**Spring4第2天课程**

**【学习目标】**

通过本内容的学习，学生：

## 能够编写spring的IOC的注解代码

1. 独立配置spring的注解组件的扫描
2. 独立编写spring的测试相关的类
3. 独立使用注解修饰类
4. 独立使用注解完成属性注入
5. 独立编写测试代码

## 能够说出spring的IOC的相关注解的含义

1. 说出@Component的含义
2. 说出@Controller的含义
3. 说出@Service的含义
4. 说出@Repository的含义
5. 说出@Autowired的含义
6. 说出@Qulifer的含义
7. 说出@Resource的含义

## 3，能够实现spring框架整合JUnit

1. 说出实现spring框架整合JUnit 的整合流程
2. 说出@RunWith注解的作用
3. 说出@ContextConfiguration注解的作用
4. 独立实现spring整合JUnit单元测试

## 4，能够理解AOP 相关概念

1. 说出AOP的概念
2. 说出AOP的应用场景
3. 说出AOP的底层实现原理

## 5，能够说出AOP 相关术语的含义

1. 说出AOP中Joinpoint的含义
2. 说出AOP中Pointcut的含义
3. 说出AOP中Advice的含义
4. 说出AOP中Target的含义
5. 说出AOP中Weaving的含义
6. 说出AOP中Proxy的含义
7. 说出AOP中Aspect的含义

## 6，能够应用AspectJ表达式语言

1. 说出execution函数的使用方法
2. 说出通配符\*的使用方法
3. 说出通配符..的使用方法
4. 说出通配符+的使用方法
5. 独立应用AspectJ表达式编写切入点表达式

## 7，能够编写Spring的AOP中不同通知的代码

1. 独立编写AOP 的前置通知的代码
2. 独立编写AOP 的后置通知的代码
3. 独立编写AOP 的环绕通知的代码
4. 独立编写AOP 的异常抛出通知的代码
5. 独立编写AOP 的最终通知的代码

# 使用Spring的IOC完成保存客户的操作:

## 案例需求

### 需求描述

对于CRM的系统而言，现在有很多的DAO类，比如客户的DAO，联系人DAO等等。客户提出一个需求要开发人员实现一个功能对所有的DAO的类中以save开头的方法实现权限的校验，需要时管理员的身份才可以进行保存操作。

# IoC容器装配Bean\_基于注解配置方式

## Bean的定义（注册） -- 扫描机制

新建web项目：spring4\_d02\_c02

第一步：

导入jar包（4个核心包+2个日志包），

导入log4j.properties,

导入applicationContext.xml

Spring XML开发和注解开发 导入jar包是相同的

第二步： 编写Service和DAO 的注册

xml做法 ： <bean id=”customerService” class=”…” />，用<bean>的方式创建对象

注解做法 ： spring2.5引入 @Component 注解 如果放置到类的上面，相当于在spring容器中定义<bean id=”” class=””>

创建包：com.igeek.ioc

创建类：CustomerService.java类

/\*\*

\* **@Component注解放置到类上**

\* 相当于spring容器中定义：<bean id="customerService" class="com.igeek.ioc.CustomerService">

\* 其中id属性默认bean的名字是类名的小写

\* ——————————————————————————————————————————————————————

\* **@Component**(value="customer")//自定义bean的名字

\* 相当于spring容器中定义：<bean id="customer" class="com.igeek.ioc.CustomerService">

\* ——————————————————————————————————————————————————————

\*/

@Component(value="customer")

**public** **class** CustomerService {

//保存业务方法

**public** **void** save(){

System.*out*.println("CustomerService业务层被调用了。。。");

}

}

第三步： 配置注解开启和注解Bean的扫描。配置的示例如下：配置applicationContext.xml

参考：

spring-framework-4.2.4.RELEASE/docs/spring-framework-reference/html/xsd-configuration.html，搜索context关键字即可

<?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***"*

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"*>

<!-- 配置注解扫描

context:component-scan:专门扫描含有@Component注解的类，自动将其作为bean

base-package：要扫描包的路径,包含子包,com.igeek.ioc表示子包下的所有类定义注解都有效

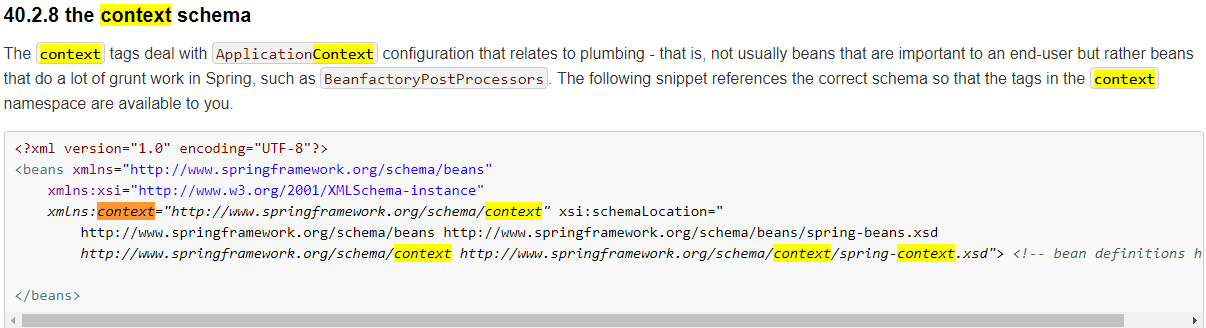
注解扫描配置的时候，会自动开启注解功能

-->

<context:component-scan base-package=*"com.igeek.ioc"*/>

</beans>

引入context 名称空间 ：

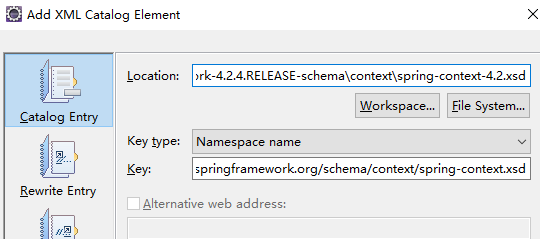


【注意】Spring的所有名称空间都需要基于Beans的名称空间。

引入后：

|  |
| --- |
| <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"*  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"*> |

配置本地提示关联，配置eclipse的XML Catalog Element：



第四步：测试：

**public** **class** SpringTest {

@Test

**public** **void** test(){

//spring容器

ApplicationContext applicationContext = **new** ClassPathXmlApplicationContext("applicationContext.xml");

//获取bean

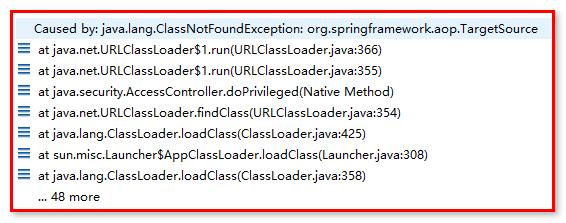
CustomerService customerService=(CustomerService) applicationContext.getBean("customer");

customerService.save();

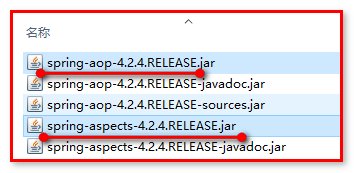
}

}

这里：如果抛出异常：



说明spring4缺少aop的包，需要导入



【扩展优化】：

衍生注解的问题

实际开发中，使用的是@Component三个衍生注解（“子注解”）

子注解的作用：有分层的意义（分层注解）。

Spring3.0为我们引入了组件自动扫描机制，它可以在类路径底下寻找标注了@Component、@Service、@Controller、@Repository注解的类，并把这些类纳入进spring容器中管理。

除了@Component外，Spring提供了3个功能基本和@Component等效的注解

功能介绍

@Service用于标注业务层组件、（如Service层）

@Controller用于标注控制层组件（如struts中的action层）

@Repository用于标注数据访问组件，（如DAO层组件）。

而@Component泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注。

第一步：

修改CutomerService.java

//@Component(value="customer")//注释掉

@Service(value="customer")

**public** **class** CustomerService {

//保存业务方法

**public** **void** save(){

System.*out*.println("CustomerService业务层被调用了。。。");

}

}

创建CustomerDao.java

//持久层

@Repository("customerDao")

**public** **class** CustomerDao {

**public** **void** save(){

System.*out*.println("CustomerDao层被调用了");

}

}

【抛出问题】：如果将Dao注入到Service呢？即使用Service类调用Dao类？

回顾：如果使用xml的配置，那么可以使用setter方法进行注入

<bean id=”” class=””>

<property name=”” ref=””></property>

</bean>

让我们接着往下学习。

## Bean属性的依赖注入

### 简单数据类型依赖注入（了解）

Spring3.0后，提供 @Value注解，可以完成简单数据的注入

//@Component(value="customer")

@Service(value="customer")

**public** **class** CustomerService {

//简单类型的成员变量

**@Value("Rose")**//参数的值简单类型

**private** String name="Jack";

//保存业务方法

**public** **void** save(){

System.*out*.println("CustomerService业务层被调用了。。。");

System.*out*.println("name:"+name);

