("pointcut1() && args(temp)")
33. **public** **void** myBeforeAdvice(String temp) {// 如果需要知道拦截的方法的信息，也可以需添加JoinPoint参数
34. String logInfoText = "这是前置通知" + temp;
35. // 将日志信息写入配置的文件中
36. logger.info(logInfoText);
37. }
39. // 后置通知
40. // 拦截 返回类型为String 的方法
41. @AfterReturning(pointcut = "pointcut1()", returning = "result")
42. **public** **void** myAfterReturnAdvice(String result) {
43. logger.info("这是后置通知  " + " result: " + result);
44. }
46. // 最终置通知
47. @After("execution(\* com.service.UserService.add\*(..))")
48. **public** **void** doAfter() {
49. logger.info("这是最终通知");
50. }
52. // 异常通知
53. @AfterThrowing(pointcut = "pointcut1()", throwing = "e")
54. **public** **void** myThrowingAdvice(JoinPoint jionpoint, Exception e) {
55. // 获取被调用的类名
56. String targetClassName = jionpoint.getTarget().getClass().getName();
57. // 获取被调用的方法名
58. String targetMethodName = jionpoint.getSignature().getName();
59. // 日志格式字符串
60. String logInfoText = "异常通知：执行" + targetClassName + "类的"
61. + targetMethodName + "方法时发生异常";
62. // 将日志信息写入配置的文件中
63. logger.info(logInfoText);
64. }
66. // 环绕通知
67. // @Around(value="pointcut1()")
68. @Around("pointcut1()")
69. **public** Object myAroundAdvice(ProceedingJoinPoint jionpoint)
70. **throws** Throwable {
71. // 获取被调用的方法名
72. String targetMethodName = jionpoint.getSignature().getName();
73. Object o = jionpoint.proceed();
74. String logInfoText = "这是环绕通知：" + targetMethodName;
75. logger.info(logInfoText);
76. //Object o = jionpoint.proceed();//注意写到这儿的话，环绕通知和其它通知的顺序
77. **return** o;
78. }
79. }

 代码里加了很多注释说明，注意这儿多了一个最终通知。aop.xml文件的内容如下：

**Xml代码  [收藏代码](javascript:void())**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans**
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:aop="http://www.springframework.org/schema/aop"
5. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6. xmlns:p="http://www.springframework.org/schema/p"
7. xsi:schemaLocation="http://www.springframework.org/schema/beans
8. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
9. http://www.springframework.org/schema/aop
10. http://www.springframework.org/schema/aop/spring-aop-2.5.xsd"**>**
12. **<bean** id="myUserService" class="com.service.UserService"**></bean>**
14. **<bean** id="allLogAdvice" class="com.aop.AllLogAdvice"**></bean>**
16. **<aop:aspectj-autoproxy** **/>**
18. <!--  如果不声明AllLogAdvice的bean，可以在com.aop下的类加上@Component
19. **<context:component-scan** base-package="com.aop"**/>**
20. --**>**
21. **</beans>**

 注意这里添加了<aop:aspectj-autoproxy />，而且声明了com.aop.AllLogAdvice的bean，如果不声明这个bean的话，在这个类的头上添加@Component，然后通过component-scan扫描也可以。

这儿我把UserService类稍微改了一下：

**Java代码  [收藏代码](javascript:void())**

1. **package** com.service;
3. **public** **class** UserService **implements** IUserService {
5. **public** **int** addUser(String name, **int** age) {
6. //省略诸如操作数据库等复杂的逻辑操作
7. System.out.println("add user "+ name +" successfully");
8. **return** 1;
9. }
11. **public** **void** deleteUser(String name) {
12. //省略诸如操作数据库等复杂的逻辑操作
13. System.out.println("deleted one user named " + name);
14. //throw new RuntimeException("这是特意抛出的异常信息！");
15. }
16. }

addUser方法改成了int型的返回值，deleteUser方法依然是void。

MainTest主测试类代码没有变化：

**Java代码  [收藏代码](javascript:void())**

1. **package** com.test;
3. **import** org.springframework.context.ApplicationContext;
4. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
6. **import** com.service.IUserService;
8. **public** **class** MainTest {
9. **public** **static** **void** main(String[] args) {
10. ApplicationContext context = **new** ClassPathXmlApplicationContext(
11. "aop.xml");
12. IUserService userService = (IUserService) context
13. .getBean("myUserService");
15. userService.addUser("ton", 56);
16. userService.deleteUser("ton");
17. }
18. }

