**Spring**

内容：

Part I：Spring框架入门、IoC、Spring Web集成、Spring Junit集成

Part II： AOP面向切面编程、AspectJ集成配置、JdbcTemplate

Part III：Spring声明式事务管理、ssh整合

Part IV：综合练习SSH、jQuery Ajax、分页

**Part II**

Part II知识点：

AOP相关概念

AOP底层实现机制（动态代理）

Spring传统AOP编程案例

Spring结合AspectJ实现AOP

JdbcTemplate

# **1.AOP相关概念**

## **1.1 AOP概念**

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

AOP采取横向抽取机制，取代传统纵向继承体系中重复代码编写的方式，多用于性能监视、事务管理、权限控制、缓存、日志记录等。

两种增强方式的比较：

继承方式的增强：纵向，父子关系；代码侵入，耦合度高

AOP增强：代理（横向，兄弟关系）；代码非侵入，耦合度低

AOP：基于动态代理机制，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，从而对原有业务的增强

【扩展1】

继承方式的增强：

要对保存操作增加日志功能

**class** UserDao {

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

//数据库操作

}

}

Dao层很多这样的操作，所以抽取到基础类

**abstract** **class** BaseDao {

**public** **void** log() {

//日志记录

}

}

对UserDao使用继承的方法增强

**class** UserDao **extends** BaseDao{

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

//数据库操作

//日志记录，代码侵入

log();

}

}

【扩展2】

AOP是OOP思想的一种延续（OOP即Object Oriented Programming，面向对象编程）

AOP思想符合OCP开闭原则：Softeware entities like classes,modules and functions should be open for extension but closed for modifications.对扩展开放，对修改关闭，是面向对象的五大原则之一。

很多设计模式，如简单工厂模式等都符合该原则。为了使程序的扩展性好，而不直接修改原有的代码。符合该原则的关键在于抽象

## **1.2 AOP两种实现方式**

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。

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。

Spring内部支持两套AOP编程的方案：

* Spring 1.2 开始，支持AOP编程（即传统方式的AOP编程）。编程复杂，但能较好的理解AOP原理
* Spring 2.0 之后，支持第三方AOP框架（AspectJ）。另一种方式的AOP编程，实现简单，开发推荐

## **1.3 AOP相关概念**

AOP的相关概念：

Target（目标对象）：需要代理的目标对象

Joinpoint（连接点）：指目标对象中的方法，spring只支持方法类型的连接点。（方法是“点”，即AOP的最小操作单位是方法，不支持方法内的增强）

Pointcut（切入点）：指要对哪些joinpoint进行拦截和增强

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）：即具体的增强，对Pointcut要做的事情。分为前置通知、后置通知、异常通知、最终通知、环绕通知

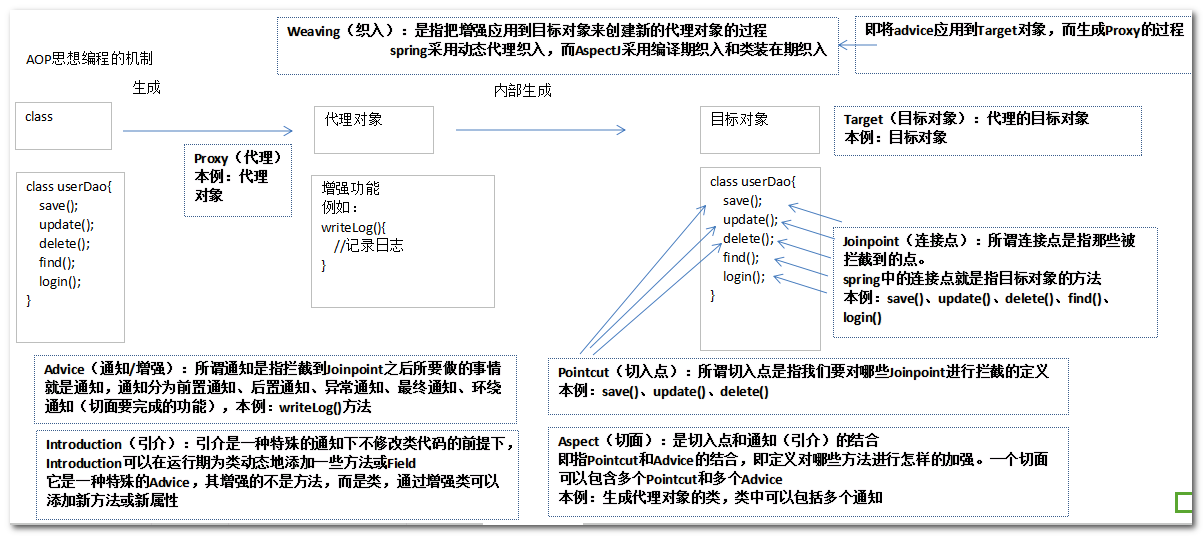
Aspect（切面）：Advice+Pointcut，即定义在何处（Pointcut）做何事（Advice）

Weaving（织入）：把Aspect应用到Target，创建代理对象的过程。切面在指定的连接点织入到目标对象

Introduction（引入）：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

通过案例理解AOP相关概念

案例：UserDao中有5个方法：save()、update()、delete()、find()、login()，在访问UserDao的save()、update()、delete()之前，进行记录日志的操作



Aspect切面：切入点 Pointcut（save()，update()，delete()） + Advice（writeLog()），即对save()，update()，delete()进行writeLog()的增强

# **2.两种动态代理**

AOP就是为目标对象创建代理对象的过程。Spring AOP基于两种动态代理机制：JDK动态代理、CGLIB动态代理

【扩展】动态代理和静态代理的区别

动态代理：在jvm内部，运行时动态生成代理类，并不是真正存在的类，格式一般为：Proxy$$（Proxy$$Customer）

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

## **2.1 JDK动态代理原理**

JDK动态代理，对目标对象的接口进行代理，动态生成接口的实现类作为代理类（父子关系）

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

【过程要点】

1.目标对象必须实现了接口

2.使用JDK提供的Proxy类，通过newProxyInstance()方法为目标对象创建代理对象

public static [Object](https://www.oschina.net/uploads/doc/javase-6-doc-api-zh_CN/java/lang/Object.html" \o "java.lang 中的类) **newProxyInstance**([ClassLoader](https://www.oschina.net/uploads/doc/javase-6-doc-api-zh_CN/java/lang/ClassLoader.html" \o "java.lang 中的类) loader, [Class](https://www.oschina.net/uploads/doc/javase-6-doc-api-zh_CN/java/lang/Class.html" \o "java.lang 中的类)<?>[] loader, [InvocationHandler](https://www.oschina.net/uploads/doc/javase-6-doc-api-zh_CN/java/lang/reflect/InvocationHandler.html" \o "java.lang.reflect 中的接口) h) throws [IllegalArgumentException](https://www.oschina.net/uploads/doc/javase-6-doc-api-zh_CN/java/lang/IllegalArgumentException.html" \o "java.lang 中的类)

该方法的三个参数 ：

代理类的类加载器loader

代理类要实现的接口列表loader

代理后的处理程序h

此方法相当于：Proxy.getProxyClass(loader, interfaces).getConstructor(new Class[] { InvocationHandler.class }).newInstance(new Object[] { handler });

3、在第三个参数中，实现InvocationHandler接口中完成invoke()。在目标对象每个方法调用时，都会执行invoke

【示例】

目标对象中有保存和查询方法，在执行保存方法时，记录日志

* 第一步：新建web工程：study\_spring02，创建包com.test.proxy.jdk
* 第二步：编写业务接口和业务类（类要实现接口）

CustomerService.java

**package** com.test.proxy.jdk;

**public** **interface** CustomerService {

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

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

}

CustomerServiceImpl.java（要实现接口）

**package** com.test.proxy.jdk;

**public** **class** CustomerServiceImpl **implements** CustomerService {

@Override

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

System.***out***.println("客户保存了...");

}

@Override

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

System.***out***.println("客户查询了...");

**return** 100;

}

}

* 第三步：使用JDK代理完成增强

这里使用工厂模式，创建代理工厂并提供获得代理对象的方法，复用性高

代理工厂：有3种方案完成JDK动态代理（实际是“为接口创建实例”的3种方式，大同小异，个人喜欢匿名内部类）

**package** com.test.proxy.jdk;

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

//成员变量-目标对象

**private** Object target;

//有参构造（如果不用有参构造传入目标对象，用setter也行）

**public** JdkProxyFactory(Object target) {

**this**.target = target;

}

//提供获取代理对象的方法

//方案一：创建InvocationHandler接口的匿名内部类

/\*public Object getProxyInstance() {

//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

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

@Override

//proxy：代理对象；method：代理对象中与目标对象对应的方法(对象)；args：方法参数

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

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

System.out.println("记录日志...");//日志记录增强

}

Object object = method.invoke(target, args);//目标对象的原有方法，并且返回返回值

return object;

}

});

}\*/

//方案二：不使用匿名内部类，自定义InvocationHandler接口内部实现类

/\*public Object getProxyInstance() {

//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

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

}

//自定义接口实现的内部类，代替匿名内部类（private不能修饰类，但可以修饰内部类，相当于成员变量）

private class MyInvocationHandler implements InvocationHandler {

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

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

System.out.println("记录日志...");//日志记录增强

}

Object object = method.invoke(target, args);//目标对象的原有方法，并且返回返回值

return object;

}

}\*/

//方案三：本工厂类自己实现InvocationHandler接口

**public** Object getProxyInstance() {

//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

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

}

@Override

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

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

System.***out***.println("记录日志...");//日志记录增强

}

Object object = method.invoke(target, args);//目标对象的原有方法，并且返回返回值

**return** object;

}

}

* 第四步：SpringTest.java测试

**package** com.test.proxy.jdk;

//使用JDK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

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

@Test

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

CustomerService target = **new** CustomerServiceImpl(); //目标对象

JdkProxyFactory jdkProxyFactory = **new** JdkProxyFactory(target); //创建代理工厂实例，有参构造注入目标对象

CustomerService proxy = (CustomerService)jdkProxyFactory.getProxyInstance(); //获取代理对象

//调用目标对象的方法

proxy.save();

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

proxy.find();

}

}

【输出结果】

记录日志...

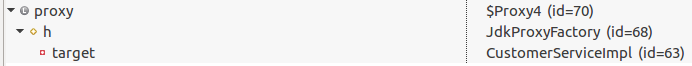
客户保存了...

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

客户查询了...