}

}

### 复杂类型数据依赖注入

下面完成，将Dao类的对象注入到Service类进行使用。

注解实现属性依赖注入，将注解加在setXxx方法上 或者 属性定义上 ！（任选其一，省代码了）

第一种： 使用@Value 结合SpEL ---- spring3.0 后用

//@Component(value="customer")

@Service(value="customer")

**public** **class** CustomerService {

//简单类型的成员变量

@Value("Rose")//参数的值简单类型

**private** String name="Jack";

//在属性声明上面注入，底层自动还是生成setCustomerDao()

//第一种： 使用@Value 结合SpEL ---- spring3.0 后用

//其中customerDao表示<bean>节点id的属性值

/\*\*第一种： 使用@Value 结合SpEL ---- spring3.0 后用\*/

//@Value(value="#{customerDao}")

**private** CustomerDao customerDao;

@Value(value="#{customerDao}")

**public** **void** setCustomerDao(CustomerDao customerDao) {

**this**.customerDao = customerDao;

}

//保存业务方法

**public** **void** save(){

System.*out*.println("CustomerService业务层被调用了。。。");

System.*out*.println("name:"+name);

customerDao.save();

}

}

第二种：使用@Autowired 结合 @Qualifier

单独使用@Autowired ，表示按照类型注入，会到spring容器中查找CustomerDao的类型，对应<bean class=””>，class的属性值，如果找到，可以匹配。

//第二种：使用spring的@Autowired

@Autowired//默认按照类型注入

**private** CustomerDao customerDao;

使用@Autowired + @ Qualifier 表示按照名称注入，会到spring容器中查找customerDao的名称，对应<bean id=””>，id的属性值，如果找到，可以匹配。

//第二种：使用spring的@Autowired 结合 @Qualifier

@Autowired//默认按照类型注入的

@Qualifier("customerDao")//必须配合@Autowired注解使用，根据名字注入

**private** CustomerDao customerDao;

第三种： JSR-250标准（基于jdk） 提供注解@Resource

单独使用@Resource注解，表示先按照名称注入，会到spring容器中查找customerDao的名称，对应<bean id=””>，id的属性值，如果找到，可以匹配。

如果没有找到，则会按照类型注入，会到spring容器中查找CustomerDao的类型，对应<bean class=””>，class的属性值，如果找到，可以匹配，如果没有找到会抛出异常。

//第三种： JSR-250标准（jdk） 提供@Resource

@Resource//默认先按照名称进行匹配，再按照类型进行匹配

**private** CustomerDao customerDao;

如果@Resource注解上添加name名称

使用@Resource注解，则按照名称注入，会到spring容器中查找customerDao的名称，对应<bean id=””>，id的属性值，如果找到，可以匹配。

如果没有找到，抛出异常。

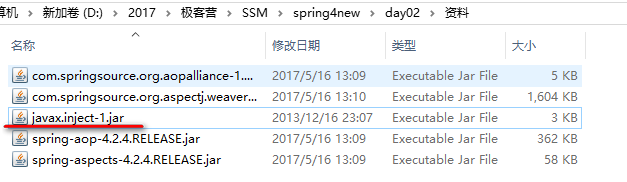
//第三种： JSR-250标准（jdk） 提供@Resource

@Resource(name="customerDao")//只能按照customerDao名称进行匹配

**private** CustomerDao customerDao;

第四种： JSR-330标准（jdk） 提供 @Inject （麻烦点）

需要先导入 javax.inject 的 jar ，在课前资料中查找。



使用@Inject注解，则按照类型注入，

//第四种： JSR-330标准（jdk） 提供 @Inject ，配合@Named注解

@Inject//默认按照类型注入

**private** CustomerDao customerDao;

使用@inject和@Named注解，则按照名称注入

//第四种： JSR-330标准（jdk） 提供 @Inject ，配合@Named注解

@Inject//默认按照类型注入

@Named("customerDao")//按照名字注入，必须配合@Inject使用

**private** CustomerDao customerDao;

## Bean的初始化和销毁

使用注解定义Bean的初始化和销毁

Spring初始化bean或销毁bean时，有时需要作一些处理工作，因此spring可以在创建和拆卸bean的时候调用bean的两个生命周期方法。

回顾配置文件的写法：<bean id=“foo” class=“...Foo” **init-method=“setup”destory-method=“teardown”**/>

注解的写法：

（1）当bean被载入到容器的时候调用setup ，

注解方式如下:

@PostConstruct

初始化

（2）当bean从容器中删除的时候调用teardown(scope= singleton有效)

注解方式如下:

@PreDestroy

销毁

使用 @PostConstruct 注解， 标明初始化方法 ---相当于 init-method 指定初始化方法

使用 @PreDestroy 注解， 标明销毁方法 ----相当于 destroy-method 指定对象销毁方法

第一步：创建类：LifeCycleBean.java，定义构造方法、初始化的方法、销毁的方法。

//测试生命周期过程中的初始化和销毁bean

@Component("lifeCycleBean")

**public** **class** LifeCycleBean {

**public** LifeCycleBean() {

System.*out*.println("LifeCycleBean构造器调用了");

}

//初始化后自动调用方法：方法名随意，但也不能太随便，一会要配置

@PostConstruct//初始化的方法

**public** **void** init(){

System.*out*.println("LifeCycleBean-init初始化时调用");

}

//bean销毁时调用的方法

@PreDestroy

**public** **void** destroy(){

System.*out*.println("LifeCycleBean-destroy销毁时调用");

}

}

第二步：配置文件，配置spring容器applicationContext.xml

<?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"*

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"*>

<!-- 配置注解扫描

context:component-scan:专门扫描含有@Component注解的类，自动将其作为bean

base-package：要扫描包的路径,包含子包,com.igeek.ioc表示子包下的所有类定义注解都有效

注解扫描配置的时候，会自动开启注解功能

-->

<context:component-scan base-package=*"com.igeek.ioc"*/>

</beans>

第三步：使用SpringTest.java完成测试

@Test

**public** **void** testLifeCycle() **throws** Exception{

//spring容器

ApplicationContext applicationContext = **new** ClassPathXmlApplicationContext("applicationContext.xml");

//单例；此时初始化的方法已经被调用

LifeCycleBean lifeCycleBean = (LifeCycleBean)applicationContext.getBean("lifeCycleBean");

//方案一：

//**((ClassPathXmlApplicationContext)applicationContext).close();**

//方案二：

//反射的机制调用close方法。

//接口只是引用了一个对象。对象本身有这个方法。

//目标：通过接口引用，调用对象本来的拥有的方法

//1。获取对象具体类的某个方法:参数1方法名，参数2：方法里面的参数类型

Method method = applicationContext.getClass().getMethod("close");

//参数1：拥有该方法的对象的名字,参数2：方法里面的参数的值

method.invoke(applicationContext);

}

注意：如果要执行对象的销毁方法

条件一： 单例Bean （在容器close时，单例Bean才会执行销毁方法 ）

条件二： 必须调用容器 close 方法

## Bean的作用域

通过@Scope注解，指定Bean的作用域（默认是 singleton 单例）

回顾：XML的方式<bean id=”” class=”” scope=”prototype”>

//测试生命周期过程中的初始化和销毁bean

@Component("lifeCycleBean")

//@Scope(value=ConfigurableBeanFactory.SCOPE\_PROTOTYPE)

@Scope("prototype")//默认是单例(singleton),更改为多例(prototype)

**public** **class** LifeCycleBean {

}

测试：

@Test

**public** **void** testLifeCycleScope() **throws** Exception{

//spring容器

ApplicationContext applicationContext = **new** ClassPathXmlApplicationContext("applicationContext.xml");

//单例；此时初始化的方法已经被调用

LifeCycleBean lifeCycleBean1 = (LifeCycleBean)applicationContext.getBean("lifeCycleBean");

LifeCycleBean lifeCycleBean2 = (LifeCycleBean)applicationContext.getBean("lifeCycleBean");

System.***out***.println(lifeCycleBean1);

System.***out***.println(lifeCycleBean2);

}

## XML和注解混合配置 （重点）

一个项目中XML和注解都有（特殊年代产物）

* Spring2.0 就有@Autowired注解
* Spring2.5 之后才有@Component注解

使用

XML 完成Bean定义

注解 完成Bean属性注入

创建包：com.igeek.ioc.mixed

第一步：使用XML的方式完成Bean的定义

创建applicationContext-mixed.xml文件，定义：

<?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"*

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"*>

<!-- xml方式定义bean -->

<bean id=*"productDao"* class=*"com.igeek.ioc.mixed.ProductDao"*/>

<bean id=*"productService"* class=*"com.igeek.ioc.mixed.ProductService"*/>

<!-- 需要单独开启注解功能 -->

<context:component-scan base-package=*"com.igeek.ioc"*></context:component-scan>

</beans>

第二步：注解完成注入

（1）创建ProductDao类

//产品的数据层

**public** **class** ProductDao {

**public** **void** save(){

System.*out*.println("查询保存到数据口--数据层调用了");