 好啦，看看执行结果：

**Txt代码  [收藏代码](javascript:void())**

1. add user ton successfully
2. [INFO ] [14:48:43] com.aop.AllLogAdvice - 这是最终通知
3. [INFO ] [14:48:43] com.aop.AllLogAdvice - 这是环绕通知：addUser
4. [INFO ] [14:48:43] com.aop.AllLogAdvice - 这是前置通知ton
5. deleted one user named ton
6. [INFO ] [14:48:43] com.aop.AllLogAdvice - 这是后置通知   result: null
7. [INFO ] [14:48:43] com.aop.AllLogAdvice - 这是环绕通知：deleteUser

 结果发现，后置通知本来是要拦截返回值是String的方法，addUser确实没有被拦截，但是deleteUser明明是void也被拦截了。

**3、通知解耦**

上面代码虽然实现了切面通知，但是通知方法都是在代理方法中实现，这样耦合度太高，我们可以抽象出一个接口，这个接口里就只有两个方法，一个是在被代理对象要执行方法之前执行的方法，我们取名为before,第二个方法就是在被代理对象执行方法之后执行的方法，我们取名为after，接口定义如下 ：

**Java代码  [收藏代码](javascript:void())**

1. **package** com.chenzehe.aop;
3. **import** java.lang.reflect.Method;
5. **public** **interface** IOperation {
6. /\*\*
7. \* 方法执行之前的操作
8. \*
9. \* @param method
10. \*/
11. **void** before(Method method);
13. /\*\*
14. \* 方法执行之后的操作
15. \*
16. \* @param method
17. \*/
18. **void** after(Method method);
19. }

 我们去写一个实现上面接口的类.我们把作他真正的操作者,如下面是日志操作者的一个类：

**Java代码  [收藏代码](javascript:void())**

1. /\*\*
2. \* Huisou.com Inc.
3. \* Copyright (c) 2011-2012 All Rights Reserved.
4. \*/
6. **package** com.chenzehe.aop;
8. **import** java.lang.reflect.Method;
10. **import** org.slf4j.Logger;
11. **import** org.slf4j.LoggerFactory;
13. **public** **class** LoggerOperation **implements** IOperation {
14. **final** Logger    log = LoggerFactory.getLogger(LoggerOperation.**class**);
16. @Override
17. **public** **void** before(Method method) {
18. log.info("Before:" + method.getName());
19. }
21. @Override
22. **public** **void** after(Method method) {
23. log.info("After:" + method.getName());
24. }
26. }

 JDK动态代理实现类：

**Java代码  [收藏代码](javascript:void())**

1. /\*\*
2. \* Huisou.com Inc.
3. \* Copyright (c) 2011-2012 All Rights Reserved.
4. \*/
6. **package** com.chenzehe.aop;
8. **import** java.lang.reflect.InvocationHandler;
9. **import** java.lang.reflect.Method;
10. **import** java.lang.reflect.Proxy;
12. **public** **class** OperationDynamicProxy **implements** InvocationHandler {
13. /\*\*
14. \* 要处理的对象(也就是我们要在方法的前后加上业务逻辑的对象)
15. \*/
16. **private** Object  delegate;
17. /\*\*
18. \* 切面操作方法
19. \*/
20. **private** Object  operation;
22. **public** OperationDynamicProxy() {
23. }
25. /\*\*
26. \* 动态生成方法被处理过后的对象 (写法固定)
27. \*/
28. **public** Object bind(Object delegate, Object operation) {
29. **this**.delegate = delegate;
30. **this**.operation = operation;
32. **return** Proxy.newProxyInstance(**this**.delegate.getClass().getClassLoader(), **this**.delegate.getClass().getInterfaces(), **this**);
33. }
35. @Override
36. **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {
37. Object result;
38. // 反射得到操作者的实例
39. Class clazz = **this**.operation.getClass();
40. // 反射得到操作者的before方法
41. Method start = clazz.getDeclaredMethod("before", **new** Class[] { Method.**class** });
42. // 反射执行before方法
43. start.invoke(**this**.operation, **new** Object[] { method });
44. // 执行要处理对象的原本方法
45. result = method.invoke(**this**.delegate, args);
46. // 反射得到操作者的end方法
47. Method end = clazz.getDeclaredMethod("after", **new** Class[] { Method.**class** });
48. // 反射执行end方法
49. end.invoke(**this**.operation, **new** Object[] { method });
50. **return** result;
51. }
53. }