从结果上看出：在save()的前面，输入了日志增强

使用断点查看JDK代理，生成的代理对象CustomerService proxy



【说明】

JDK代理是对接口进行代理，JDK代理类是目标接口的实现类，与目标类是兄弟关系，所以代理类中也有save,find方法。只不过只对save()增强了，而find()不变。相当于：

class $Proxy4 implements CustomerService{

ICustomerService customerService = new CustomerServiceImpl();

public void save() {

writeLog()

customerService.save();

}

public int find() {

int returnValue = customerService.find();

return returnValue;

}

private static void writeLog(){

System.out.println("记录日志了...");

}

}

## **2.2 cglib动态代理原理**

JDK动态代理要求目标对象必须实现接口。cglib的引入就是为了解决类的直接代理问题，即cglib不需要接口也可以代理

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。

该代理方式需要相应的jar包，但不需要单独导入，Spring core包中已经包含了cglib，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】

需求同上：目标对象中存在保存和查询的方法，在执行保存时，才记录日志

* 第一步：导入spring-core的jar（包含了cglib和其依赖的asm）

spring-core-4.3.2.RELEASE.jar

* 第二步：编写业务类ProductServiceImpl.java，此时不实现接口

**package** com.test.proxy.cglib;

**public** **class** ProductServiceImpl {

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

System.***out***.println("商品保存了...");

}

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

System.***out***.println("商品查询了...");

**return** 22;

}

}

* 第三步：使用cglib代理，同样使用工厂模式，创建代理工厂类CglibProxyFactory.java

这里使用cglib提供的Enhancer类来获取增强后的代理对象。

该类是ClassGenerator接口的实现类，可以对类/接口进行动态代理（setSuperclass()，setInterfaces()）

MethodInterceptor是CallBack接口的子接口，设置回调函数enhancer.setCallback时，使用的是匿名内部类（当然也可以自定义内部类或该类自己实现MethodInterceptor接口传this，跟JDK代理中的InvocationHandler一样）

**package** com.test.proxy.cglib;

**public** **class** CglibProxyFactory {

//成员变量-目标对象

**private** Object target;

//有参构造（不用有参构造传入目标对象，用setter也行）

**public** CglibProxyFactory(Object target) {

**this**.target = target;

}

//获取代理对象

**public** Object getProxyInstance(){

//1.使用代理对象生成器(工厂思想)

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

//2.在增强器上设置两个属性

//设置目标对象

enhancer.setSuperclass(target.getClass());

//设置回调方法

enhancer.setCallback(**new** MethodInterceptor() {

//参数1：代理对象；参数2：目标对象的方法对象；参数3：目标对象的方法实参；参数4：代理对象的方法对象

@Override

**public** Object intercept(Object proxy, Method method, Object[] arg, MethodProxy methodProxy) **throws** Throwable {

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

System.***out***.println("记录日志...");

}

Object object = method.invoke(target, arg);

**return** object;

}

});

//3.返回代理对象

**return** enhancer.create();

}

}

* 第四步：SpringTest.java测试

**package** com.test.proxy.cglib;

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

@Test

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

//目标对象

ProductServiceImpl target = **new** ProductServiceImpl();

//代理工厂对象，注入目标

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

//织入，生成代理对象

ProductServiceImpl proxy = (ProductServiceImpl) cglibProxyFactory.getProxyInstance();

//调用代理对象的方法

proxy.save();ss

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

proxy.find();

}

}

【输出结果】

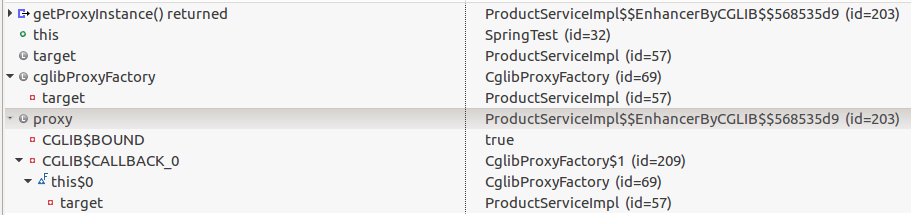
记录日志...

商品保存了...

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

商品查询了...

使用断点查看cglib生成的代理对象



CGLIB 生成代理类的命名规则：className$$classNameByCGLIB$$hashCode

【说明】

CGLIB生成的动态代理类是目标类的子类（JDK代理是兄弟关系，两个都是接口的子类；而这里是父子关系，没有借口也可以）

Class ProductServiceImpl$$EnhancerByCGLIB$$568535d9 extends ProductServiceImpl {

ProductServiceImpl productService= new ProductServiceImpl();

public void save() {

writeLog()

productService.save();

}

public int find() {

int returnValue = productService.find();

return returnValue;

}

private static void writeLog(){

System.out.println("记录日志...");

}

}

## **2.3代理小结**

【两者区别】

JDK代理：只能基于接口代理。会生成目标对象的接口类型的子对象，代理对象和目标对象兄弟关系。

CGLIB代理：可以基于类代理，不一定要有接口。基于类代理时会生成目标对象类型的子对象，代理对象和目标对象父子关系。

【Spring中代理的使用】

1.spring在运行期生成动态代理对象，不需要特殊的编译器

2.Spring底层两种代理方式的使用：

若目标对象实现了若干接口，使用JDK的java.lang.reflect.Proxy类进行代理

若目标对象没有实现任何接口，则使用CGLIB库生成目标对象的子类

【注意】

1.对接口代理优于对类代理。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增强，算是备用方案

2.被final修饰的方法不能够被通知/增强。spring是为目标类产生子类。任何需要被通知的方法都要被复写，将通知织入。而final方法不允许重写的。

3.spring只支持方法连接点（在方法前后增强），不提供属性接入点(不能进行方法内增强，侵入性低)。属性拦截（方法内增强）有悖于面向对象的思想：面向对象是对象处理工作，其他对象只能通过该对象调用其方法来得到结果

【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）

# **3.传统方式AOP编程**

Spring在1.2版本之后开始支持AOP编程。传统方式指的是不使用AspectJ来实现AOP。需要：

1. 确定目标
2. 编写通知(需要实现对应接口)
3. 配置ProxyFactoryBean生成代理对象，该步配置较复杂

由于配置ProxyFactoryBean的方式较复杂，且在开发中已很少使用。所以这里配置部分采用AspectJ语法来代替（用aop:config标签），只为了明确配置的原理

## **3.1传统方式的Advice**

Advice即通知或增强。狭义的指AOP联盟提供的org.aopalliance.aop.Advice接口（aopalliance-1.0.jar中）

为了遵循aop联盟的接口规范，Spring对该Advice接口，扩展了多个子接口，即提供了多样的Advice实现方式（5种）：

传统的Spring AOP编程中，按照Advice在目标类方法的连接点位置，划分成5种类型：前置通知、后置通知、环绕通知、异常通知、引介通知。每种通知的实现必须实现对应的接口：

1、前置通知org.springframework.aop.MethodBeforeAdvice（在目标对象方法执行前进行增强）

2、后置通知org.springframework.aop.AfterReturningAdvice（在目标对象方法执行后进行增强）

3、环绕通知org.aopalliance.intercept.MethodInterceptor（在目标对象方法执行前后都进行增强）

4、抛出通知org.springframework.aop.ThrowsAdvice（在目标对象方法执行抛出异常时进行增强）

5、引介通知org.springframework.aop.IntroductionInterceptor（在目标类中添加一些新的方法和属性）

【开发步骤】

面向切面编程的开发步骤，该步骤不仅适用于传统方式的AOP开发，对于aspectJ方式的AOP编码同样适用：

1、确定目标对象（target）（交由spring管理的bean）

2、编写Advice类 （增强代码）

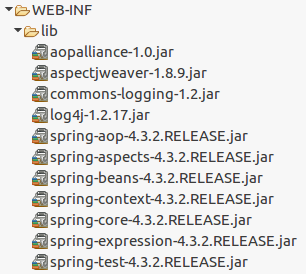
3、配置切入点和切面 （将Advice作用于target）

## **3.2传统AOP编程示例**

这里以CustomerServiceImpl（有接口）和ProductServiceImpl（无接口）作为 target，织入记录方法运行时间的环绕通知，并写入日志

创建包：com.test.aop.traditional

第一步：导入jar包（11个）



其中：

4个核心包+2个日志包+1个测试包

aopalliance-1.0.jar：aop联盟规范

spring-aop-4.3.2.RELEASE.jar：spring aop整合扩展包

aspectj.weaver-1.8.9.jar：aspectJ官方包

spring-aspects-4.3.2.RELEASE.jar：spring对aspectJ集成

第二步：导入log4j.properties打开文件记录，并将CustomerService.java、CustomerServiceImpl.java、ProductServiceImpl.java直接拷贝到该包下

### direct messages to stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.Target=System.out

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=[%-5p] %d{yyyy-MM-dd HH:mm:ss,SSS} [method]%l%n%m%n

### direct messages to file

log4j.appender.file=org.apache.log4j.FileAppender

log4j.appender.file.File=/home/yuzhou/mylog.log

log4j.appender.file.layout=org.apache.log4j.PatternLayout

log4j.appender.file.layout.ConversionPattern=[%-5p] %d{yyyy-MM-dd HH:mm:ss,SSS} [method]%l%n%m%n

### set log levels

log4j.rootLogger=info,stdout,file

第三步：编写Advice通知类（增强代码）

创建类TimeLogAdvice.java

传统aop编程方式，必须实现对应的接口。注意：这里实现的是aopalliance的MethodInterceptor接口，不是cglib包下的那个

**package** com.test.aop.traditional;

//自定义环绕Advice，实现对应的接口，记录运行时间

**public** **class** TimeLogAdvice **implements** MethodInterceptor {

**private** **static** Logger *logger* = Logger.*getLogger*(TimeLogAdvice.**class**);

//参数：目标方法回调函数的包装类，通过该类可获取调用方法的相关属性、方法名、调用该方法的对象等

@Override

**public** Object invoke(MethodInvocation methodInvocation) **throws** Throwable {

**long** begin = System.*currentTimeMillis*();

//目标对象原有方法执行

Object object = methodInvocation.proceed();

**long** end = System.*currentTimeMillis*();

//记录日志，配置文件需要打开file

**long** during = end - begin;

*logger*.info(methodInvocation.getThis().getClass().getName() + "的" + methodInvocation.getMethod().getName() + "方法运行了" + during + "ms");