}

}

（2）创建ProductService类

//产品的业务层

**public** **class** ProductService {

//注入dao

//强调：注入必须是bean注入bean

**@Autowired**

**private** ProductDao productDao;

//产品的保存

**public** **void** save(){

System.*out*.println("产品保存了，--业务层");

//调用dao层

productDao.save();

}

}

测试类：

**public** **class** SpringTest {

@Test

**public** **void** test(){

//spring容器

ApplicationContext applicationContext = **new** ClassPathXmlApplicationContext("applicationContext-mixed.xml");

//获取bean

ProductService productService=(ProductService) applicationContext.getBean("productService");

productService.save();

}

}

备注：这里配置 <context:component-scan base-package=*"com.igeek.ioc"*></context:component-scan>

才能使用 @PostConstruct @PreDestroy @Autowired @Resource等注解

<!-- 需要在spring容器中单独开启注解功能，扫描com.igeek.ioc包及其子包中所有类都有效，类中都可以使用注解 -->

<context:component-scan base-package=*"com.igeek.ioc"*></context:component-scan>

## Spring的junit测试集成

Spring提供spring-test-4.2.4.RELEASE.jar 可以整合junit。

优势：可以简化测试代码（不需要手动创建上下文，即手动创建spring容器）

使用spring和junit集成

第一步：新建项目导入junit 开发包

第二步：导入spring-test-4.2.4.RELEASE.jar



第三步： 创建包com.igeek.test,创建类SpringTest

通过@RunWith注解，使用junit整合spring

通过@ContextConfiguration注解，指定spring容器的位置

//目标：测试一下spring的bean的某些功能

@RunWith(SpringJUnit4ClassRunner.**class**)//junit整合spring的测试//立马开启了spring的注解

@ContextConfiguration(locations="classpath:applicationContext.xml")//加载核心配置文件，自动构建spring容器

**public** **class** SpringTest {

//使用注解注入要测试的bean

@Autowired

**private** HelloService helloService;

@Test

**public** **void** testSayHello(){

//获取spring容器

// ApplicationContext applicationContext = new ClassPathXmlApplicationContext("applicationContext.xml");

//从spring容器中获取bean对象

// HelloService helloService=(HelloService)applicationContext.getBean("helloService");

//测试业务功能

helloService.sayHello();

}

}

上述代码表示：在测试类运行前的初始化的时候，会自动创建ApplicationContext对象

第四步： 通过@Autowired注解，注入需要测试的对象

在这里注意2点：

1. 将测试对象注入到测试用例中
2. 测试用例不需要配置<context:annotion-config/>，或者是<context:component-scan base-package="com.igeek"/>，因为使用测试类运行的时候，会自动启动注解的支持。

//使用注解注入要测试的bean

@Autowired

**private** HelloService helloService;

第五步：调用测试方法完成测试

@Test

**public** **void** testSayHello(){

helloService.sayHello();

}

第六步：在applicationContext.xml中添加：

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:p=*"http://www.springframework.org/schema/p"*

xmlns:context=*"http://www.springframework.org/schema/context"*

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"*>

<!-- 开启组件扫描 -->

<context:component-scan base-package=*"com.igeek.ioc"*/>

<!-- 创建Service -->

<bean id=*"helloService"* class=*"com.igeek.test.HelloService"*></bean>

</beans>

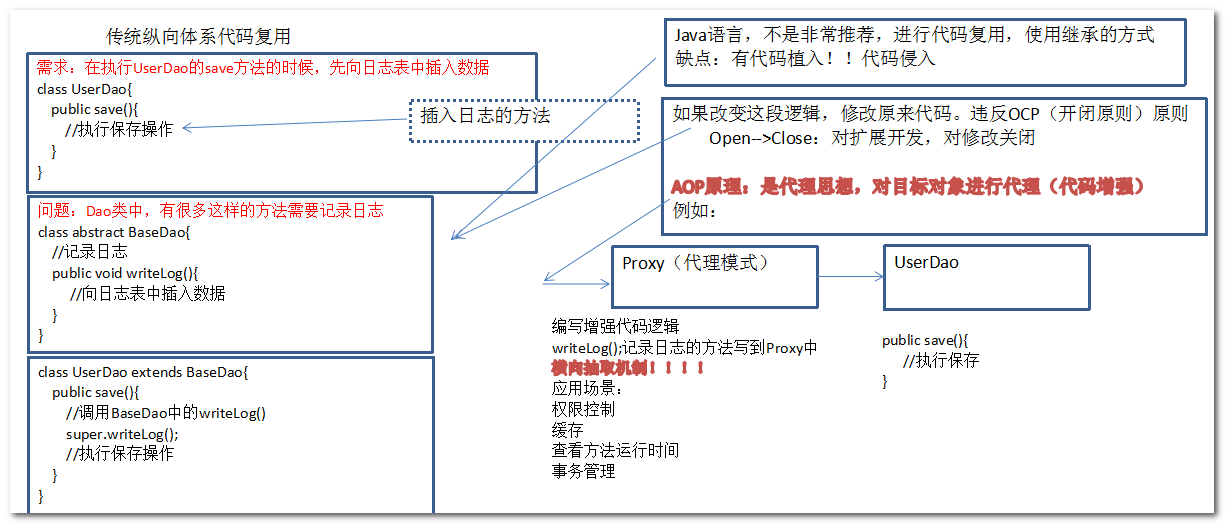
# AOP面向切面编程的相关概念

## 什么是AOP ？

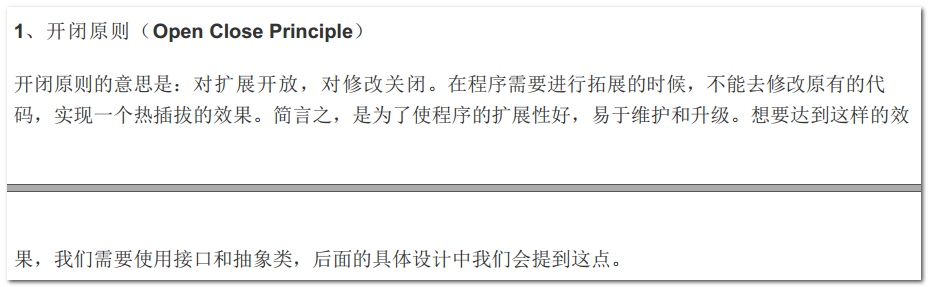
AOP (Aspect Oriented Programing) 称为：面向切面编程，它是一种编程思想。

AOP采取**横向抽取机制**，取代了**传统纵向继承**体系重复性代码的编写方式（例如性能监视、事务管理、安全检查、缓存、日志记录等）。

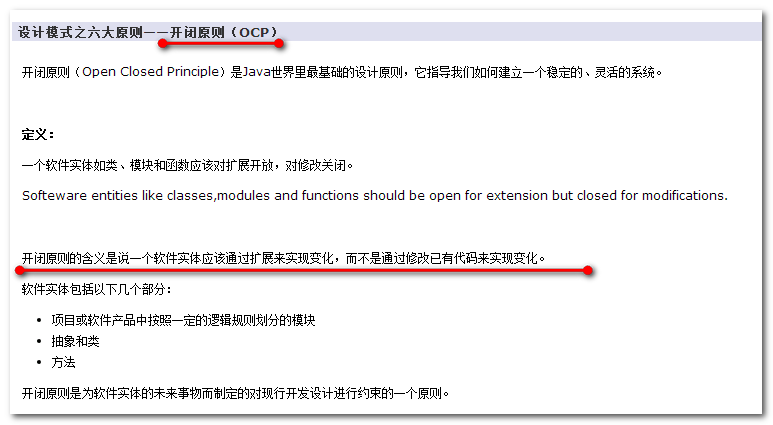
【扩展了解】AOP 是 OOP（面向对象编程（Object Oriented Programming，OOP，面向对象程序设计）是一种计算机编程架构），思想延续 ！



什么是OCP：即开闭原则。



参考网站：http://www.cnblogs.com/muzongyan/archive/2010/08/05/1793454.html



AOP 思想： 基于代理思想，对原来目标对象，创建代理对象，在不修改原对象代码情况下，通过代理对象，调用增强功能的代码，从而对原有业务方法进行增强 ！

切面：需要代理一些方法和增强代码 。

## AOP的应用场景

场景一： 记录日志

场景二： 监控方法运行时间 （监控性能）

场景三： 权限控制

场景四： 缓存优化 （第一次调用查询数据库，将查询结果放入内存对象， 第二次调用， 直接从内存对象返回，不需要查询数据库 ）

场景五： 事务管理 （调用方法前开启事务， 调用方法后提交或者回滚、关闭事务 ）

## Spring AOP编程两种方式

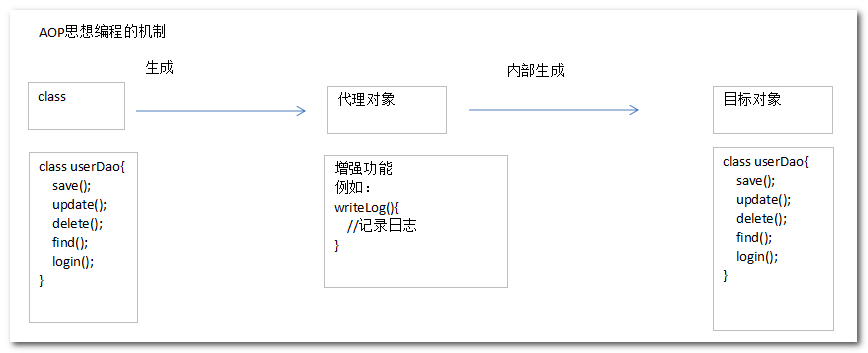
* Spring AOP使用纯Java实现，不需要专门的编译过程和类加载器，在运行期通过代理方式向目标类植入增强代码。
* AsPecJ是一个基于Java语言的AOP框架，Spring2.0开始，Spring AOP引入对Aspect的支持。

