AOP 面向切面的编程

# 一 java代理

代理是一种常用的设计模式，其目的就是为其他对象提供一个代理以控制对某个对象的访问。通过代理，可以实现对目标对象的间接访问，即通过代理对象访问目标对象

代理模式中涉及到两方：

委托方、代理方

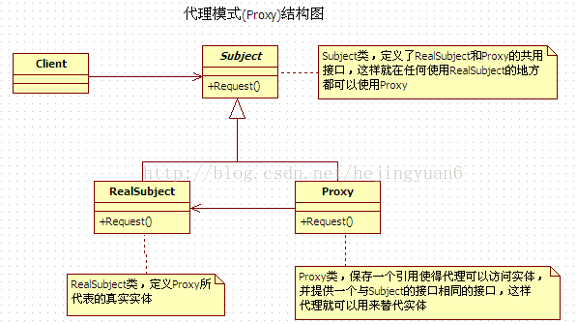
目标对象、代理对象

## 1 静态代理（掌握）

代理类和目标类实现相同的接口，在代理类中维护目标类的对象，以此实现对目标对象方法的调用。

优点：可以实现不修改目标对象代码的情况下，对目标对象的功能进行扩展。

缺点：代理对象与目标对象一样的接口，不易维护，一旦接口增加方法，则目标对象和代理类都需要维护



例如：老板通知开会，秘书传达指示

老板目标对象

秘书代理对象

动态代理

## 2 jdk代理

通过动态代理，可以动态的在内容中构建代理对象（在程序运行时运用反射机制动态创建）

使用jdk代理，要求目标对象必须实现了接口

1）定义接口

2）目标对象类实现接口

3）代理对象类实现InvocationHandler接口

**public** **class** XXX **implements** InvocationHandler{

//目标对象

**private** Object target;

@Override

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

// **TODO** Auto-generated method stub

String name = method.getName();

**if**(name.equals("xianglong18")){

//调用目标对象方法

Object ret = method.invoke(**this**.target, args);

System.***out***.println("增加的内容");

**return** ret;

}**else**{

**return** method.invoke(**this**.target, args);

}

}

//设置目标对象，返回代理对象

**public** Object bind(Object target){

**this**.target = target;

**return** Proxy.*newProxyInstance*(target.getClass().getClassLoader(), //类加载器

target.getClass().getInterfaces(), //对象接口

**this**);

}

}

## 3 cglib代理

针对没有实现接口的目标对象进行代理

原理：生成目标对象的子类，子类中覆盖父类的方法

需要支持cglib的jar文件

本例中使用spring中提供的相关类

//实现方法拦截器接口

**public** **class** BookProxy **implements** MethodInterceptor{

//目标对象

**private** Object target;

**public** Object bind(Object target){

**this**.target = target;

//增强类

Enhancer en = **new** Enhancer();

//设置父类

en.setSuperclass(**this**.target.getClass());

//设置回调

en.setCallback(**this**);

//创建代理对象

**return** en.create();

}

@Override

**public** Object intercept(Object proxy, Method arg1, Object[] arg2, MethodProxy methodProxy) **throws** Throwable {

// **TODO** Auto-generated method stub

System.***out***.println("调用方法前");

System.***out***.println("Method:" + arg1.getName());

System.***out***.println("Method:" + methodProxy.getSuperName());

//调用目标的方法（父类的方法）

Object ret= methodProxy.invokeSuper(proxy, arg2);

System.***out***.println("调用方法后");

**return** ret;

}

}

spring在运行期创建代理，有两种代理方式：

若目标对象实现了若干接口，spring就会使用JDK动态代理。

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

# 二 AOP

## 1 AOP概述

Aspect Oriented Program面向切面的编程

在运行时，动态地将代码切入到类的指定方法、指定位置上的编程思想就是面向切面的编程。

面向切面的编程将关注点的代码与业务逻辑分离

## 2 实现原理

AOP技术是建立在Java语言的反射机制与动态代理（jdk代理、cglib代理）机制之上的。

业务逻辑组件在运行过程中，AOP容器会动态创建一个代理对象供使用者调用，该代理对象已经将切面成功切入到目标方法的连接点上，从而使切面的功能与业务逻辑的功能同时得以执行。

调用者直接调用的其实是AOP容器动态生成的代理对象，再由代理对象调用目标对象完成原始的业务逻辑处理，而代理对象则已经将切面与业务逻辑方法进行了合成。

关注点（通知、增强）：重复执行的代码称为关注点

切面：关注点形成的类，切面类

切入点：需要执行的目标方法

织入

关注点是「干啥」，切入点是「啥时候干」

## 2 手动实现aop

## 3 xml配置实现aop

1. 需要导入jar包：

spring-aop-4.3.5.RELEASE.jar

aopalliance.jar

aspectjrt.jar aspectj-1.8.2\lib

aspectjweaver.jar aspectj-1.8.2\lib

1. 引入aop的名称空间

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

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

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

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

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

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

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

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

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

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

3)创建切面类对象和aop配置

<!-- aop的配置 -->

<aop:config>

<!-- 切入点，对哪些方法织入切面方法 -->

<!-- 第一个\* 任意访问权限（public、private等等）

com.rr.aop 包

第二个\* 包下面的任意类

\*(..) 任意名称和参数的方法

-->

<aop:pointcut expression=*"execution(\* com.rr.aop.\*.\*(..))"* id=*"pc"*/>

<!-- 切面类，通过ref指定对应的切面类的bean -->

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

<!-- 前置通知 ，执行目标方法前执行的方法 -->

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

<!-- 后置通知 ，执行目标方法后执行的方法 -->

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

<!-- 方法返回后通知 -->

<aop:after-returning method=*"afterReturn"* pointcut-ref=*"pc"*/>

<!-- 环绕通知 -->

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

</aop:aspect>

</aop:config>

切入点表达式用法：

参考帮助文档11.2.3 Declaring a pointcut

Some examples of common pointcut expressions are given below.