//放行

**return** object;

}

}

【扩展】

关于aop联盟提供的MethodInvocation接口，API参考http://aopalliance.sourceforge.net/doc/org/aopalliance/intercept/MethodInvocation.html#method\_summary

第四步：在applicationContext-traditional.xml中配置target和advice的bean：CustomerServiceImpl和ProductService，以及通知类TimeLogAdvice

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

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

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

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

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

<!-- target:spring管理的bean实例 -->

<!-- 有接口的bean -->

<bean id=*"customerService"* class=*"com.test.aop.traditional.CustomerServiceImpl"*/>

<!-- 无接口的bean -->

<bean id=*"productService"* class=*"com.test.aop.traditional.ProductServiceImpl"*/>

<!-- 通知类 -->

<bean id=*"timeLogAdvice"* class=*"com.test.aop.traditional.TimeLogAdvice"*/>

</beans>

现在Advice对象和target对象之间还没有关系，目前只是把它们都交给spring管理

第五步：配置切面（切入点+通知）

目的：让哪个类（切面）的哪个方法（切入点），进行怎样的增强（通知），即让通知类对象作用于目标类对象的切入点方法

先引用aop的名称空间和xsd约束

<?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"*>

<!-- target:spring管理的bean实例 -->

<!-- 有接口的bean -->

<bean id=*"customerService"* class=*"com.test.aop.traditional.CustomerServiceImpl"*/>

<!-- 无接口的bean -->

<bean id=*"productService"* class=*"com.test.aop.traditional.ProductServiceImpl"*/>

<!-- 通知类 -->

<bean id=*"timeLogAdvice"* class=*"com.test.aop.traditional.TimeLogAdvice"*/>

<!-- AOP配置（切面） -->

<aop:config>

<!-- 配置Pointcut切入点：即要拦截的jointpoint（方法），expression：这里使用aspectj表达式语法，简单灵活 -->

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

<!-- 配置切面：对指定的pointcut，进行指定的Advice增强 -->

<aop:advisor advice-ref=*"timeLogAdvice"* pointcut-ref=*"myPointCut"*/>

</aop:config>

</beans>

第六步：SpringTest测试

**package** com.test.aop.traditional;

@RunWith(SpringJUnit4ClassRunner.**class**)

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

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

@Autowired

**private** CustomerService customerService;//有接口，注入的是代理对象

@Autowired

**private** ProductServiceImpl productService;//无接口，注入的是代理对象

@Test

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

//基于接口

customerService.save();

customerService.find();

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

//基于类的

productService.save();

productService.find();

}

}

执行结果：

Refreshing org.springframework.context.support.GenericApplicationContext@548a9f61: startup date [Wed May 16 15:06:47 CST 2018]; root of context hierarchy

客户保存了...

[INFO ] 2018-05-16 15:06:48,087 [method]com.test.aop.traditional.TimeLogAdvice.invoke(TimeLogAdvice.java:20)

com.test.aop.traditional.CustomerServiceImpl的save方法运行了0ms

客户查询了...

[INFO ] 2018-05-16 15:06:48,087 [method]com.test.aop.traditional.TimeLogAdvice.invoke(TimeLogAdvice.java:20)

com.test.aop.traditional.CustomerServiceImpl的find方法运行了0ms

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

商品保存了...

[INFO ] 2018-05-16 15:06:48,102 [method]com.test.aop.traditional.TimeLogAdvice.invoke(TimeLogAdvice.java:20)

com.test.aop.traditional.ProductServiceImpl的save方法运行了14ms

商品查询了...

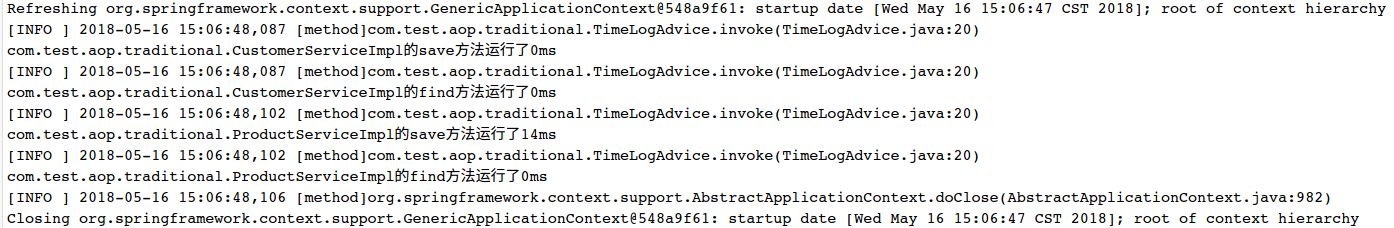
[INFO ] 2018-05-16 15:06:48,102 [method]com.test.aop.traditional.TimeLogAdvice.invoke(TimeLogAdvice.java:20)

com.test.aop.traditional.ProductServiceImpl的find方法运行了0ms

[INFO ] 2018-05-16 15:06:48,106 [method]org.springframework.context.support.AbstractApplicationContext.doClose(AbstractApplicationContext.java:982)

Closing org.springframework.context.support.GenericApplicationContext@548a9f61: startup date [Wed May 16 15:06:47 CST 2018]; root of context hierarchy

查看mylog.log文件

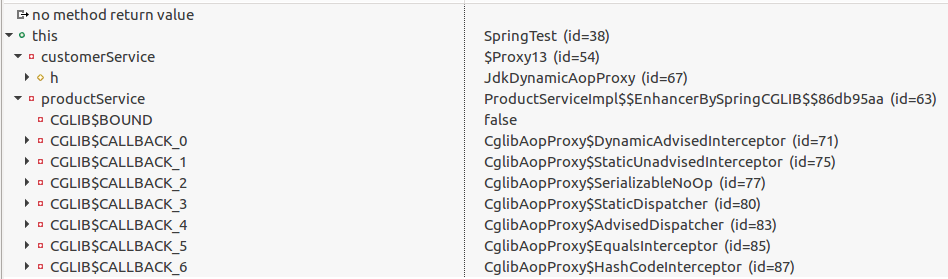


【补充】

运行时，注入的两个Service已经是代理对象了，使用Debug断点查看，其中

CustomerService有接口，使用JDK动态代理生成的代理对象$Proxy13

ProductService没有接口，使用cglib动态代理生成的代理对象ProductServiceImpl$$EnhancerBySpringCGLIB$$86db95aa



【扩展】

aspectJ表达式灵活简单，有多种表达形式，下面举例一二：

<aop:config>

<!-- 方式1:配置切面时，同时定义切入点 -->

<!-- <aop:advisor advice-ref="timeLogAdvice" pointcut="bean(\*Service\*)"/> -->

<!-- 方式2:切入点和切面分开配置，更灵活 -->

<!-- 配置Pointcut切入点：即要拦截的jointpoint（方法）,这里使用aspectj表达式语法,相对简单 -->

<!-- <aop:pointcut expression="bean(\*ServiceImpl)" id="myPointCut"/> -->

<!-- <aop:pointcut expression="bean(customerService)" id="myPointCut"/> -->

<!-- <aop:pointcut expression="bean(productService)" id="myPointCut"/> -->

<!-- <aop:pointcut expression="target(com.test.aop.traditional.ProductServiceImpl)" id="myPointCut"/> -->

<!-- <aop:pointcut expression="execution(\* com.test.aop.traditional.\*ServiceImpl.\*(..))" id="myPointCut"/> -->

<!-- <aop:pointcut expression="execution(\* com.test.aop.traditional.CustomerService.\*(..))" id="myPointCut"/> -->

<!-- <aop:pointcut expression="execution(\* com.test.aop.traditional.CustomerServiceImpl.\*(..))" id="myPointCut"/> -->

<!-- <aop:pointcut expression="execution(\* com.test.aop.traditional.CustomerServiceImpl.find(..))" id="myPointCut"/> -->

<!-- <aop:pointcut expression="execution(int com.test.aop.traditional.CustomerServiceImpl.\*(..))" id="myPointCut"/> -->

<!-- <aop:pointcut expression="execution(void com.test.aop.traditional.CustomerServiceImpl.\*(..))" id="myPointCut"/> -->

<aop:pointcut expression=*"within(com.test.aop.traditional..\*)"* id=*"myPointCut"*/>

<!-- 配置切面：对指定的pointcut，进行指定的Advice增强 -->

<aop:advisor advice-ref=*"timeLogAdvice"* pointcut-ref=*"myPointCut"*/>

</aop:config>

【参考】

AspectJ切入点表达式的语法整理

bean(bean Id/bean name)

bean(customerService) 增强spring容器中id/name=customerService的bean中所有方法

execution(<访问修饰符>?<返回类型>空格<方法名>(<参数>)<异常>?)

execution(void com.test.aop.traditional.UserServiceImpl.saveUser(java.lang.String,java.lang.String))表示：无返回类型，com.test.aop.traditional包中的UserServiceImpl类，中的saveUser方法，参数2个，都是String类型

execution(\* com.test.aop.traditional.CustomerServiceImpl.\*(..)) 增强该bean对象所有方法

execution(\* cn.test.aop..\*.\*(..)) 增强aop包和子包所有bean的所有方法

表达式的写法

execution（modifiers-pattern?（方法访问修饰符，非必填项） ret-type-pattern（返回类型，必填项） declaring-type-pattern? （非必填项） name-pattern（param-pattern）（必填项）--<方法名>(<参数>)

throws-pattern?（非必填项）<异常>?）

一共有5个参数，其中的?表示非必填项

：

除了返回类型模式（上面代码片断中的ret-type-pattern），名字模式和参数模式以外， 所有的部分都是可选的。

返回类型模式决定了方法的返回类型必须依次匹配一个连接点。 你会使用的最频繁的返回类型模式是\*，它代表了匹配任意的返回类型。

一个全限定的类型名将只会匹配返回给定类型的方法。名字模式匹配的是方法名。 你可以使用\*通配符作为所有或者部分命名模式。

参数模式稍微有点复杂：()匹配了一个不接受任何参数的方法， 而(..)匹配了一个接受任意数量参数的方法（零或者更多）。

模式(\*)匹配了一个接受一个任何类型的参数的方法。 模式(\*,String)匹配了一个接受两个参数的方法，第一个可以是任意类型， 第二个则必须是String类型。更多的信息请参阅AspectJ编程指南中 语言语义的部分。

1：modifiers-pattern? （非必填项）：表示方法的修饰符

execution(public void com.test.aop.traditional.UserServiceImpl.saveUser(java.lang.String,java.lang.String))