简单的说，Spring内部支持两套AOP编程的方案：

* Spring 1.2 开始支持AOP编程 （传统SpringAOP 编程），编程非常复杂 ---- 更好学习Spring 内置传统AOP代码
* Spring 2.0 之后支持第三方 AOP框架（AspectJ ），实现另一种 AOP编程 -- 推荐

## AOP编程相关术语

**AOP思想编程的机制**



**AOP的相关术语**

**Aspect(切面):** 是通知和切入点的结合,通知和切入点共同定义了关于切面的全部内容---它的功能、在何时和何地完成其功能

**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.

**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.

通知定义了切面的”什么”和”何时”，切入点就定义了”何地”.

**Advice(通知):**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)

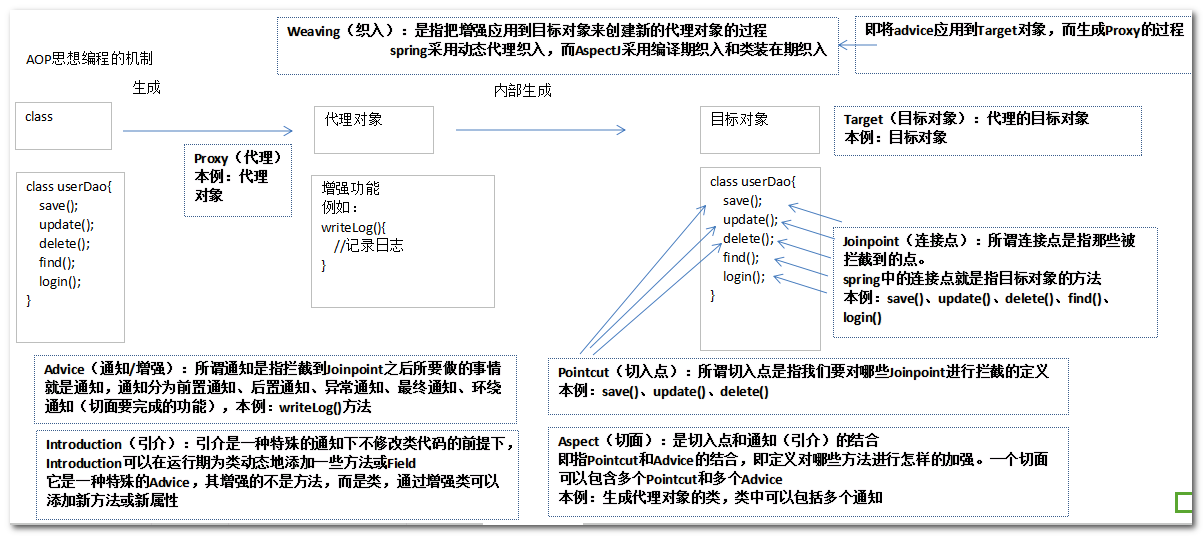
**Target(目标对象):**代理的目标对象

**Weaving(织入):**是指把切面应用到目标对象来创建新的代理对象的过程.切面在指定的连接点织入到目标对象

**Introduction(引入)（不要求掌握）:**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.

**通过案例解释AOP相关概念**

需求：UserDao中有5个方法，分别是save()、update()、delete()、find()、login()；在访问UserDao的save()、update()、delete()方法之前，进行记录日志的操作。



Aspect切面（类）：增强代码 Advice（writeLog方法）和 切入点 Pointcut（save，update，delete） 的结合。换句话说：对哪些方法进行怎样的代码增强。

# AOP编程底层实现机制（了解）

AOP 就是要对目标进行代理对象的创建， Spring AOP是基于动态代理的，基于两种动态代理机制： JDK动态代理和CGLIB动态代理

面试题：动态代理和静态代理区别？

动态代理：在虚拟机内部，运行的时候，动态生成代理类(运行时生成，runtime生成) ，并不是真正存在的类， 一般格式：Proxy$$ （Proxy$$Customer）

静态代理：实际存在代理类 （例如：struts2 Action的代理类 ActionProxy，struts2的拦截器）

附录：什么是ActionProxy

参考：<http://www.tuicool.com/articles/RNRRriz>



## JDK动态代理

JDK动态代理，针对目标对象的接口进行代理 ，动态生成接口的实现类 ！（必须有接口）

【过程要点】：

1、 必须对接口生成代理

2、 采用Proxy对象，通过newProxyInstance方法为目标创建代理对象。

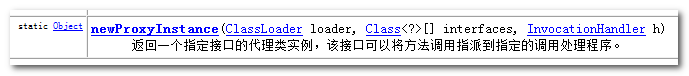
该方法接收三个参数 ：

（1）目标对象类加载器

（2）目标对象实现的接口

（3）代理后的处理程序InvocationHandler

使用 Proxy类提供 newProxyInstance 方法对目标对象接口进行代理



参数说明：

loader：定义代理类的类加载器

interfaces：代理类要实现的接口列表

h:指派方法调用的调用处理程序

3、 实现InvocationHandler 接口中 invoke方法，在目标对象每个方法调用时，都会执行invoke

【示例】：需求：对目标对象中存在保存和查询的方法，在执行保存的方法的时候，记录日志

第一步：新建web工程：spring4\_d02\_c04。创建包com.igeek

第二步：编写业务接口，接口中定义save()和find()的方法。

//接口（表示代理的目标接口）

**public** **interface** ICustomerService {

//保存

**public** **void** save();

//查询

**public** **int** find();

}

第三步：编写业务类，类要实现接口

//实现类

**public** **class** CustomerServiceImpl **implements** ICustomerService{

//保存

**public** **void** save() {

System.*out*.println("客户保存了。。。。。");

}

//查询

**public** **int** find() {

System.*out*.println("客户查询数量了。。。。。");

**return** 100;

}

}

第四步：使用JDK代理完成

代理工厂：

有三种方案完成JDK动态代理：

方案一：在内部实现new InvocationHandler()，指定匿名类

//专门用来生成jdk的动态代理对象的-通用

**public** **class** JdkProxyFactory{

//成员变量

**private** Object target;

//注入target目标对象

**public** JdkProxyFactory(Object target) {

**this**.target = target;

}

**public** Object getProxyObject(){

//参数1：目标对象的类加载器

//参数2：目标对象实现的接口

//参数3：回调方法对象

/\*\*方案一：在内部实现new InvocationHandler()，指定匿名类\*/

**return** Proxy.*newProxyInstance*(target.getClass().getClassLoader(), target.getClass().getInterfaces(), **new** InvocationHandler(){

**public** Object invoke(Object proxy, Method method, Object[] args)

**throws** Throwable {

//如果是保存的方法，执行记录日志操作

**if**(method.getName().equals("save")){

*writeLog*();

}

//目标对象原来的方法执行

Object object = method.invoke(target, args);//调用目标对象的某个方法，并且返回目标对象

**return** object;

}

});

}

//记录日志

**private** **static** **void** writeLog(){

System.*out*.println("增强代码：写日志了。。。");

}

}

方案二：传递内部类的对象，指定内部类

//专门用来生成jdk的动态代理对象的-通用

**public** **class** JdkProxyFactory{

//成员变量

**private** Object target;

//注入target

**public** JdkProxyFactory(Object target) {

**this**.target = target;

}

**public** Object getProxyObject(){

//参数1：目标对象的类加载器

//参数2：目标对象实现的接口

//参数3：回调方法对象

/\*\*方案二：传递内部类的对象，指定内部类\*/

**return** Proxy.*newProxyInstance*(target.getClass().getClassLoader(), target.getClass().getInterfaces(),**new** **MyInvocationHandler**());

}

//自己制定内部类:类的内部可以多次使用类型

**private** **class** **MyInvocationHandler** **implements** InvocationHandler{

**public** Object invoke(Object proxy, Method method, Object[] args)

**throws** Throwable {

//如果是保存的方法，执行记录日志操作

**if**(method.getName().equals("save")){

*writeLog*();

}

//目标对象原来的方法执行

Object object = method.invoke(target, args);//调用目标对象的某个方法，并且返回目标对象

**return** object;

}

}

//记录日志

**private** **static** **void** writeLog(){

System.*out*.println("增强代码：写日志了。。。");