 测试类：

**Java代码  [收藏代码](javascript:void())**

1. /\*\*
2. \* Huisou.com Inc.
3. \* Copyright (c) 2011-2012 All Rights Reserved.
4. \*/
6. **package** com.chenzehe.aop;
8. **import** org.junit.Test;
10. **public** **class** OperationDynamicProxyTest {
11. @Test
12. **public** **void** testSave() {
13. IService service = (IService) **new** OperationDynamicProxy().bind(**new** Service(), **new** LoggerOperation());
14. service.save();
15. }
16. }
18. 输出：
19. LoggerOperation - Before:save
20. \*\*\*\*\*save\*\*\*\*\*
21. LoggerOperation - After:save

**4、使用CGLib实现**

上面实现AOP中被代理对象都是提供接口的，有时候我们的需求中被代理类并不提供接口，此时使用CGLib来实现，CGLib实现原理是继承被代理类，重写被代理类的的方法，然后在重写方法中添加自己的切面方法。

没有实现提供的类：

**Java代码  [收藏代码](javascript:void())**

1. /\*\*
2. \* Huisou.com Inc.
3. \* Copyright (c) 2011-2012 All Rights Reserved.
4. \*/
6. **package** com.chenzehe.aop;
8. **public** **class** ServiceNotInterface {
9. **final** Logger    log = LoggerFactory.getLogger(ServiceNotInterface.**class**);
11. **public** **int** save() {
12. log.info("\*\*\*\*\*save\*\*\*\*\*");
13. **return** 0;
14. }
15. }

 CGLib实现类：

**Java代码  [收藏代码](javascript:void())**

1. /\*\*
2. \* Huisou.com Inc.
3. \* Copyright (c) 2011-2012 All Rights Reserved.
4. \*/
6. **package** com.chenzehe.aop;
8. **import** java.lang.reflect.Method;
10. **import** net.sf.cglib.proxy.Enhancer;
11. **import** net.sf.cglib.proxy.MethodInterceptor;
12. **import** net.sf.cglib.proxy.MethodProxy;
14. **public** **class** CGLibProxy **implements** MethodInterceptor {
15. /\*\*
16. \* 要处理的对象
17. \*/
18. **private** Object  delegate;
19. /\*\*
20. \* 切面操作方法
21. \*/
22. **private** Object  operation;
24. /\*\*
25. \* 动态生成方法被处理过后的对象
26. \*/
27. **public** Object bind(Object delegate, Object operation) {
28. **this**.delegate = delegate;
29. **this**.operation = operation;
30. Enhancer enhancer = **new** Enhancer();
31. enhancer.setSuperclass(**this**.delegate.getClass());
32. // 回调方法
33. enhancer.setCallback(**this**);
34. // 创建代理对象
35. **return** enhancer.create();
37. }
39. @Override
40. **public** Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) **throws** Throwable {
41. Object result;
42. // 反射得到操作者的实例
43. Class clazz = **this**.operation.getClass();
44. // 反射得到操作者的before方法
45. Method start = clazz.getDeclaredMethod("before", **new** Class[] { Method.**class** });
46. // 反射执行before方法
47. start.invoke(**this**.operation, **new** Object[] { method });
48. // 执行要处理对象的原本方法
49. result = methodProxy.invokeSuper(proxy, args);
50. // 反射得到操作者的end方法
51. Method end = clazz.getDeclaredMethod("after", **new** Class[] { Method.**class** });
52. // 反射执行end方法
53. end.invoke(**this**.operation, **new** Object[] { method });
54. **return** result;
55. }
57. }