公有方法，无返回类型，com.test.aop.traditional包中的UserServiceImpl类，中的saveUser方法，参数2个，都是String类型

execution(private void com.test.aop.traditional.UserServiceImpl.saveUser(java.lang.String,java.lang.String))

私有方法，无返回类型，com.test.aop.traditional包中的UserServiceImpl类，中的saveUser方法，参数2个，都是String类型

2：ret-type-pattern （必填项）：表示方法的返回类型

execution(void com.test.aop.traditional.UserServiceImpl.saveUser(java.lang.String,java.lang.String))

无返回类型，com.test.aop.traditional包中的UserServiceImpl类，中的saveUser方法，参数2个，都是String类型

execution(java.lang.String com.test.aop.traditional.UserServiceImpl.saveUser(java.lang.String,java.lang.String))

返回类型String类型，com.test.aop.traditional包中的UserServiceImpl类，中的saveUser方法，参数2个，都是String类型

execution(\* com.test.aop.traditional.UserServiceImpl.saveUser(java.lang.String,java.lang.String))

返回类型任意，com.test.aop.traditional包中的UserServiceImpl类，中的saveUser方法，参数2个，都是String类型

3：declaring-type-pattern? （非必填项）：表示包，或者子包的，或者类的修饰符

execution(\* com.test.aop.traditional.UserServiceImpl.saveUser(java.lang.String,java.lang.String))

返回类型任意，com.test.aop.traditional包中的UserServiceImpl类，中的saveUser方法，参数2个，都是String类型

execution(\* com.test.aop.\*.UserServiceImpl.saveUser(java.lang.String,java.lang.String))

返回类型任意，com.test.aop包中的所有子包，中的UserServiceImpl类，中的saveUser方法，参数2个，都是String类型

execution(\* com.test.aop.\*.saveUser(java.lang.String,java.lang.String))

返回类型任意，com.test.aop包中的所有类，中的saveUser方法，参数2个，都是String类型

execution(\*com.test.aop..\*.saveUser(java.lang.String,java.lang.String))

返回类型任意，com.test.aop包中及其子包中的所有类，中的saveUser方法，参数2个，都是String类型

execution(\* \*.saveUser(java.lang.String,java.lang.String))

返回类型任意，所有包中的所有类，中的saveUser方法，参数2个，都是String类型

execution(\* saveUser(java.lang.String,java.lang.String))

返回类型任意，所有包中的所有类，中的saveUser方法，参数2个，都是String类型

4：name-pattern（param-pattern）（必填项）：方法的名称（方法的参数）

（1）方法名称

execution(\* com.test.aop.traditional.UserServiceImpl.save\*(java.lang.String,java.lang.String))

返回类型任意，com.test.aop.traditional包中的UserServiceImpl类，中的以save开头的方法，参数2个，都是String类型

execution(\* com.test.aop.traditional.UserServiceImpl.\*(java.lang.String,java.lang.String))

返回类型任意，com.test.aop.traditional包中的UserServiceImpl类，中的所有方法，参数2个，都是String类型

（2）方法的参数

execution(\* com.test.aop.traditional.UserServiceImpl.saveUser(java.lang.String,java.lang.Integer))

返回类型任意，com.test.aop.traditional包中的UserServiceImpl类，类中的saveUser方法，参数2个，参数1是String类型，参数二是Integer

execution(\* com.test.aop.traditional.UserServiceImpl.saveUser(java.lang.String,\*))

返回类型任意，com.test.aop.traditional包中的UserServiceImpl类，类中的saveUser方法，参数2个，参数1是String类型，参数二是任意类型

execution(\* com.test.aop.traditional.UserServiceImpl.saveUser(\*))

返回类型任意，com.test.aop.traditional包中的UserServiceImpl类，类中的saveUser方法，参数1个，参数是任意类型

execution(\* com.test.aop.traditional.UserServiceImpl.saveUser())

返回类型任意，com.test.aop.traditional包中的UserServiceImpl类，类中的saveUser方法，没有参数

execution(\* com.test.aop.traditional.UserServiceImpl.saveUser(..))

返回类型任意，com.test.aop.traditional包中的UserServiceImpl类，类中的saveUser方法，参数任意（可以是0个，也可以多个）

5：throws-pattern?（非必填项）：方法上抛出的异常

项目开发中表达式（最多用）

execution(\* com.test.aop.traditional..\*.\*(..))

返回类型任意，com.test.aop.traditional包及其子包中所有类，类中所有方法，参数任意

execution(\* \*..\*.\*(..))

返回类型任意，任意包中及其子包中所有类，类中所有方法，参数任意

execution(\* \*(..))

返回类型任意，任意包中及其子包中所有类，类中所有方法，参数任意

任意公共方法的执行：

execution（public \* \*（..））

任何一个名字以“set”开始的方法的执行：

execution（\* set\*（..））

AccountService接口定义的任意方法的执行：

execution（\* com.xyz.service.AccountService.\*（..））

在service包中定义的任意方法的执行：

execution（\* com.xyz.service.\*.\*（..））

在service包或其子包中定义的任意方法的执行：

execution（\* com.xyz.service..\*.\*（..））

within(包.类)

例如： within(com.test.aop.traditional..\*) 增强traditional包和子包所有bean“所有方法 ”

this(完整类型)/target(完整类型)

范围最小，只针对某个类型。

this对某一个类-（对代理对象有效），target对代理对象无效(只对目标对象有效)

例如： this(com.test.aop.traditional.CustomerServiceImpl) 增强类型所有方法（对代理对象有效）

target(com.test.aop.traditional.CustomerServiceImpl)增强类型所有方法（对目标对象有效）

注意：我们一般都对目标对象进行拦截，很少对代理对象进行拦截

【AspectJ表达式通配符】

\*：匹配任何数量字符（一个）；

..：匹配任何数量字符的重复（多个），如在类型模式中匹配任何数量子包；而在方法参数模式中匹配任何数量参数。

+：匹配指定类型的子类型；仅能作为后缀放在类型模式后边。

# **4.AspectJ方式AOP编程---xml**

早期传统的Spring1.2 AOP编程需要配置ProxyFactoryBean生成代理对象，配置复杂。spring2.0支持AspectJ语法，简化了AOP开发。

## **4.1 AspectJ方式的Advice**

AspectJ编程方式需要的Advice类型，是普通的pojo即可，不需要像传统方式要求实现对应接口

AspectJ提供的通知类型：

1.前置通知 Before，相当于BeforeAdvice

2.后置通知 AfterReturning，相当于AfterReturningAdvice

3.环绕通知 Around，相当于MethodInterceptor

4.抛出通知 AfterThrowing，相当于ThrowAdvice

5.最终通知 After，不管是否异常，该通知都会执行

6.引介通知 DeclareParents，相当于IntroductionInterceptor

相比传统的Spring AOP通知类型多了After最终通知。

【开发步骤】仍是三步：

1.确定目标对象（target-bean）

2.编写通知（advice）

3.配置切入点（pointcut）、切面（aspect）

【优点】

1. 和传统的aop配置相比，通知类型更多
2. 更灵活：advice不需要实现接口，简单的pojo即可；且一个通知类中可以写N个不同增强类型的方法，但传统的只能是一个类对应一个方法（重写的接口方法，如MethodInterceptor的invoke()）

3.一个通知可以增强多个连接点，一个连接点可以被多次增

## **4.2 AspectJ AOP编程示例**

示例：对传统方式中的2个target进行前置通知

创建包com.test.aop.aspectj

第一步：jar包和传统方式相同（传统方式讲解使用了AspectJ语法，所以也导入了AspectJ的相关包，故相同）

第二步：将CustomerService.java、CustomerServiceImpl.java、ProductServiceImpl.java直接拷贝到该包下

第三步：编写Advice通知类，无需实现接口

//aspectJ通知类

**public** **class** MyAspectjAdvice {

**public** **void** firstBefore() {

System.***out***.println("第1个前置通知...");

}

}

第四步：创建applicationContext-aspectj.xml，配置目标对象和通知类对象bean，并配置切面（两者的作用关系）

<?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"*>

<!-- target:spring管理的bean实例 -->

<!-- 有接口的bean -->

<bean id=*"customerService"* class=*"com.test.aop.aspectj.CustomerServiceImpl"*/>

<!-- 无接口的bean -->

<bean id=*"productService"* class=*"com.test.aop.aspectj.ProductServiceImpl"*/>

<!-- 通知类 -->

<bean id=*"myAspectjAdvice"* class=*"com.test.aop.aspectj.MyAspectjAdvice"*/>

<!-- 传统形式使用aop:advisor -->

<!-- <aop:config>

<aop:pointcut expression="within(com.test.aop.traditional..\*)" id="myPointCut"/>

<aop:advisor advice-ref="timeLogAdvice" pointcut-ref="myPointCut"/>

</aop:config> -->

<aop:config>

<!-- 方式1 -->

<!-- <aop:aspect ref="myAspectjAdvice">

<aop:before method="firstBefore" pointcut="within(com.test.aop.aspectj..\*)"/>

</aop:aspect> -->

<!-- 方式2 -->

<aop:pointcut expression=*"within(com.test.aop.aspectj..\*)"* id=*"myPointCut"*/>

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

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

</aop:aspect>

</aop:config>

</beans>

第五步：SpringTest测试

**package** com.test.aop.aspectj;

@RunWith(SpringJUnit4ClassRunner.**class**)

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

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

@Autowired

**private** CustomerService customerService;//有接口，注入的是代理对象

@Autowired

**private** ProductServiceImpl productService;//无接口，注入的是代理对象

@Test

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

//基于接口

customerService.save();

customerService.find();

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

//基于类的

productService.save();

productService.find();

}

}

测试结果：

第1个前置通知...

客户保存了...

第1个前置通知...

客户查询了...

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

第1个前置通知...

商品保存了...

第1个前置通知...

商品查询了...

【扩展优化】

配置多个Advice

<aop:config>

<aop:pointcut expression=*"within(com.test.aop.aspectj..\*)"* id=*"myPointCut"*/>

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

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

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

</aop:aspect>

</aop:config>

执行结果（有顺序的）

第2个前置通知...

第1个前置通知...

客户保存了...

第2个前置通知...

第1个前置通知...

客户查询了...

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

第2个前置通知...

第1个前置通知...

商品保存了...

第2个前置通知...

第1个前置通知...

商品查询了...