}

}

方案三：直接使用调用类作为接口实现类，实现InvocationHandler接口

//专门用来生成jdk的动态代理对象的-通用

**public** **class** JdkProxyFactory **implements InvocationHandler**{

//成员变量

**private** Object target;

//注入target

**public** JdkProxyFactory(Object target) {

**this**.target = target;

}

**public** Object getProxyObject(){

//参数1：目标对象的类加载器

//参数2：目标对象实现的接口

//参数3：回调方法对象

/\*\*方案三：直接使用调用类作为接口实现类，实现InvocationHandler接口\*/

**return** Proxy.*newProxyInstance*(target.getClass().getClassLoader(), target.getClass().getInterfaces(),**this**);

}

//记录日志

**private** **static** **void** writeLog(){

System.*out*.println("增强代码：写日志了。。。");

}

//参数1：代理对象

//参数2：目标的方法对象

//参数3：目标的方法的参数

**public** Object invoke(Object proxy, Method method, Object[] args)

**throws** Throwable {

//如果是保存的方法，执行记录日志操作

**if**(method.getName().equals("save")){

*writeLog*();

}

//目标对象原来的方法执行

Object object = method.invoke(target, args);//调用目标对象的某个方法，并且返回目标对象

**return** object;

}

}

第五步：使用SpringTest.java进行测试

//目标：使用动态代理，对原来的方法进行功能增强,而无需更改原来的代码。

//JDK动态代理：基于接口的（对象的类型，必须实现接口！）

@Test

**public** **void** testJdkProxy(){

//target（目标对象）

ICustomerService target = **new** CustomerServiceImpl();

//实例化注入目标对象

JdkProxyFactory jdkProxyFactory = **new** JdkProxyFactory(target);

//获取Proxy Object代理对象:基于目标对象类型的接口的类型的子类型的对象

ICustomerService **proxy** = (ICustomerService)jdkProxyFactory.getProxyObject();

//调用目标对象的方法

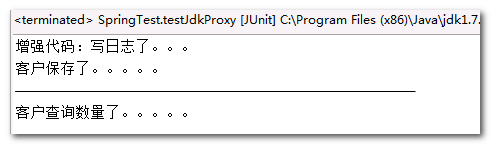
proxy.save();

System.*out*.println("————————————————————————————————————————");

proxy.find();

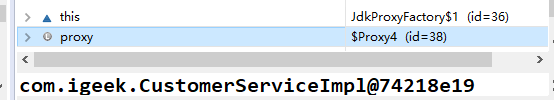
}

第六步：在控制台查看输出结果



从结果上看出：在保存方法的前面，输入了日志增强。

最后，使用断点查看JDK代理，生成的代理对象



说明

Interface ICustomerService{

//目标接口

}

Class CustomerServiceImpl implements ICustomerService{

//目标类实现接口

}

JDK代理对接口代理

Class $Proxy2 implements ICustomerService{

//JDK代理类是目标接口的实现类

**ICustomerService** **customerService = new CustomerServiceImpl();**

public void save() {

**writeLog()**

**customerService.save();**

}

public int find() {

**int returnValue = customerService.find();**

**return returnValue;**

}

//记录日志

private static void writeLog(){

System.out.println("增强代码：写日志了。。。");

}

}

注意：

JDK动态代理的缺点： 只能面向接口代理，不能直接对目标类进行代理 ，如果没有接口，则不能使用JDK代理。

## Cglib动态代理

Cglib的引入为了解决类的直接代理问题(生成代理子类)，不需要接口也可以代理 ！

什么是cglib ？

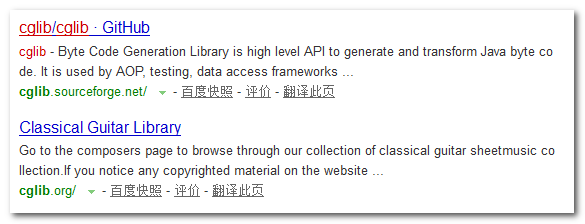
CGLIB(Code Generation Library)是一个开源项目！是一个强大的，高性能，高质量的Code生成类库，它可以在运行期扩展Java类与实现Java接口。

百度百科：参考网站

http://baike.baidu.com/link?url=A19bwp5SinV\_KZPV6TMorLe4oGpPVdAHgdRBJZC3HCuVqNm7JaDCnkOpvxdF4bnq-f9Wni4CBm5rZFiygUgHCa



该代理方式需要相应的jar包，但不需要导入。因为Spring core包已经包含cglib ，而且同时包含了cglib 依赖的asm的包（动态字节码的操作类库）



【示例】：需求：对目标对象中存在保存和查询的方法，在执行保存的方法的时候，记录日志

第一步：将spring的核心jar导入进来，因为spring的包，包含了cglib的包



将spring的core包引进来（他包含了cglib）

第二步：创建包com.igeek.cglib,编写业务类，创建类ProductService.java，类不需要实现接口

//没有接口的类

**public** **class** ProductService {

**public** **void** save() {

System.*out*.println("商品保存了。。。。。");

}

**public** **int** find() {

System.*out*.println("商品查询数量了。。。。。");

**return** 99;

}

}

第三步：使用cglib代理，创建类CglibProxyFactory.java

//cglib动态代理工厂:用来生成cglib代理对象

**public** **class** CglibProxyFactory **implements** **MethodInterceptor**{

//声明一个代理对象引用

**private** Object target;

//注入代理对象

**public** CglibProxyFactory(Object target) {

**this**.target = target;

}

//获取代理对象

**public** Object getProxyObject(){

//1.代理对象生成器(工厂思想)

Enhancer enhancer = **new** Enhancer();

// 类加载器

enhancer.setClassLoader(target.getClass().getClassLoader());

//2.在增强器上设置两个属性

//设置要生成代理对象的目标对象：生成的目标对象类型的子类型

enhancer.setSuperclass(target.getClass());

//设置回调方法

enhancer.setCallback(**this**);

// Callback

//3.创建获取对象

**return** enhancer.create();

}

//回调方法（代理对象的方法）

//参数1：代理对象

//参数2：目标对象的方法对象

//参数3：目标对象的方法的参数的值

//参数4：代理对象的方法对象

**public** Object intercept(Object proxy, Method method, Object[] args,

MethodProxy methodProxy) **throws** Throwable {

//如果是保存的方法，执行记录日志操作

**if**(method.getName().equals("save")){

*writeLog*();

}

//目标对象原来的方法执行

Object object = method.invoke(target, args);//调用目标对象的某个方法，并且返回目标对象

**return** object;

}

//写日志的增强功能

//Advice通知、增强

//记录日志

**private** **static** **void** writeLog(){

System.*out*.println("增强代码：写日志了。。。");

}

}

第四步：测试代码，使用SpringTest.java进行测试

//cglib动态代理：可以基于类（无需实现接口）生成代理对象

@Test

**public** **void** testCglibProxy(){

//target目标：

ProductService target = **new** ProductService();

//weave织入，生成proxy代理对象

//代理工厂对象，注入目标

CglibProxyFactory cglibProxyFactory = **new** CglibProxyFactory(target);

//获取proxy:思考：对象的类型

//代理对象，其实是目标对象类型的子类型

ProductService proxy=(ProductService) cglibProxyFactory.getProxyObject();

//调用代理对象的方法

proxy.save();

System.*out*.println("————————————————————————————————————————");

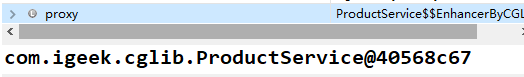
proxy.find();

}

第五步：控制台输出结果



最后，使用断点查看cglib代理，生成的代理对象



说明

Class ProductService{

//目标类

}

Cglib对类代理（动态生成）

Class ProductService$$EnhancerByCGLIB$$df9980d0 extends ProductService{

//CGLIB代理类是目标类的子类

**ProductService** **productService= new ProductService();**

public void save() {

**writeLog()**

**productService.save();**

}

public int find() {

**int returnValue = productService.find();**

**return returnValue;**

}

//记录日志

private static void writeLog(){

System.out.println("增强代码：写日志了。。。");

}

}

## 代理知识小结

区别：

* Jdk代理：基于接口的代理，一定是基于接口，会生成目标对象的接口类型的子对象。
* Cglib代理：基于类的代理，不需要基于接口，会生成目标对象类型的子对象。

代理知识总结：

* spring在运行期，生成动态代理对象，不需要特殊的编译器.
* spring有两种代理方式：

1.若目标对象实现了若干接口，spring使用JDK的java.lang.reflect.Proxy类代理。（默认）

2.若目标对象没有实现任何接口，spring使用CGLIB库生成目标对象的子类。

* 使用该方式时需要注意:

1.对接口创建代理优于对类创建代理，因为会产生更加松耦合的系统，所以spring默认是使用JDK代理。

对类代理是让遗留系统或无法实现接口的第三方类库同样可以得到通知，这种方式应该是备用方案。

2.标记为final的方法不能够被通知。spring是为目标类产生子类。任何需要被通知的方法都被复写，将通知织入。final方法是不允许重写的。

3.spring只支持方法连接点：不提供属性接入点，spring的观点是属性拦截破坏了封装。

面向对象的概念是对象自己处理工作，其他对象只能通过方法调用的得到的结果。

提示：

* Spring AOP 优先对接口进行代理 （使用Jdk动态代理）
* 如果目标对象没有实现任何接口，才会对类进行代理 （使用cglib动态代理）

# AspectJ 切面编程（xml方式）

百度百科文档：

http://baike.baidu.com/link?url=2h5jS5pGRtcNexdPQBt1CSgPXpqFyyCu0WR-dJKh9mLxjdpLloT9XsY7pQIwGdOLQB1n7Shfoqr65JXiOtmDEK