 测试类：

**Java代码  [收藏代码](javascript:void())**

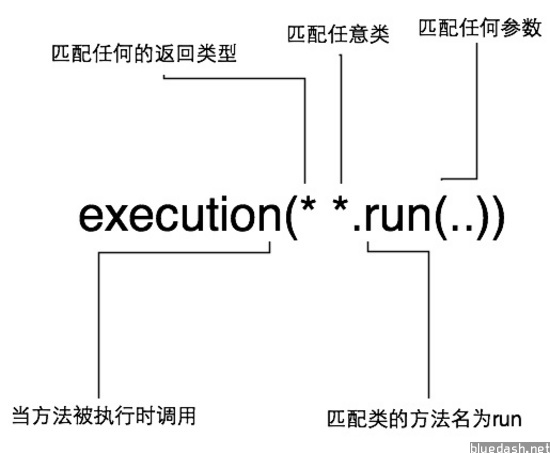
1. /\*\*
2. \* Huisou.com Inc.
3. \* Copyright (c) 2011-2012 All Rights Reserved.
4. \*/
6. **package** com.chenzehe.aop;
8. **import** java.lang.reflect.Method;
10. **import** org.junit.Test;
12. **import** net.sf.cglib.proxy.Enhancer;
13. **import** net.sf.cglib.proxy.MethodInterceptor;
14. **import** net.sf.cglib.proxy.MethodProxy;
16. **public** **class** CGLibProxyTest {
17. @Test
18. **public** **void** testSave() {
19. ServiceNotInterface service = (ServiceNotInterface) **new** CGLibProxy().bind(**new** ServiceNotInterface(), **new** LoggerOperation());
20. service.save();
21. }
22. }
24. 输出：
25. LoggerOperation - Before:save
26. \*\*\*\*\*save\*\*\*\*\*
27. LoggerOperation - After:save

下面留一个问题给大家,如果我们不想让所有方法都被日志记录,我们应该怎么去解藕呢.?我的想法是在代理对象的public Object invoke(Object proxy, Method method, Object[] args)方法里面加上个if(),对传进来的method的名字进行判断,判断的条件存在XML里面.这样我们就可以配置文件时行解藕了.如果有兴趣的朋友可以把操作者,被代理者,都通过配置文件进行配置 ,那么就可以写一个简单的SpringAOP框架了。

JDK动态代理：其代理对象必须是某个接口的实现，它是通过在运行期间创建一个接口的实现类来完成对目标对象的代理。

CGLIB代理：实现原理类似于JDK动态代理，只是它在运行期间生成的代理对象是针对目标类扩展的子类。CGLIB是高效的代码生成包，底层是依靠ASM（开源的java字节码编辑类库）操作字节码实现的，性能比JDK强

CGLib采用了非常底层的字节码技术，其原理是通过字节码技术为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。



**Transactional注解使用**

**原理: 其实就是利用 AOP , spring 生成了一个代理类 这个代理类加入了事务的控制来实现。**

**事务隔离级别**

隔离级别是指若干个并发的事务之间的隔离程度。TransactionDefinition 接口中定义了五个表示隔离级别的常量：

TransactionDefinition.ISOLATION\_DEFAULT：这是默认值，表示使用底层数据库的默认隔离级别。对大部分数据库而言，通常这值就是TransactionDefinition.ISOLATION\_READ\_COMMITTED。

TransactionDefinition.ISOLATION\_READ\_UNCOMMITTED：该隔离级别表示一个事务可以读取另一个事务修改但还没有提交的数据。该级别不能防止脏读，不可重复读和幻读，因此很少使用该隔离级别。比如PostgreSQL实际上并没有此级别。

TransactionDefinition.ISOLATION\_READ\_COMMITTED：该隔离级别表示一个事务只能读取另一个事务已经提交的数据。该级别可以防止脏读，这也是大多数情况下的推荐值。

TransactionDefinition.ISOLATION\_REPEATABLE\_READ：该隔离级别表示一个事务在整个过程中可以多次重复执行某个查询，并且每次返回的记录都相同。该级别可以防止脏读和不可重复读。

TransactionDefinition.ISOLATION\_SERIALIZABLE：所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。但是这将严重影响程序的性能。通常情况下也不会用到该级别。

**事务传播行为**

所谓事务的传播行为是指，如果在开始当前事务之前，一个事务上下文已经存在，此时有若干选项可以指定一个事务性方法的执行行为。在TransactionDefinition定义中包括了如下几个表示传播行为的常量：

TransactionDefinition.PROPAGATION\_REQUIRED：如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务。这是默认值。

TransactionDefinition.PROPAGATION\_REQUIRES\_NEW：创建一个新的事务，如果当前存在事务，则把当前事务挂起。

TransactionDefinition.PROPAGATION\_SUPPORTS：如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。

TransactionDefinition.PROPAGATION\_NOT\_SUPPORTED：以非事务方式运行，如果当前存在事务，则把当前事务挂起。

TransactionDefinition.PROPAGATION\_NEVER：以非事务方式运行，如果当前存在事务，则抛出异常。

TransactionDefinition.PROPAGATION\_MANDATORY：如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。