## **4.3 AspectJ的其他通知**

AspectJ方式的面向切面编程，相比于传统的SpringAOP方式，定义的通知类型更多（6种），常用的如下

【补充】

连接点对象

AspectJ使用org.aspectj.lang.JoinPoint接口表示目标类的连接点对象；如果是环绕通知，使用org.aspectj.lang.ProceedingJoinPoint表示连接点对象，它是JoinPoint的子接口。

任何一个增强方法都可以通过将第一个入参声明为JoinPoint，来访问连接点的信息。这两个接口的主要方法：

* JoinPoint

Object[] getArgs()：获取连接点方法运行时的入参列表；

Signature getSignature() ：获取连接点方法签名对象；

Object getTarget() ：获取连接点所在的目标对象；

Object getThis() ：获取代理对象本身；

* ProceedingJoinPoint

ProceedingJoinPoint继承JoinPoint子接口，它新增了两个用于执行连接点方法的方法：

Object proceed() throws java.lang.Throwable：通过反射，执行目标对象的连接点方法；

Object proceed(Object[] args) throws Throwable：通过反射，执行目标对象的连接点方法，不过使用新的入参替换原来的入参

更多API，参考https://www.eclipse.org/aspectj/doc/released/runtime-api/org/aspectj/lang/JoinPoint.html

注意是AspectJ包中的JointPoint，不是其他包中的

### **4.3.1 Before前置通知**

应用： 实现权限控制、日志记录等

第一步：在MyAspectjAdvice.java中增加前置通知的方法（方法名任意，需配置到xml）

**package** com.test.aop.aspectj;

**import** org.aspectj.lang.JoinPoint;

//aspectJ通知类

**public** **class** MyAspectjAdvice {

/\*\*

\* 1.（会被配置为）前置通知

\* **@param** joinPoint 连接点对象

\*/

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

System.***out***.println("前置通知执行了...");

//JoinPoint接口的API，获取连接点的相关信息

System.***out***.println("连接点方法:" + joinPoint.getSignature());

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

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

//权限拦截

String name = "rose";

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

**throw** **new** RuntimeException("无权访问" + joinPoint.getTarget().getClass().getName() + "中的" + joinPoint.getSignature());

}

}

}

第二步：applicationContext-aspectj.xml中添加如下AOP配置

<!-- before通知 -->

<aop:config>

<!-- <aop:pointcut expression="execution(\* com.test.aop.aspectj.\*.\*(..))" id="myPointCut"/> -->

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

<!-- <aop:before method="before" pointcut-ref="myPointCut"/> -->

<aop:before method=*"before"* pointcut=*"execution(\* com.test.aop.aspectj.\*.\*(..))"*/>

</aop:aspect>

</aop:config>

第三步：使用SpringTest.java测试

package com.test.aop.aspectj;

@RunWith(SpringJUnit4ClassRunner.class)

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

public class SpringTest {

@Autowired

private CustomerService customerService;//有接口的target实例

@Autowired

private ProductServiceImpl productService;//无接口的target实例

@Test

public void test(){

//基于接口

customerService.save();

customerService.find();

System.out.println("-------------");

//基于类的

productService.save();

productService.find();

}

}

测试结果：前置通知执行了，并在权限不满足时跑出了异常

Refreshing org.springframework.context.support.GenericApplicationContext@548a9f61: startup date [Wed May 16 15:27:20 CST 2018]; root of context hierarchy

前置通知执行了...

连接点方法:void com.test.aop.aspectj.CustomerService.save()

目标对象:com.test.aop.aspectj.CustomerServiceImpl

代理对象:com.sun.proxy.$Proxy13

[INFO ] 2018-05-16 15:27:20,312 [method]org.springframework.context.support.AbstractApplicationContext.doClose(AbstractApplicationContext.java:982)

Closing org.springframework.context.support.GenericApplicationContext@548a9f61: startup date [Wed May 16 15:27:20 CST 2018]; root of context hierarchy



### **4.3.2 AfterReturing后置通知**

后置通知可以获取到目标方法JointPoint的返回值，如果想对返回值进行操作，就使用后置通知

应用：短信功能等

第一步：在MyAspectjAdvice.java中增加后置通知的方法（方法名任意，需配置到xml）

/\*\*

\* 2.（会被配置为）后置通知

\* **@param** joinPoint 连接点对象

\* **@param** returnVal 目标方法执行后的返回值,类型是object，参数名任意，会配置到xml

\*/

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

System.***out***.println("后置通知执行了...");

System.***out***.println("尊敬的用户，您的余额为：" + returnVal);

}

第二步：applicationContext-aspectj.xml中添加如下AOP配置

<!-- afterReturing通知 -->

<aop:config>

<aop:pointcut expression=*"execution(\* com.test.aop.aspectj.\*.\*(..))"* id=*"myPointCut"*/>

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

<!-- returning指定目标对象方法（jointpoint）返回值的接收参数，spring会自动将jointpoint的返回值，传入该afterReturing方法的该参数中 -->

<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，代码不变。测试结果：

客户保存了...

后置通知执行了...

尊敬的用户，您的余额为：null

客户查询了...

后置通知执行了...

尊敬的用户，您的余额为：100

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

商品保存了...

后置通知执行了...

尊敬的用户，您的余额为：null

商品查询了...

后置通知执行了...

尊敬的用户，您的余额为：22

### **4.3.3 Around环绕通知**

应用：日志、缓存、权限、性能监控、事务管理

【环绕增强方法的格式要求】

参数：ProceedingJoinPoint（执行中的连接点，是JointPoint的子接口）

返回值类型：Object

抛出：Throwable（顶级接口）

第一步：配置MyAspect类（切面），配置around方法（通知）

/\*\*

\* 3.（会被配置为）环绕通知

\* @param proceedingJoinPoint (执行中的)连接点对象

\* @return 返回目标方法的返回值

\* @throws Throwable 必须抛出Throwable

\*/

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

System.***out***.println("环绕通知执行了...");

System.***out***.println("开启事务...");

//调用proceed()方法可以随时随地执行目标对象的方法

Object returnVal = proceedingJoinPoint.proceed();

System.***out***.println("提交事物...");

**return** returnVal;

}

第二步：applicationContext-aspectj.xml中添加如下AOP配置

<!-- around通知 -->

<aop:config>

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

<aop:around method=*"around"* pointcut=*"execution(\* com.test.aop.aspectj.\*.find(..))"*/>

</aop:aspect>

</aop:config>

第三步：使用SpringTest.java进行测试，代码不变，测试结果：

客户保存了...

环绕通知执行了...

开启事务...

客户查询了...

提交事物...

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

商品保存了...

环绕通知执行了...

开启事务...

商品查询了...

提交事物...

### **4.3.4 AfterThrowing抛出通知**

应用：处理异常（一般是不可预知的异常）记录日志、通知管理员操作（短信、邮件）

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

/\*\*

\* 4.（会被配置为）抛出通知

\* **@param** joinPoint 连接点对象

\* **@param** ex 目标方法抛出的异常，类型是Throwable，参数名任意，会配置到xml

\*/

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

System.***out***.println("抛出通知执行了...");

System.***out***.println("管理员" + joinPoint.getTarget().getClass().getName() + "的方法：" + joinPoint.getSignature().getName() + "发生了异常，异常为：" + ex.getMessage());

}

第二步：applicationContext-aspectj.xml中添加如下AOP配置

<!-- afterThrowing通知 -->

<aop:config>

<aop:pointcut expression=*"execution(\* com.test.aop.aspectj.\*.\*(..))"* id=*"myPointCut"*/>

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

<!-- throwing指定目标对象方法（jointpoint）抛出异常的接收参数，spring会自动将该值传入该afterThrowing方法的ex参数中 -->

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

</aop:aspect>

</aop:config>

第三步：在ProductServiceImpl中制造异常

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

**int** i = 1/ 0;

System.***out***.println("商品保存了...");

}

第四步：使用SpringTest.java进行测试，代码不变，测试结果：

客户保存了...

客户查询了...

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

抛出通知执行了...

管理员com.test.aop.aspectj.ProductServiceImpl的方法：save发生了异常，异常为：/ by zero

### **4.3.5 After最终通知**

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：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）

第一步：配置MyAspect类（切面），配置after方法（通知）

/\*\*

\* 5.（会被配置为）最终通知

\* **@param** joinPoint 连接点对象

\*/

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

System.***out***.println("最终通知执行了...");

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

}

第二步：applicationContext-aspectj.xml中添加如下AOP配置

<!-- after通知 -->

<aop:config>

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

<aop:after method=*"after"* pointcut=*"execution(\* com.test.aop.aspectj.\*.\*(..))"*/>

</aop:aspect>

</aop:config>

第三步：保持ProductServiceImpl中的异常，和抛出通知的配置。使用SpringTest.java进行测试，结果：

即使抛出异常ProductServiceImpl.save()的最终通知还是执行了，随后才抛出异常，后面的程序就终止了

客户保存了...

最终通知执行了...

数据库连接被释放了。执行的方法是：save

客户查询了...

最终通知执行了...

数据库连接被释放了。执行的方法是：find

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

最终通知执行了...

数据库连接被释放了。执行的方法是：save

抛出通知执行了...