早期传统Spring1.2 AOP编程 ， 确定目标 --- 编写通知(通知需要实现一个接口)

---- 配置ProxyFactoryBean生成代理对象，配置非常复杂，

spring2.0 支持AspectJ 语法 ，简化AOP开发。

开发方法还是三步：

1. 确定目标对象（bean）
2. 编写通知，对目标对象增强（advice）
3. 配置切入点（pointcut）、切面（aspect）

### AspectJ 提供Advice类型

普通的pojo即可。（不需要实现接口）

AspectJ提供不同的通知类型：

* Before 前置通知，相当于BeforeAdvice
* AfterReturning 后置通知，相当于AfterReturningAdvice
* Around 环绕通知，相当于MethodInterceptor
* AfterThrowing抛出通知，相当于ThrowAdvice
* After 最终final通知，不管是否异常，该通知都会执行
* DeclareParents 引介通知，相当于IntroductionInterceptor (不要求掌握)

相比传统Spring AOP通知类型多了 After最终通知 （类似 finally ）。

实现步骤：

第一步：确定目标对象，即确定bean对象

第二步：advice通知（编写）

第三步：配置切面（包括切入点），让切入点关联通知

第一步：新建web项目spring4\_d02\_c05

导入jar包 (11个)

基本spring开发 6个 （4+2）（包含配置文件）

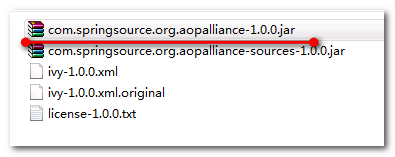
Spring测试集成 1个

Spring AOP编程 4个 （2对）

* com.springsource.org.aopalliance-1.0.0.jar： AOP联盟规范
* spring-aop-4.2.4.RELEASE.jar：springAOP 整合扩展
* com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar：aspectJ官方包
* spring-aspects-4.2.4.RELEASE.jar：spring对aspectJ集成

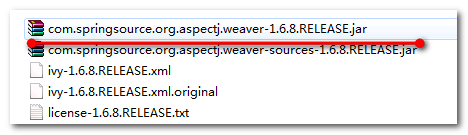
AOP联盟包：

路径：spring-framework-3.0.2.RELEASE-dependencies\org.aopalliance\com.springsource.org.aopalliance\1.0.0

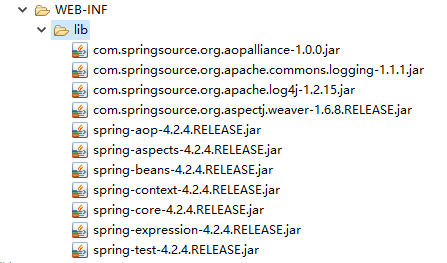


Aspectj包：

路径：spring-framework-3.0.2.RELEASE-dependencies\org.aspectj\com.springsource.org.aspectj.weaver\1.6.8.RELEASE



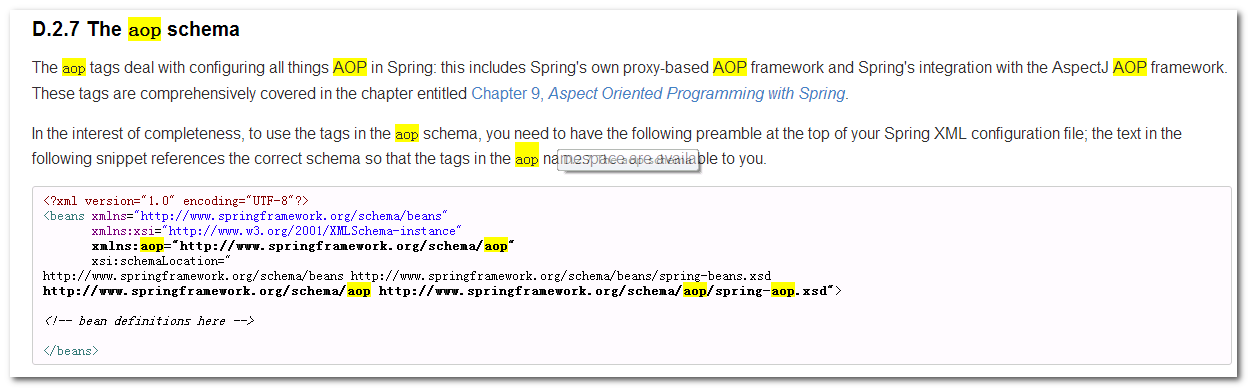
共需要导入的jar包



在src下，创建applicationContext.xml。

第一步： 引用aop的名称空间

查看：spring-framework-4.2.4.RELEASE\docs\spring-framework-reference\html中的xsd-config.html，搜素aop，发现：



需要我们引入aop的文件约束。如图

<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*

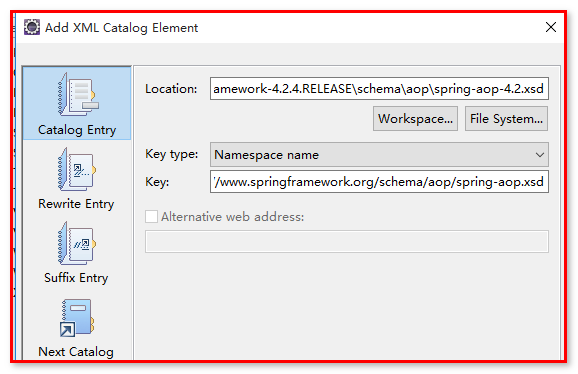
*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop.xsd"*>

配置aop提示



第二步：编写Before 前置通知Advice增强 ：

创建包：com.igeek,复制上个项目中的

CustomerServiceImpl.java,ICustomerService.java,ProductService.java到包中

配置bean

<!-- 1.确定了要增强的target对象 -->

<!-- 对于spring来说，目标对象：就是bean对象 -->

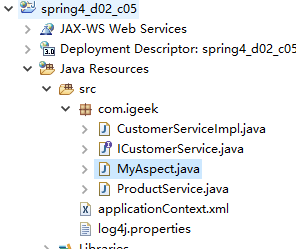
<!-- 基于接口类 -->

<bean id=*"customerService"* class=*"com.igeek.CustomerServiceImpl"*/>

<!-- 基于一般类 -->

<bean id=*"productService"* class=*"com.igeek.ProductService"*/>

创建类：MyAspect.java



编写MyAspect.java

//aspectj的advice通知增强类，无需实现任何接口

**public** **class** MyAspect {

//前置通知

//普通的方法。方法名随便，但也不能太随便，一会要配置

**public** **void** firstbefore(){

System.*out*.println("------------第一个个前置通知执行了。。。");

}

}

将前置通知配置到spring的容器中

<!-- 2.配置advice通知增强 -->

<bean id=*"myAspectAdvice"* class=*"com.igeek.MyAspect"*/>

### 配置切入点和切面（让切入点关联通知）

第三步：配置切面（包括切入点），让切入点关联通知

核心配置文件applicationContext.xml中添加：

<!-- 3：配置aop -->

<aop:config>

<!-- 切入点:拦截哪些bean的方法 -->

<aop:pointcut expression=*"bean(\*Service)"* id=*"myPointcut"*/>

<!--

切面：要对哪些方法进行怎样的增强

aop:aspect:aspejctj的方式！

ref:配置通知

-->

<aop:aspect ref=*"myAspectAdvice"*>

<!-- 第一个前置通知 ：在访问目标对象方法之前，先执行通知的方法

method：advice类中的方法名，

pointcut-ref="myPointcut"：注入切入点

目的是让通知关联切入点

-->

<aop:before method=*"firstbefore"* pointcut-ref=*"myPointcut"*/>

</aop:aspect>

</aop:config>

使用SpringTest测试代码：

//springjunit集成测试

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations="classpath:applicationContext.xml")

**public** **class** SpringTest {

//注入要测试bean

@Autowired

**private** ICustomerService customerService;

@Autowired

**private** ProductService productService;

//测试

@Test

**public** **void** test(){

//基于接口

customerService.save();

customerService.find();

//基于类的

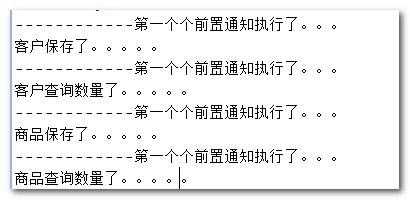
productService.save();

productService.find();