TransactionDefinition.PROPAGATION\_NESTED：如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于TransactionDefinition.PROPAGATION\_REQUIRED。

**事务超时**

所谓事务超时，就是指一个事务所允许执行的最长时间，如果超过该时间限制但事务还没有完成，则自动回滚事务。在 TransactionDefinition 中以 int 的值来表示超时时间，其单位是秒。

默认设置为底层事务系统的超时值，如果底层数据库事务系统没有设置超时值，那么就是none，没有超时限制。

**事务只读属性**

只读事务用于客户代码只读但不修改数据的情形，只读事务用于特定情景下的优化，比如使用Hibernate的时候。

默认为读写事务。

**spring事务回滚规则**

指示spring事务管理器回滚一个事务的推荐方法是在当前事务的上下文内抛出异常。spring事务管理器会捕捉任何未处理的异常，然后依据规则决定是否回滚抛出异常的事务。

默认配置下，spring只有在抛出的异常为运行时unchecked异常时才回滚该事务，也就是抛出的异常为RuntimeException的子类(Errors也会导致事务回滚)，而抛出checked异常则不会导致事务回滚。

可以明确的配置在抛出那些异常时回滚事务，包括checked异常。也可以明确定义那些异常抛出时不回滚事务。

还可以编程性的通过setRollbackOnly()方法来指示一个事务必须回滚，在调用完setRollbackOnly()后你所能执行的唯一操作就是回滚。

**@Transactional注解**

@Transactional属性



**用法**

@Transactional 可以作用于接口、接口方法、类以及类方法上。当作用于类上时，该类的所有 public 方法将都具有该类型的事务属性，同时，我们也可以在方法级别使用该标注来覆盖类级别的定义。

虽然 @Transactional 注解可以作用于接口、接口方法、类以及类方法上，但是 Spring 建议不要在接口或者接口方法上使用该注解，因为这只有在使用基于接口的代理时它才会生效。另外， @Transactional 注解应该只被应用到 public方法上，这是由Spring AOP的本质决定的。如果你在protected、private 或者默认可见性的方法上使用 @Transactional 注解，这将被忽略，也不会抛出任何异常。

默认情况下，只有来自外部的方法调用才会被AOP代理捕获，也就是，类内部方法调用本类内部的其他方法并不会引起事务行为，即使被调用方法使用@Transactional注解进行修饰。

@Transactional(readOnly = true)

public class DefaultFooService implements FooService {

  public Foo getFoo(String fooName) {

    // do something

  }

  // these settings have precedence for this method

  //方法上注解属性会覆盖类注解上的相同属性

@Transactional(readOnly = false, propagation = Propagation.REQUIRES\_NEW)

  public void updateFoo(Foo foo) {

    // do something

  }

}

# 事务的嵌套概念

所谓事务的嵌套就是两个事务方法之间相互调用。spring事务开启 ，或者是基于接口的或者是基于类的代理被创建（**注意一定要是代理，不能手动new 一个对象，并且此类（有无接口都行）一定要被代理——spring中的bean只要纳入了IOC管理都是被代理的**）。所以在同一个类中一个方法调用另一个方法有事务的方法，事务是不会起作用的。

Spring默认情况下会对运行期例外(RunTimeException)，即uncheck异常，进行事务回滚。

如果遇到checked异常就不回滚。

如何改变默认规则：

1、 让checked例外也回滚：在整个方法前加上 @Transactional(rollbackFor=Exception.class)

2、 让unchecked例外不回滚： @Transactional(notRollbackFor=RunTimeException.class)

3、 不需要事务管理的(只查询的)方法：@Transactional(propagation=Propagation.NOT\_SUPPORTED)