管理员com.test.aop.aspectj.ProductServiceImpl的方法：save发生了异常，异常为：/ by zero

### **4.3.6五种常用通知小结**

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，因为可在环绕通知的方法中编写如下：

try {

//前置通知

Object result = proceedingJoinPoint.proceed();

//后置通知

}catch(Exception){

//抛出通知

}finally{

//最终通知

}

2.各种Advice方法格式（参考）

方法名：任意（需配置到xml中）

|  |  |  |  |
| --- | --- | --- | --- |
| 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 |  |

# **5.AspectJ方式AOP编程---注解**

搭建环境：

jar包：同xml方式一样（11个）

创建包：com.test.aop.aspectj.anno

## **5.1准备工作**

包括：目标类（有接口、无接口），并注解到spring容器、开启包扫描、测试类

CustomerService.java接口

**package** com.test.aop.aspectj.anno;

**public** **interface** CustomerService {

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

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

}

CustomerServiceImpl.java

**package** com.test.aop.aspectj.anno;

@Service("customerService")

**public** **class** CustomerServiceImpl **implements** CustomerService{

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

System.***out***.println("客户保存了...");

}

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

System.***out***.println("客户查询了...");

**return** 100;

}

}

ProductServiceImpl.java

**package** com.test.aop.aspectj.anno;

@Service("productService")

**public** **class** ProductServiceImpl {

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

System.***out***.println("商品保存了...");

}

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

System.***out***.println("商品查询了...");

**return** 99;

}

}

applicationContext-anno.xml：包含beans、context、aop名称空间，开启注解包扫描

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

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

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

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/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"*>

<!-- 开启注解功能 -->

<!-- <context:annotation-config/> -->

<!-- 包扫描 -->

<context:component-scan base-package=*"com.test.aop.aspectj.anno"*/>

</beans>

测试类SpringTest.java

package com.test.aop.aspectj.anno;

@RunWith(SpringJUnit4ClassRunner.class)

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

public class SpringTest {

@Autowired

private CustomerService customerService;//有接口的target实例

@Autowired

private ProductServiceImpl productService;//无接口的target实例

@Test

public void test(){

//基于接口

customerService.save();

customerService.find();

System.out.println("-------------");

//基于类的

productService.save();

productService.find();

}

}

## **5.2编写通知和配置切面**

【回顾】

Xml配置的两种方式，以前置通知为例，先配置好通知类，再配置切面：

1. 单独配置pointcut，更灵活，可对该切入点进行多个通知增强
2. 配置切面时，同时配置pointcut（匿名），简单，但不灵活

<bean id=*"myAspectjAdvice"* class=*"com.test.aop.aspectj.MyAspectjAdvice"*/>

<!-- before通知 -->

<aop:config>

<!-- <aop:pointcut expression="execution(\* com.test.aop.aspectj.\*.\*(..))" id="myPointCut"/> -->

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

<!-- <aop:before method="before" pointcut-ref="myPointCut"/> -->

<aop:before method=*"before"* pointcut=*"execution(\* com.test.aop.aspectj.\*.\*(..))"*/>

</aop:aspect>

</aop:config>

【注解通知和切面配置的步骤】

* 1.开启aspectj自动代理<aop:aspectj-autoproxy/>
* 2.编写通知类，添加@Component + @Aspect，表示这是一个通知类+切面类（该类中即包含通知方法，又包含切面=切入点和通知关联）

@Component：将通知类交给spring管理，相当于<bean id=*"myAspectjAdvice"* class=*"com.test.aop.aspectj.MyAspectjAdvice"*/>

@Aspect：将该类同时标记为切面类，相当于<aop:aspect ref=*"myAspectjAdvice"*>

* 3.在通知类中编写通知方法，在方法上添加相应的注解，并配置切面

1.前置通知 @Before，相当于<aop:before>和BeforeAdvice

2.后置通知 @AfterReturning，相当于<aop:after-returning>和AfterReturningAdvice

3.环绕通知 @Around，相当于<aop:around>和MethodInterceptor

4.抛出通知 @AfterThrowing，相当于<aop:after-throwing>和ThrowAdvice

5.最终通知 @After，相当于<aop:after>

6.引介通知 @DeclareParents，相当于<aop:declare-parents>和IntroductionInterceptor

【<aop:aspectj-autoproxy />】

1.该配置会自动，为spring容器中被@aspectJ注解的bean（切面类）创建自动代理。即对该类中指定的切入点进行指定的增强（为目标对象织入切面），相当于<aop:config>。

当然，spring 在内部依旧采用AnnotationAwareAspectJAutoProxyCreator进行自动代理的创建工作，只不过具体实现的细节已经被<aop:aspectj-autoproxy />隐藏起来了   
2.<aop:aspectj-autoproxy />的proxy-target-class属性，默认为false，表示使用jdk动态代理织入增强，生成代理对象。如果目标类没有声明接口，则自动使用CGLib动态代理；当poxy-target-class=true时，表示不管目标类有没有实现接口，都使用CGLib动态代理织入增强

【注解方式的切入点定义】

和xml方式一样，注解方式定义切入点，也有2种方式，以前置通知为例：

* 方式1：单独配置pointcut，即使用@Pointcut

//相当于：<aop:pointcut expression="execution(\* com.test.aop.aspectj.anno.\*.\*(..))" id="myPointCut1"/>

@Pointcut("execution(\* com.test.aop.aspectj.anno.\*.\*(..))")

**private** **void** myPointCut1() {}

@Pointcut("execution(\* com.test.aop.aspectj.anno.ProductServiceImpl.\*(..))")

**private** **void** myPointCut2() {}

//@Before("execution(\* com.test.aop.aspectj.anno.\*.\*(..))")

@Before("myPointCut1() || myPointCut2()")

**public** **void** before() {

System.***out***.println("前置通知执行了...");

}

【该方式要求】

1.切入点方法格式：private void 无参数、无方法体，方法名为切入点的id

2.一个通知方法，如@Before上，可以使用多个切入点，中间使用“||”符合分隔，用来表示多个切入点。运算的结果是“补集”（即有重复覆盖的切入点，也只执行一次增强。上例中ProductServiceImpl的切入点只增强一次）

* 方式2：配置切面时，同时配置pointcut（简单，推荐）

//相当于：<aop:before method="before" pointcut="execution(\* com.test.aop.aspectj.anno.\*.\*(..))"/>

@Before("execution(\* com.test.aop.aspectj.anno.\*.\*(..))")

**public** **void** before() {

System.***out***.println("前置通知执行了...");

}

### **5.2.1前置通知**

package com.test.aop.aspectj.anno;

//通知类+切面类

@Component("myAspectj")

@Aspect

public class MyAspectj {

//自定义切入点

/\*@Pointcut("execution(\* com.test.aop.aspectj.anno.\*.\*(..))")

private void myPointCut1() {}

@Pointcut("execution(\* com.test.aop.aspectj.anno.ProductServiceImpl.\*(..))")

private void myPointCut2() {}\*/

/\*\*

\* 1.（会被自动配置为）前置通知

\* @param joinPoint 连接点对象

\*/

//@Before("myPointCut1() || myPointCut2()")

@Before("execution(\* com.test.aop.aspectj.anno.\*.\*(..))")

public void before(JoinPoint joinPoint) {

System.out.println("前置通知执行了...");

//JoinPoint接口的API，获取连接点的相关信息

System.out.println("连接点方法:" + joinPoint.getSignature());

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

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

//权限拦截

String name = "rose";

if (!"admin".equals(name)) {

throw new RuntimeException("无权访问" + joinPoint.getTarget().getClass().getName() + "中的" + joinPoint.getSignature());

}

}

}

使用SpringTest进行测试：

前置通知执行了...

连接点方法:void com.test.aop.aspectj.anno.CustomerService.save()

目标对象:com.test.aop.aspectj.anno.CustomerServiceImpl

代理对象:com.sun.proxy.$Proxy16

并抛出异常：

java.lang.RuntimeException: 无权访问com.test.aop.aspectj.anno.CustomerServiceImpl中的void com.test.aop.aspectj.anno.CustomerService.save()

### **5.2.2后置通知**

在切面的类MyAspect.java类中添加通知方法

//自定义切入点

@Pointcut("execution(\* com.test.aop.aspectj.anno.ProductServiceImpl.\*(..))")

**private** **void** myPointCut2() {}

/\*\*

\* 2.（会被自动配置为）后置通知

\* **@param** joinPoint 连接点对象

\* **@param** returnVal 目标方法执行后的返回值,类型是object，参数名任意

\*/

//@AfterReturning(value="execution(\* com.test.aop.aspectj.anno.ProductServiceImpl.\*(..))", returning="returnVal")

@AfterReturning(value="myPointCut2()", returning="returnVal")

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

System.***out***.println("后置通知执行了...");

System.***out***.println("尊敬的用户，您的余额为：" + returnVal);

}

关闭前置通知后（因为会抛异常），使用SpringTest测试

客户保存了...

客户查询了...

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

商品保存了...

后置通知执行了...

尊敬的用户，您的余额为：null

商品查询了...

后置通知执行了...

尊敬的用户，您的余额为：99

### **5.2.3环绕通知**

//自定义切入点

@Pointcut("execution(\* com.test.aop.aspectj.anno.ProductServiceImpl.\*(..))")

**private** **void** myPointCut2() {}

/\*\*

\* 3.（会被自动配置为）环绕通知

\* **@param** proceedingJoinPoint (执行中的)连接点对象

\* **@return** 返回目标方法的返回值

\* **@throws** Throwable 必须抛出Throwable

\*/

//@Around("execution(\* com.test.aop.aspectj.anno.ProductServiceImpl.\*(..))")

@Around("myPointCut2()")

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

System.***out***.println("环绕通知执行了...");

System.***out***.println("开启事务...");

//调用proceed()方法可以随时随地执行目标对象的方法

Object returnVal = proceedingJoinPoint.proceed();

System.***out***.println("提交事物...");

**return** returnVal;

}

保持前置和后置通知配置有效，并让前置通知权限通过，此时测试结果：

客户保存了...

客户查询了...

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

环绕通知执行了...

开启事务...

前置通知执行了...

连接点方法:void com.test.aop.aspectj.anno.ProductServiceImpl.save()

目标对象:com.test.aop.aspectj.anno.ProductServiceImpl

代理对象:com.test.aop.aspectj.anno.ProductServiceImpl$$EnhancerBySpringCGLIB$$f7b808b

商品保存了...

提交事物...

后置通知执行了...

尊敬的用户，您的余额为：null

环绕通知执行了...

开启事务...

前置通知执行了...

连接点方法:int com.test.aop.aspectj.anno.ProductServiceImpl.find()

目标对象:com.test.aop.aspectj.anno.ProductServiceImpl

代理对象:com.test.aop.aspectj.anno.ProductServiceImpl$$EnhancerBySpringCGLIB$$f7b808b

商品查询了...

提交事物...

后置通知执行了...

尊敬的用户，您的余额为：99

### **5.2.4抛出通知**

在ProductServiceImpl中制造异常

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

**int** i = 1/ 0;

System.***out***.println("商品保存了...");

}

/\*\*

\* 4.（会被自动配置为）抛出通知

\* **@param** joinPoint 连接点对象

\* **@param** ex 目标方法抛出的异常，类型是Throwable，参数名任意

\*/

//@AfterThrowing(pointcut="execution(\* com.test.aop.aspectj.anno.ProductServiceImpl.\*(..))", throwing="ex")

@AfterThrowing(pointcut="myPointCut2()", throwing="ex")

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

System.***out***.println("抛出通知执行了...");

System.***out***.println("管理员" + joinPoint.getTarget().getClass().getName() + "的方法：" + joinPoint.getSignature().getName() + "发生了异常，异常为：" + ex.getMessage());

}

保持前置、后置和环绕通知配置有效，并让前置通知权限通过，此时测试结果：

客户保存了...