}

}

测试结果：



和传统的aop配置相比，更灵活，advice不需要实现接口，简单的pojo就可以了；一个通知可以增强多个连接点，一个连接点可以被多次增强。

【扩展优化】：

1．将切入点放入aspectj标签里面写，同时配置多个通知方法

<!-- 2.配置advice通知增强 -->

<bean id=*"myAspectAdvice"* class=*"cn.itcast.spring.c\_aspectjaop.MyAspect"*/>

<!-- 3：配置aop -->

<aop:config>

<!--

切面：要对哪些方法进行怎样的增强

aop:aspect:aspejctj的方式！

ref:配置通知

-->

<aop:aspect ref=*"myAspectAdvice"*>

<aop:pointcut expression=*"bean(\*Service)"* id=*"myPointcut"*/>

<!-- 第一个前置通知 ：在访问目标对象方法之前，先执行通知的方法

method：advice类中的方法名，

pointcut-ref="myPointcut"：注入切入点

目的是让通知关联切入点

-->

<aop:before method=*"firstbefore"* pointcut-ref=*"myPointcut"*/>

<aop:before method=*"firstbefore2"* pointcut-ref=*"myPointcut"*/>

</aop:aspect>

</aop:config>

2.配置多个通知方法：

//aspectj的advice通知增强类，无需实现任何接口

**public** **class** MyAspect {

//前置通知

//普通的方法。方法名随便，但也不能太随便，一会要配置

**public** **void** firstbefore(){

System.*out*.println("------------第一个前置通知执行了。。。");

}

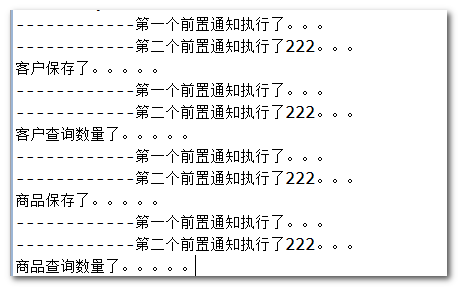
**public** **void** firstbefore2(){

System.*out*.println("------------第二个前置通知执行了222。。。");

}

}

1. 执行结果：表示在执行目标对象方法之前执行



AspectJ切面编程，相比于传统的SpringAOP，定义的通知方法更多。

### 分析各种通知应用

### Before前置通知

案例应用： 实现权限控制 （即：权限不足的时候，抛出异常）、 记录方法调用信息日志

第一步：配置MyAspect类（切面），配置before方法（通知）

//aspectj的advice通知增强类，无需实现任何接口

**public** **class** MyAspect {

//前置通知的：方法运行之前增强

//应用： 权限控制 （权限不足，抛出异常）、 记录方法调用信息日志

//参数：org.aspectj.lang.JoinPoint

//参数：连接点对象（方法的包装对象:方法，参数，目标对象）

**public** **void** before(JoinPoint joinPoint){

//分析：抛出异常拦截的

//当前登录用户

String loginName = "Rose";

System.*out*.println("方法名称："+joinPoint.getSignature().getName());

System.*out*.println("目标对象："+joinPoint.getTarget().getClass().getName());

System.*out*.println("代理对象："+joinPoint.getThis().getClass().getName());

//判断当前用户有没有执行方法权限

**if**(joinPoint.getSignature().getName().equals("find")){

**if**(!loginName.equals("admin")){

//只有超级管理员admin有权限，其他人不能执行某个方法，比如查询方法

**throw** **new** RuntimeException("您没有权限执行方法："+joinPoint.getSignature().getName()+"，类型为："+joinPoint.getTarget().getClass().getName());

}

}

}

}

通过JoinPoint 连接点对象，获取目标对象信息 ！

这里注意：引包不要引错了，使用aspectj中的连接点（org.aspectj.lang.JoinPoint）：



第二步：Spring容器中配置，配置applicationContext.xml

<!-- 3：配置aop -->

<aop:config>

<aop:aspect ref=*"myAspectAdvice"*>

<aop:pointcut expression=*"bean(\*Service)"* id=*"myPointcut"*/>

<!-- 前置通知 -->

<aop:before method=*"before"* pointcut-ref="*myPointcut*" />

</aop:aspect>

</aop:config>

第三步：使用SpringTest.java进行测试：

//springjunit集成测试

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations="classpath:applicationContext.xml")

**public** **class** SpringTest {

//注入要测试bean

@Autowired

**private** ICustomerService customerService;

@Autowired

**private** ProductService productService;

//测试

@Test

**public** **void** test(){

//基于接口

customerService.save();

customerService.find();

//基于类的

productService.save();

productService.find();

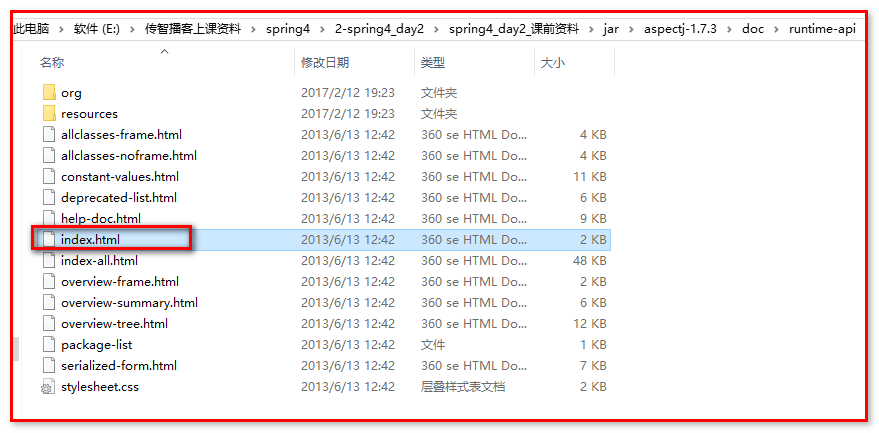
}

}

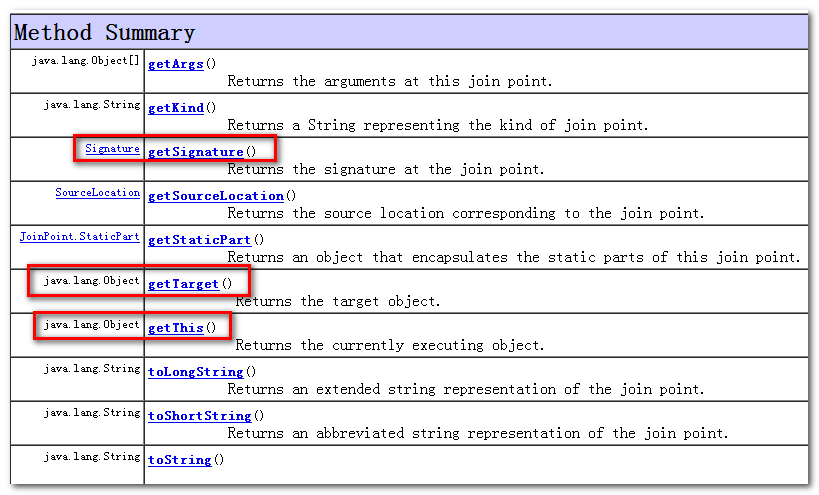
提示：

Aspectj的文档

JoinPoint使用参考文档：资料\jar\aspectj-1.7.3\doc\runtime-api\index.html







提示：和传统aop的对比发现，aspectj更灵活，一个类中可以写N个增强方法，但传统的只能是一个类对应一个方法。

### AfterReturing 后置通知

特点：在目标方法运行后，返回值后执行通知增强代码逻辑。

应用场景：与业务相关的，如网上营业厅查询余额后，自动下发短信功能。

分析： 后置通知可以获取到目标方法返回值，如果想对返回值进行操作，使用后置通知（但不能修改目标方法返回 ）

第一步：配置MyAspect类（切面），配置afterReturing方法（通知）

//aspectj的advice通知增强类，无需实现任何接口

**public** **class** MyAspect {

//应用场景：与业务相关的，如网上营业厅查询余额后，自动下发短信。

//后置通知：会在目标方法执行之后调用通知方法增强。

//参数1：连接点对象（方法的包装对象:方法，参数，目标对象）

//参数2：目标方法执行后的返回值,类型是object，“参数名”随便，但也不能太随便，一会要配置

**public** **void** afterReturing(JoinPoint joinPoint,Object returnVal){

//下发短信:调用运行商的接口，短信猫。。。

System.*out*.println("-++++++++-后置通知-当前下发短信的方法"+"-尊敬的用户，您调用的方法返回余额为："+returnVal);

//同时可以可以在下发后，记录日志：

System.*out*.println("----日志记录，操作的类型："+joinPoint.getTarget().getClass().getSimpleName()

+"，操作的方法："+joinPoint.getSignature().getName()

);

}

}

第二步：Spring容器中配置，配置applicationContext.xml

<!-- 3：配置aop -->

<aop:config>

<aop:aspect ref=*"myAspectAdvice"*>

<aop:pointcut expression=*"bean(\*Service)"* id=*"myPointcut"*/>

<!-- 后置通知

returning:配置方法中的参数名字，与通知方法的第二个参数的名字，名字必须对应。

在运行的时候，spring会自动将返回值传入该参数中。

-->

<aop:after-returning method=*"afterReturing"* returning=*"returnVal"* pointcut-ref=*"myPointcut"*/>