上面三种方式也可在xml配置

# spring事务传播属性

 在 spring的 TransactionDefinition接口中一共定义了六种事务传播属性：

**PROPAGATION\_REQUIRED** -- 支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。

**PROPAGATION\_SUPPORTS** -- 支持当前事务，如果当前没有事务，就以非事务方式执行。

**PROPAGATION\_MANDATORY** -- 支持当前事务，如果当前没有事务，就抛出异常。

**PROPAGATION\_REQUIRES\_NEW** -- 新建事务，如果当前存在事务，把当前事务挂起。

**PROPAGATION\_NOT\_SUPPORTED** -- 以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

**PROPAGATION\_NEVER** -- 以非事务方式执行，如果当前存在事务，则抛出异常。

**PROPAGATION\_NESTED** -- 如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则进行与PROPAGATION\_REQUIRED类似的操作。 

前六个策略类似于EJB CMT，第七个（PROPAGATION\_NESTED）是Spring所提供的一个特殊变量。

它要求事务管理器或者使用JDBC 3.0 Savepoint API提供嵌套事务行为（如Spring的DataSourceTransactionManager）

举例浅析Spring嵌套事务

**ServiceA#methodA(我们称之为外部事务)，ServiceB#methodB(我们称之为外部事务)**

1. ServiceA {
3. **void** methodA() {
4. ServiceB.methodB();
5. }
7. }
9. ServiceB {
11. **void** methodB() {
12. }
14. }

## PROPAGATION\_REQUIRED

假如当前正要执行的事务不在另外一个事务里，那么就起一个新的事务。比如说，ServiceB.methodB的事务级别定义为PROPAGATION\_REQUIRED, 那么由于执行ServiceA.methodA的时候

  1、如果ServiceA.methodA已经起了事务，这时调用ServiceB.methodB，ServiceB.methodB看到自己已经运行在ServiceA.methodA的事务内部，就不再起新的事务。这时只有外部事务并且他们是共用的，所以这时ServiceA.methodA或者ServiceB.methodB无论哪个发生异常methodA和methodB作为一个整体都将一起回滚。

  2、如果ServiceA.methodA没有事务，ServiceB.methodB就会为自己分配一个事务。这样，在ServiceA.methodA中是没有事务控制的。只是在ServiceB.methodB内的任何地方出现异常，ServiceB.methodB将会被回滚，不会引起ServiceA.methodA的回滚

## PROPAGATION\_SUPPORTS

如果当前在事务中，即以事务的形式运行，如果当前不再一个事务中，那么就以非事务的形式运行

## PROPAGATION\_MANDATORY

必须在一个事务中运行。也就是说，他只能被一个父事务调用。否则，他就要抛出异常

## PROPAGATION\_REQUIRES\_NEW

启动一个新的, 不依赖于环境的 "内部" 事务. 这个事务将被完全 commited 或 rolled back 而不依赖于外部事务, 它拥有自己的隔离范围, 自己的锁, 等等. 当内部事务开始执行时, 外部事务将被挂起, 内务事务结束时, 外部事务将继续执行.

 比如我们设计ServiceA.methodA的事务级别为PROPAGATION\_REQUIRED，ServiceB.methodB的事务级别为PROPAGATION\_REQUIRES\_NEW，那么当执行到ServiceB.methodB的时候，ServiceA.methodA所在的事务就会挂起，ServiceB.methodB会起一个新的事务，等待ServiceB.methodB的事务完成以后，他才继续执行。他与PROPAGATION\_REQUIRED 的事务区别在于事务的回滚程度了。因为ServiceB.methodB是新起一个事务，那么就是存在两个不同的事务。

1、如果ServiceB.methodB已经提交，那么ServiceA.methodA失败回滚，ServiceB.methodB是不会回滚的。

2、如果ServiceB.methodB失败回滚，如果他抛出的异常被ServiceA.methodA的try..catch捕获并处理，ServiceA.methodA事务仍然可能提交；如果他抛出的异常未被ServiceA.methodA捕获处理，ServiceA.methodA事务将回滚。

**使用场景：**

不管业务逻辑的service是否有异常，Log Service都应该能够记录成功，所以Log Service的传播属性可以配为此属性。最下面将会贴出配置代码。

## PROPAGATION\_NOT\_SUPPORTED

当前不支持事务。比如ServiceA.methodA的事务级别是PROPAGATION\_REQUIRED ，而ServiceB.methodB的事务级别是PROPAGATION\_NOT\_SUPPORTED ，那么当执行到ServiceB.methodB时，ServiceA.methodA的事务挂起，而他以非事务的状态运行完，再继续ServiceA.methodA的事务。

## PROPAGATION\_NEVER

不能在事务中运行。假设ServiceA.methodA的事务级别是PROPAGATION\_REQUIRED， 而ServiceB.methodB的事务级别是PROPAGATION\_NEVER ，那么ServiceB.methodB就要抛出异常了。

## PROPAGATION\_NESTED

开始一个 "嵌套的" 事务,  它是已经存在事务的一个真正的子事务. 潜套事务开始执行时,  它将取得一个 savepoint. 如果这个嵌套事务失败, 我们将回滚到此 savepoint. 潜套事务是外部事务的一部分, 只有外部事务结束后它才会被提交.