客户查询了...

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

环绕通知执行了...

开启事务...

前置通知执行了...

连接点方法:void com.test.aop.aspectj.anno.ProductServiceImpl.save()

目标对象:com.test.aop.aspectj.anno.ProductServiceImpl

代理对象:com.test.aop.aspectj.anno.ProductServiceImpl$$EnhancerBySpringCGLIB$$d73d3144

抛出通知执行了...

管理员com.test.aop.aspectj.anno.ProductServiceImpl的方法：save发生了异常，异常为：/ by zero

### **5.2.5最终通知**

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）

/\*\*

\* 5.（会被配置为）最终通知

\* **@param** joinPoint 连接点对象

\*/

@After(value="execution(\* com.test.aop.aspectj.anno.ProductServiceImpl.\*(..))")

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

System.***out***.println("最终通知执行了...");

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

}

保持前置、后置、环绕和抛出通知配置有效，保持异常，并让前置通知权限通过，此时测试结果：

客户保存了...

客户查询了...

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

环绕通知执行了...

开启事务...

前置通知执行了...

连接点方法:void com.test.aop.aspectj.anno.ProductServiceImpl.save()

目标对象:com.test.aop.aspectj.anno.ProductServiceImpl

代理对象:com.test.aop.aspectj.anno.ProductServiceImpl$$EnhancerBySpringCGLIB$$e894bb73

最终通知执行了...

数据库连接被释放了。执行的方法是：save

抛出通知执行了...

管理员com.test.aop.aspectj.anno.ProductServiceImpl的方法：save发生了异常，异常为：/ by zero

## **5.3代理切换**

Spring aop基于代理机制，有接口时默认对接口代理（即jdk动态代理），没有接口时默认对类代理（即cglib动态代理）

【问题】如果目标对象有接口，能否也只对实现类代理，而不对接口进行代理呢？（即都使用cglib）

【示例】

第一步：在CustomerServiceImpl中新增code()（接口中没有定义，即实现类特有的方法）

@Service("customerService")

**public** **class** CustomerServiceImpl **implements** CustomerService{

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

System.***out***.println("客户保存了...");

}

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

System.***out***.println("客户查询了...");

**return** 100;

}

//实现类特有的方法，接口中没有定义

**public** **void** code() {

System.***out***.println("客户敲代码了...");

}

}

第二步：在测试类中调用子类的扩展方法：

@RunWith(SpringJUnit4ClassRunner.**class**)

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

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

@Autowired

**private** CustomerService customerService;//有接口的target实例

@Test

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

//customerService是CustomerService类型的引用，多态，要调用实现类特有的方法，就要向下强转

//但customerService引用指向的，是JDK动态代理对象，该对象是接口的子类型的对象，和CustomerServiceImpl是兄弟关系

((CustomerServiceImpl) customerService).code();

}

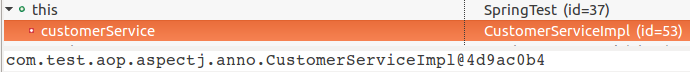
}

此时不对CustomerServiceImpl进行任何增强，测试结果正常：

客户敲代码了...

此时customerService是CustomerService类型的引用，指向的是CustomerServiceImpl实例，多态；要调用实现类特有的方法，需要向下强转，即：

CustomerServiceImpl customerService = com.test.aop.aspectj.anno.CustomerServiceImpl@4d9ac0b4



第三步：对CustomerServiceImpl进行前置通知

@Before("execution(\* com.test.aop.aspectj.anno.CustomerServiceImpl.\*(..))")

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

System.***out***.println("前置通知执行了...");

}

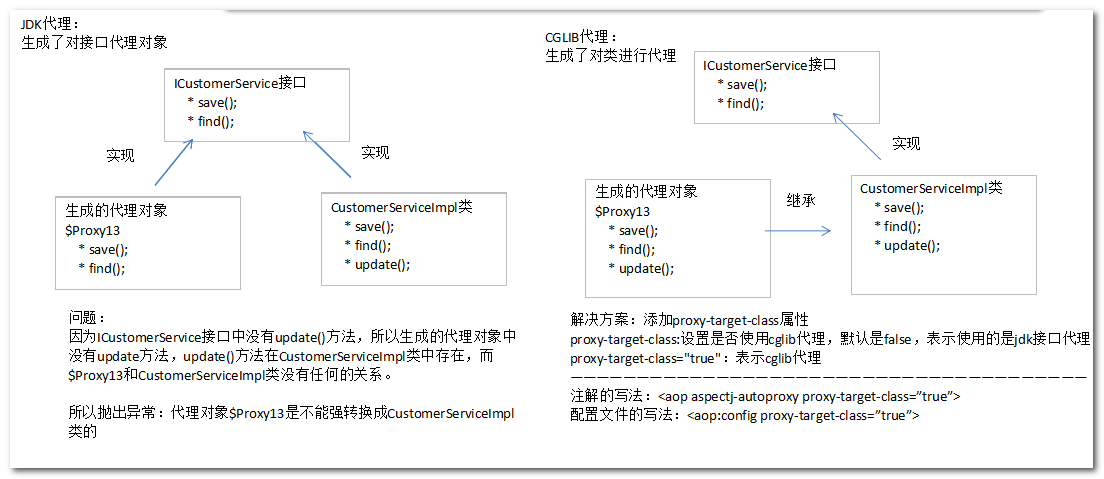
测试发现异常：

java.lang.ClassCastException: com.sun.proxy.$Proxy17 cannot be cast to com.test.aop.aspectj.anno.CustomerServiceImpl

【原因】

这里使用的是JDK代理，代理是接口，代理类实例无法转换成实现类。

即CustomerServiceImpl customerService = com.sun.proxy.$Proxy17是不可以的，不能把JDK代理类实例赋值给目标类引用（只是兄弟关系。既不是同一类型，也不是父子类型，不能赋值）



【解决方案】

此时即使有接口，也设置成cglib代理，即直接对实现类CustomerServiceImpl进行代理，不再对接口进行代理。

代理类和实现类是父子关系：接口----实现类----代理类，这样就可以CustomerServiceImpl customerService = com.sun.proxy.$Proxy17了

实现：设置 proxy-target-class = true

* 方式一：注解方式：

<!--

配置aspectj自动代理：自动扫描bean组件，将含有@Aspect的bean，作为aop管理，相当于:<aop:config>

proxy-target-class:默认是false，表示使用jdk接口代理；true表示对类代理，即使有接口，也使用cglib代理

-->

<aop:aspectj-autoproxy proxy-target-class=*"true"*/>

* 方案二：XML方式

<aop:config proxy-target-class=*"true"*>

...

</aop:config>

测试结果：

前置通知执行了...

连接点方法:void com.test.aop.aspectj.anno.CustomerServiceImpl.code()

目标对象:com.test.aop.aspectj.anno.CustomerServiceImpl

代理对象:com.test.aop.aspectj.anno.CustomerServiceImpl$$EnhancerBySpringCGLIB$$d3047c9a

客户敲代码了...

# **6.Spring JdbcTemplate**

Spring JdbcTemplate是一个模板工具类，为了简化Jdbc编程（类似Apache DbUtils ）。为了方便Dao编程，Spring为很多持久化技术都提供了对应的模板类：

|  |  |
| --- | --- |
| ORM持久化技术 | 模板类 |
| JDBC | org.springframework.jdbc.core.JdbcTemplate |
| Hibernate3.0 | org.springframework.orm.hibernate3.HibernateTemplate |
| IBatis(MyBatis) | org.springframework.orm.ibatis.SqlMapClientTemplate |
| JPA | org.springframework.orm.jpa.JpaTemplate |

【使用方法】

1. 继承支持类
2. 注入数据源，获取模板类即可

为了方便Dao中使用这些模板类，Spring又提供了对应的支持类，可以继承方便使用：

|  |  |
| --- | --- |
| ORM持久化技术 | 模板类 |
| JDBC | org.springframework.jdbc.core.support.JdbcDaoSupport |
| Hibernate3.0 | org.springframework.orm.hibernate3.support.HibernateDaoSupport |
| IBatis(MyBatis) | org.springframework.orm.ibatis.support.SqlMapClientDaoSupport |

以JdbcDaoSupport为例分析源码，不难发现，需要先注入数据源。注入dataSource后，就能得到和该数据源对应的jdbcTemplate实例：（该类也有setJdbcTemplate()，直接注入jdbcTemplate也可以）

**public** **abstract** **class** JdbcDaoSupport **extends** DaoSupport {

**private** JdbcTemplate jdbcTemplate;

/\*\*

\* Set the JDBC DataSource to be used by this DAO.

\*/

**public** **final** **void** setDataSource(DataSource dataSource) {

**if** (**this**.jdbcTemplate == **null** || dataSource != **this**.jdbcTemplate.getDataSource()) {

**this**.jdbcTemplate = createJdbcTemplate(dataSource);

initTemplateConfig();

}

}

/\*\*

\* Create a JdbcTemplate for the given DataSource.

\* Only invoked if populating the DAO with a DataSource reference!

\* <p>Can be overridden in subclasses to provide a JdbcTemplate instance

\* with different configuration, or a custom JdbcTemplate subclass.

\* **@param** dataSource the JDBC DataSource to create a JdbcTemplate for

\* **@return** the new JdbcTemplate instance

\* **@see** #setDataSource

\*/

**protected** JdbcTemplate createJdbcTemplate(DataSource dataSource) {

**return** **new** JdbcTemplate(dataSource);

}

...