</aop:aspect>

</aop:config>

第三步：使用SpringTest.java进行测试：

//springjunit集成测试

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations="classpath:applicationContext.xml")

**public** **class** SpringTest {

//注入要测试bean

@Autowired

**private** ICustomerService customerService;

@Autowired

**private** ProductService productService;

//测试

@Test

**public** **void** test(){

//基于接口

customerService.save();

customerService.find();

//基于类的

productService.save();

productService.find();

}

}

### Around 环绕通知

特点：目标执行前后，都进行增强（控制目标方法执行）

应用场景：日志、缓存、权限、性能监控、**事务管理**

**增强代码的方法要求：**

接受的参数：ProceedingJoinPoint（可执行的连接点）

返回值：Object返回值（即目标对象方法的返回值）

抛出Throwable异常。

【示例】

第一步：配置MyAspect类（切面），配置around方法（通知）

//aspectj的advice通知增强类，无需实现任何接口

**public** **class** MyAspect {

//应用场景：日志、缓存、权限、性能监控、事务管理

//环绕通知：在目标对象方法的执行前+后，可以增强

//参数：可以执行的连接点对象ProcessdingJoinPoint（方法），特点是调用proceed()方法可以随时随地执行目标对象的方法（相当于目标对象的方法执行了）

//必须抛出一个Throwable

**public** **Object** around(ProceedingJoinPoint proceedingJoinPoint) **throws** Throwable{

//目标：事务的控制：

//开启事务：

System.*out*.println("-----开启了事务。。。。。。。。。");

//执行了目标对象的方法

Object resultObject = proceedingJoinPoint.proceed();

//结束事务

System.*out*.println("-----提交了事务。。。。。。。。。");

**return** resultObject;//目标对象执行的结果

}

}

第二步：Spring容器中配置，配置applicationContext.xml

<!-- 3：配置aop -->

<aop:config>

<aop:aspect ref=*"myAspectAdvice"*>

<aop:pointcut expression=*"bean(\*Service)"* id=*"myPointcut"*/>

<!-- 环绕通知 -->

<aop:around method=*"around"* pointcut-ref=*"myPointcut"*/>

</aop:aspect>

</aop:config>

第三步：使用SpringTest.java进行测试：

//springjunit集成测试

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations="classpath:applicationContext.xml")

**public** **class** SpringTest {

//注入要测试bean

@Autowired

**private** ICustomerService customerService;

@Autowired

**private** ProductService productService;

//测试

@Test

**public** **void** test(){

//基于接口

customerService.save();

customerService.find();

//基于类的

productService.save();

productService.find();

}

}

### AfterThrowing 抛出通知（异常通知）

作用：目标代码出现异常，通知执行。记录异常日志、通知管理员（短信、邮件）

应用场景：处理异常（一般不可预知），记录日志

【示例】

第一步：配置MyAspect类（切面），配置aterThrowing方法（通知）

//aspectj的advice通知增强类，无需实现任何接口

**public** **class** MyAspect {

//作用：目标代码出现异常，通知执行。记录异常日志、通知管理员（短信、邮件）

//只有目标对象方法抛出异常，通知才会执行

//参数1：静态连接点（方法对象）

//参数2：目标方法抛出的异常，参数名随便，但也不能太随便

**public** **void** afterThrowing(JoinPoint joinPoint,Throwable ex){

//一旦发生异常，发送邮件或者短信给管理员

System.*out*.println("++管理员您好，"+joinPoint.getTarget().getClass().getName()+"的方法："

+joinPoint.getSignature().getName()+"发生了异常，异常为："+ex.getMessage());

}

}

第二步：Spring容器中配置，配置applicationContext.xml

<!-- 3：配置aop -->

<aop:config>

<aop:aspect ref=*"myAspectAdvice"*>

<aop:pointcut expression=*"bean(\*Service)"* id=*"myPointcut"*/>

<!-- 抛出通知

throwing:通知中的方法的第二个参数，异常类型的参数的名字，在运行的时候，spring会自动将异常传入该参数中。-->

<aop:after-throwing method=*"afterThrowing"* throwing=*"ex"* pointcut-ref=*"myPointcut"*/>

</aop:aspect>

</aop:config>

第三步：使用SpringTest.java进行测试：

//springjunit集成测试

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations="classpath:applicationContext.xml")

**public** **class** SpringTest {

//注入要测试bean

@Autowired

**private** ICustomerService customerService;

@Autowired

**private** ProductService productService;

//测试

@Test

**public** **void** test(){

//基于接口

customerService.save();

customerService.find();

//基于类的

productService.save();

productService.find();

}

}

此时发现通知的方法并没有执行。

那我们在目标对象的方法中故意抛出异常，大家看看效果

测试：

在ProductService.java中save的方法中，制造异常：

//没有接口的类

**public** **class** ProductService {

**public** **void** save() {

System.*out*.println("商品保存了。。。。。");

//故意制造异常

int d = 1/0;

}

**public** **int** find() {

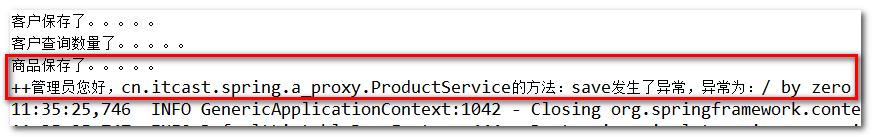
System.*out*.println("商品查询数量了。。。。。");

**return** 99;

}

}

查看测试结果：



### After 最终通知

作用：不管目标方法是否发生异常，最终通知都会执行（类似于finally代码功能）

应用场景：释放资源 （关闭文件、 关闭数据库连接、 网络连接、 释放内存对象 ）

【示例】

第一步：配置MyAspect类（切面），配置after方法（通知）

//aspectj的advice通知增强类，无需实现任何接口

**public** **class** MyAspect {

//应用场景：释放资源 （关闭文件、 关闭数据库连接、 网络连接、 释放内存对象 ）

//最终通知：不管是否有异常都会执行

**public** **void** after(JoinPoint joinPoint){

//释放数据库连接

System.*out*.println("数据库的connection被释放了。。。。。,执行的方法是："+joinPoint.getSignature().getName());

}

}

第二步：Spring容器中配置，配置applicationContext-aspectJ.xml

<!-- 3：配置aop -->

<aop:config>

<aop:aspect ref=*"myAspectAdvice"*>

<aop:pointcut expression=*"bean(\*Service)"* id=*"myPointcut"*/>

<!-- 最终通知 -->

<aop:after method=*"after"* pointcut-ref=*"myPointcut"*/>

<!-- **以上代码也可以写成**：pointcut切入点表达式:只能给一个通知方法来用，相当于省略了<aop:pointcut expression="bean(\*Service)" id="myPointcut"/>

<aop:after method="after" pointcut="bean(\*Service)"/>-->

</aop:aspect>

</aop:config>

第三步：使用SpringTest.java进行测试：

//springjunit集成测试

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations="classpath:applicationContext.xml")

**public** **class** SpringTest {

//注入要测试bean

@Autowired

**private** ICustomerService customerService;

@Autowired

**private** ProductService productService;

//测试

@Test

**public** **void** test(){

//基于接口

customerService.save();

customerService.find();

//基于类的

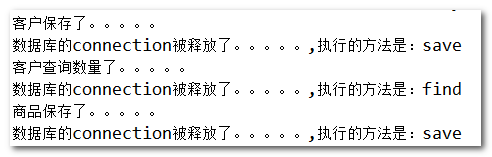
productService.save();

productService.find();

}

}

查看测试结果：



五种通知小结：

（1）只要掌握Around（环绕通知）通知类型，就可实现其他四种通知效果。

（2）因为你可以在环绕通知的方法中编写如下代码：

try {

//前置通知

Object result = proceedingJoinPoint.proceed();

//后置通知

}catch(Exception){

//抛出通知

}finally{

//最终通知

}

### 各种Advice方法可接收的参数和返回值小结（参考）

方法格式：

public returnType method (param)

public 返回值类型 方法名 (参数类型 参数名)

返回值类型：void和Object

方法名：任意名称（但是也不能太随意）

参数类型：

\* 参数类型为JoinPoint接口类型，返回值类型为void

\* 参数类型为ProceedingJoinPoint接口类型，返回值类型为Object

具体为：

|  |  |  |  |
| --- | --- | --- | --- |
| 通知类型 | 输入参数(可选) | 返回值类型 | 其他 |
| Before前置通知 | JoinPoint（静态连接点信息） | void |  |
| AfterReturning后置通知 | JoinPoint, Object | void |  |
| Around环绕通知 | ProceedingJoinPoint（可执行的连接点信息） | Object | throws Throwable |
| AfterThrowing抛出通知 | JoinPoint, Throwable | void |  |
| After最终通知 | JoinPoint | void |  |

总结：

1. 注解方式装配bean对象
2. 混合配置
3. 测试集成 (@runwith @ContextConfiguration(核心配置))---熟悉
4. AOP的思想（如何实现），AOP在哪些地方使用？
5. aop的原理（动态代理：jdk，cglib）
6. AspectJ 切面编程（xml方式）
7. 各种通知应用