比如我们设计ServiceA.methodA的事务级别为PROPAGATION\_REQUIRED，ServiceB.methodB的事务级别为PROPAGATION\_NESTED，那么当执行到ServiceB.methodB的时候，ServiceA.methodA所在的事务就会挂起，ServiceB.methodB会起一个**新的子事务并设置savepoint**，等待ServiceB.methodB的事务完成以后，他才继续执行。。因为ServiceB.methodB是外部事务的子事务，那么

1、如果ServiceB.methodB已经提交，那么ServiceA.methodA失败回滚，ServiceB.methodB也将回滚。

2、如果ServiceB.methodB失败回滚，如果他抛出的异常被ServiceA.methodA的try..catch捕获并处理，ServiceA.methodA事务仍然可能提交；如果他抛出的异常未被ServiceA.methodA捕获处理，ServiceA.methodA事务将回滚。

理解Nested的关键是savepoint。他与PROPAGATION\_REQUIRES\_NEW的区别是：

PROPAGATION\_REQUIRES\_NEW 完全是一个新的事务,它与外部事务相互独立； 而 PROPAGATION\_NESTED 则是外部事务的子事务, 如果外部事务 commit, 嵌套事务也会被 commit, 这个规则同样适用于 roll back.

### 在 spring 中使用 PROPAGATION\_NESTED的前提：

#### 1. 我们要设置 transactionManager 的 nestedTransactionAllowed 属性为 true, 注意, 此属性默认为 false!!!

#### 2. java.sql.Savepoint 必须存在, 即 jdk 版本要 1.4+

#### 3. Connection.getMetaData().supportsSavepoints() 必须为 true, 即 jdbc drive 必须支持 JDBC 3.0

#### 确保以上条件都满足后, 你就可以尝试使用 PROPAGATION\_NESTED 了.

# Log Service配置事务传播

不管业务逻辑的service是否有异常，Log Service都应该能够记录成功，通常有异常的调用更是用户关心的。Log Service如果沿用业务逻辑Service的事务的话在抛出异常时将没有办法记录日志(事实上是回滚了)。所以希望Log Service能够有独立的事务。日志和普通的服务应该具有不同的策略。Spring 配置文件transaction.xml:

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:aop="http://www.springframework.org/schema/aop"
4. xmlns:tx="http://www.springframework.org/schema/tx"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd"**>**
6. <!-- configure transaction -->
8. **<tx:advice** id="defaultTxAdvice" transaction-manager="transactionManager"**>**
9. **<tx:attributes>**
10. **<tx:method** name="get\*" read-only="true" **/>**
11. **<tx:method** name="query\*" read-only="true" **/>**
12. **<tx:method** name="find\*" read-only="true" **/>**
13. **<tx:method** name="\*" propagation="REQUIRED" rollback-for="java.lang.Exception" **/>**
14. **</tx:attributes>**
15. **</tx:advice>**
17. **<tx:advice** id="logTxAdvice" transaction-manager="transactionManager"**>**
18. **<tx:attributes>**
19. **<tx:method** name="get\*" read-only="true" **/>**
20. **<tx:method** name="query\*" read-only="true" **/>**
21. **<tx:method** name="find\*" read-only="true" **/>**
22. **<tx:method** name="\*" propagation="REQUIRES\_NEW"
23. rollback-for="java.lang.Exception" **/>**
24. **</tx:attributes>**
25. **</tx:advice>**
27. **<aop:config>**
28. **<aop:pointcut** id="defaultOperation"
29. expression="@within(com.homent.util.DefaultTransaction)" **/>**
30. **<aop:pointcut** id="logServiceOperation"
31. expression="execution(\* com.homent.service.LogService.\*(..))" **/>**
33. **<aop:advisor** advice-ref="defaultTxAdvice" pointcut-ref="defaultOperation" **/>**
34. **<aop:advisor** advice-ref="logTxAdvice" pointcut-ref="logServiceOperation" **/>**
35. **</aop:config>**
36. **</beans>**

 如上面的Spring配置文件所示，日志服务的事务策略配置为propagation="REQUIRES\_NEW"，告诉Spring不管上下文是否有事务，Log Service被调用时都要求一个完全新的只属于Log Service自己的事务。通过该事务策略，Log Service可以独立的记录日志信息，不再受到业务逻辑事务的干扰。