这里主要使用2个模板：

JdbcTemplate 简化 jdbc编程

HibernateTemplate 极大的简化 Hibernate编程

## **6.1 JdbcTemplate 快速入门**

用JDBCTemplate编程，基本步骤如下：

1）构建连接池DataSource

2）构建JDBCTemplate

3）调用JDBCTemplate的execute方法

第一步：新建web项目study\_spring03

导入jar包和引入配置文件

Jar：共10个 = 4个spring核心包+2个日志包+测试包+事务包+jdbc包+MySQL驱动包。其中JDBC模版开发主要涉及spring-jdbc和spring-tx

配置文件：log4j.properties和applicationContiext.xml

第二步：数据库准备：创建库

CREATE DATABASE `test\_spring1` DEFAULT CHARACTER SET utf8mb4;

第三步：创建JdbcTemplateTest.java进行测试

**package** com.test.jdbctemplate;

**public** **class** JdbcTemplateTest {

@Test

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

//1.创建数据源

DriverManagerDataSource dataSource = **new** DriverManagerDataSource();

dataSource.setDriverClassName("com.mysql.jdbc.Driver");

dataSource.setUrl("jdbc:mysql://localhost:3306/test\_spring");

dataSource.setUsername("root");

dataSource.setPassword("root");

//2.创建jdbcTemplate模板类

JdbcTemplate jdbcTemplate = **new** JdbcTemplate(dataSource); //jdbcTemplate.setDataSource(dataSource)

//3.执行语句

jdbcTemplate.execute("create table test1(id int,name varchar(20),price decimal(10,2))");

}

}

结果：

成功创建表test1

## **6.2将JdbcTemplate交由spring工厂**

JdbcTemplate需要数据源（连接池），这里使用几种数据源/进行配置，将数据源和jdbcTemplate都交给Spring来管理

### **6.2.1 Spring内置数据源**

DriverManagerDataSource是spring提供的数据源，但不推荐生产使用。DriverManagerDataSource的工作机制：有连接时就new一个connection，根本没有连接池作用。适合在本地测试等中使用，因为不需要额外的依赖类

查看该类源码，进行属性参数注入配置：

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"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

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

<!-- spring内置的连接池 -->

<bean id=*"dataSource"* class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>

<property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"*/>

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/test\_spring"*/>

<property name=*"username"* value=*"root"*/>

<property name=*"password"* value=*"root"*/>

</bean>

<bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>

<property name=*"dataSource"* ref=*"dataSource"*></property>

<!-- <constructor-arg name="dataSource" ref="dataSource"/> -->

</bean>

</beans>

使用SpringTest.java进行测试（使用Junit4这种测试需要引aop的包）

**package** com.test.jdbctemplate;

@RunWith(SpringJUnit4ClassRunner.**class**)

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

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

@Autowired

**private** JdbcTemplate jdbcTemplate;

@Test

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

**this**.jdbcTemplate.execute("create table test2(id int,name varchar(20),price decimal(10,2))");

}

}

测试结果：

成功创建表test2

### **6.2.2 Apache DBCP连接池**

Apache commons-dbcp需要导入：commons.dbcp-1.4.jar、commons.pool-1.6.jar

以下几个连接池是真正的连接池机制，配置销毁方法不是真正的销毁，而是关闭归还给连接池。其他连接参数这里不做详述，只是简单的配置四大金刚。

applicationContext.xml

<!-- Apache DBCP连接池 -->

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"* destroy-method=*"close"*>

<property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"*/>

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/test\_spring"*/>

<property name=*"username"* value=*"root"*/>

<property name=*"password"* value=*"root"*/>

</bean>

### **6.2.3 C3P0连接池**

导入C3P0的jar，版本不同需要导入的包可能不同：

c3p0-0.9.1.2.jar只需要导这1个包即可

c3p0-0.9.5.2.jar需要此包外，还需要mchange-commons-java-0.2.11.jar包

applicationContext.xml

<!-- C3P0连接池 -->

<bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"* destroy-method=*"close"*>

<property name=*"driverClass"* value=*"com.mysql.jdbc.Driver"*/>

<property name=*"jdbcUrl"* value=*"jdbc:mysql://localhost:3306/test\_spring"*/>

<property name=*"user"* value=*"root"*/>

<property name=*"password"* value=*"root"*/>

</bean>

### **6.2.4 Druid连接池**

导入druid-1.1.9.jar，下载http://central.maven.org/maven2/com/alibaba/druid/

applicationContext.xml

<!-- Druid连接池 -->

<bean id=*"dataSource"* class=*"com.alibaba.druid.pool.DruidDataSource"* destroy-method=*"close"*>

<property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"*/>

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/test\_spring"*/>

<property name=*"username"* value=*"root"*/>

<property name=*"password"* value=*"root"*/>

</bean>

### **6.2.5 HikariCP连接池**

导入该连接池包。该连接池使用的是slf4j日志实现，前面几种都是log4j实现，所以这里还需要额外引入转换包和slf4j的包：

HikariCP-2.5.1.jar

slf4j-api-1.7.25.jar

slf4j-log4j12-1.7.25.jar

日志配置不变，还是使用Log4j的配置即可

applicationContext.xml

<!-- HikariCP连接池 -->

<bean id=*"dataSource"* class=*"com.zaxxer.hikari.HikariDataSource"* destroy-method=*"close"*>

<property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"*/>

<property name=*"jdbcUrl"* value=*"jdbc:mysql://localhost:3306/test\_spring"*/>

<property name=*"username"* value=*"root"*/>

<property name=*"password"* value=*"root"*/>

</bean>

## **6.3外部属性文件的配置**

将xml配置中经常修改内容，抽取到properties文件，如数据库连接参数等。

第一步：创建jdbc.properties

jdbc.driverClassName=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/test\_spring

jdbc.username=root

jdbc.password=root

第二步： 修改applicationContext.xml，引入外部属性文件，使用EL表达式取值

<context:property-placeholder location=*"classpath:jdbc.properties"*/>

<!-- spring内置的连接池 -->

<bean id=*"dataSource"* class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>

<property name=*"driverClassName"* value=*"${jdbc.driverClassName}"*/>

<property name=*"url"* value=*"${jdbc.url}"*/>

<property name=*"username"* value=*"${jdbc.username}"*/>

<property name=*"password"* value=*"${jdbc.password}"*/>

</bean>

测试通过

## **6.4 JdbcTemplate的CURD**

第一步：创建一个表book：

CREATE TABLE book (id int(11) PRIMARY KEY NOT NULL AUTO\_INCREMENT COMMENT '主键', name varchar(20), price decimal(10,2)) DEFAULT CHARSET=utf8mb4

第二步：创建包com.test.jdbctemplate.crud，创建Book.java

**package** com.test.jdbctemplate.crud;

**public** **class** Book {

**private** Integer id;

**private** String name;

**private** BigDecimal price;

**public** Integer getId() {

**return** id;

}

**public** **void** setId(Integer id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** BigDecimal getPrice() {

**return** price;

}

**public** **void** setPrice(BigDecimal price) {

**this**.price = price;

}

**public** String toString() {

**return** "Book [id=" + id + ", name=" + name + ", price=" + price + "]";

}

}

第三步：编写Dao类，继承JdbcDaoSupport

参考https://docs.spring.io/spring/docs/4.3.2.BUILD-SNAPSHOT/javadoc-api/

**package** com.test.jdbctemplate.crud;

**public** **class** BookDao **extends** JdbcDaoSupport {

**public** **void** save(Book book) {

String sql = "insert into book values(null,?,?)";

//jdbcTemplate成员变量是父类private的，子类只能通过getter方法访问

**this**.getJdbcTemplate().update(sql,book.getName(),book.getPrice());

}

**public** **void** update(Book book) {

String sql = "update book set name=?,price=? where id=?";

**this**.getJdbcTemplate().update(sql,book.getName(),book.getPrice(),book.getId());

}

**public** Book getById(Integer id) {

String sql = "select \* from book where id=?";

**return** **this**.getJdbcTemplate().queryForObject(sql, **new** BeanPropertyRowMapper<>(Book.**class**), id);

}

**public** Integer getCount(Map<String, Object> map) {

String sql = "select count(\*) from book where name like ?";

**return** **this**.getJdbcTemplate().queryForObject(sql, Integer.**class**, "%" + map.get("keyWord") + "%");

}

**public** **void** deleteById(Integer id) {

String sql = "delete from book where id=?";

**this**.getJdbcTemplate().update(sql, id);

}

**public** List<Book> getAll() {

String sql = "select \* from book";

**return** **this**.getJdbcTemplate().query(sql, **new** BeanPropertyRowMapper(Book.**class**));

}

**public** List<Book> getByParam(Map<String, Object> map) {

String sql = "select \* from book where name like ?";

**return** **this**.getJdbcTemplate().query(sql, **new** BeanPropertyRowMapper(Book.**class**), "%" + map.get("name") + "%");

}

}

第四步：在ApplicationContext中配置bean和依赖注入

<context:property-placeholder location=*"classpath:jdbc.properties"*/>

<!-- spring内置的连接池 -->

<bean id=*"dataSource"* class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>

<property name=*"driverClassName"* value=*"${jdbc.driverClassName}"*/>

<property name=*"url"* value=*"${jdbc.url}"*/>

<property name=*"username"* value=*"${jdbc.username}"*/>

<property name=*"password"* value=*"${jdbc.password}"*/>

</bean>

<bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>

<property name=*"dataSource"* ref=*"dataSource"*></property>

<!-- <constructor-arg name="dataSource" ref="dataSource"/> -->

</bean>

<bean id=*"bookDao"* class=*"com.test.jdbctemplate.crud.BookDao"*>

<!-- 方式1：直接走setJdbcTemplate()，即注入jdbcTemplate实例 -->

<!-- <property name="jdbcTemplate" ref="jdbcTemplate"/> -->

<!-- 方式2：走setDataSource()，即注入dataSource实例，此时就不需要配置jdbcTemplate的bean了 -->

<property name=*"dataSource"* ref=*"dataSource"*/>

</bean>

第五步：编写单元测试SpringTest.java

**package** com.test.jdbctemplate.crud;

@RunWith(SpringJUnit4ClassRunner.**class**)

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

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

@Autowired

**private** BookDao bookDao;

@Test

**public** **void** testSave() {

Book book = **new** Book();

book.setName("MySQL优化");

book.setPrice(**new** BigDecimal(15.6));

bookDao.save(book);

}

@Test

**public** **void** testUpdate() {

Book book = **new** Book();

book.setId(1);

book.setName("MySQL优化1");

book.setPrice(**new** BigDecimal(15.61));

bookDao.update(book);

}

@Test

**public** **void** testGetById() {

System.***out***.println(bookDao.getById(1));

}

@Test

**public** **void** testGetCount() {

Map<String, Object> map = **new** HashMap<>(4);

map.put("keyWord", "1");

System.***out***.println(bookDao.getCount(map));

}

@Test

**public** **void** testDeleteById() {

bookDao.deleteById(1);

}

@Test

**public** **void** testGetAll() {

System.***out***.println(bookDao.getAll());

}

@Test

**public** **void** testGetByParam() {

Map<String, Object> map = **new** HashMap<>(4);

map.put("name", "MySQL");

System.***out***.println(bookDao.getByParam(map));

}

}