* the execution of any public method:

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

* the execution of any method with a name beginning with "set":

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

* the execution of any method defined by the AccountService interface:

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

* the execution of any method defined in the service package:

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

* the execution of any method defined in the service package or a sub-package:

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

* any join point (method execution only in Spring AOP) within the service package:

within(com.xyz.service.\*)

* any join point (method execution only in Spring AOP) within the service package or a sub-package:

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

* any join point (method execution only in Spring AOP) where the proxy implements the AccountService interface:

this(com.xyz.service.AccountService)

**五类通知类型：**

**前置通知[Before advice]**：在连接点前面执行，前置通知不会影响连接点的执行，除非此处抛出异常。   
**正常返回通知[After returning advice]**：在连接点正常执行完成后执行，如果连接点抛出异常，则不会执行。   
**异常返回通知[After throwing advice]**：在连接点抛出异常后执行。   
**返回通知[After (finally) advice]**：在连接点执行完成后执行，不管是正常执行完成，还是抛出异常，都会执行返回通知中的内容。   
**环绕通知[Around advice]**：环绕通知围绕在连接点前后，比如一个方法调用的前后。这是最强大的通知类型，能在方法调用前后自定义一些操作。环绕通知还需要负责决定是继续处理join point(调用ProceedingJoinPoint的proceed方法)还是中断执行。

## 4 注解方式实现aop

@Aspect

@PointCut

@Before

...

@Component

@Aspect //指定当前类为切面类

**public** **class** LogAop {

// 指定切入点表单式： 拦截哪些方法； 即为哪些类生成代理对象

@Pointcut("execution(\* com.rr.xmlaop.MyDao.\*(..))")

**public** **void** pointcut(){

}

//前置通知 执行目标方法前执行

@Before("pointcut()")

**public** **void** begin(JoinPoint jp) {

System.***out***.println("begin:");

System.***out***.println(jp.getSignature().getName());

}

//后置通知，执行目标方法后执行，不论是否异常

@After("pointcut()")

// public void end(){

**public** **void** end(JoinPoint jp) **throws** Exception, SecurityException {

}

//正常返回后通知，调用目标方法结束后执行，在代理返回前执行，异常不执行

@AfterReturning("pointcut()")

**public** **void** afterReturn(){

System.***out***.println("after return");

}

//环绕方法，必须使用ProceedingJoinPoint参数

@Around("pointcut()")

**public** **void** around(ProceedingJoinPoint pjp) **throws** Throwable{

System.***out***.println("环绕前....");

pjp.proceed(); // 执行目标方法

System.***out***.println("环绕后....");

}

}

<!-- aop的扫描 -->

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

# 三 JdbcTemplete （了解）

Spring提供的数据库的操作

1引入jar文件

spring-jdbc-4.3.5.RELEASE.jar

spring-tx-4.3.5.RELEASE.jar

用到数据库连接池，需要引入c3p0的jar文件

使用mysql数据库，导入mysql的驱动jar文件

2 引入tx命名空间

<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"*

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

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

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*

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

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

3 配置数据源和JdbcTemplete的bean

<!-- 数据源对象: C3P0连接池 -->

<bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>

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

<property name=*"jdbcUrl"* value=*"jdbc:mysql:///hib"*></property>

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

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

<property name=*"initialPoolSize"* value=*"3"*></property>

<property name=*"maxPoolSize"* value=*"10"*></property>

<property name=*"maxStatements"* value=*"100"*></property>

<property name=*"acquireIncrement"* value=*"2"*></property>

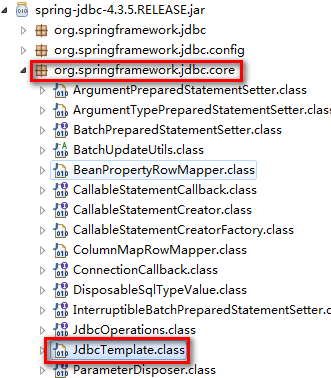
</bean>

<!-- JdbcTemplate工具类实例 -->

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

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

</bean>



3 调用

@Component

**public** **class** UserDao {

@Autowired

**private** JdbcTemplate jdbcTemplate;

**public** **void** add(){

//？ 占位符

String sql = "insert into t\_user(uname) values(?)";

//update方法可以执行增删改操作

jdbcTemplate.update(sql, "lisi");

}

**public** **void** findAll(){

String sql = "select \* from t\_user";

//返回集合数据

List<User> list = jdbcTemplate.query(sql, **new** BeanPropertyRowMapper<User>(User.**class**));

**for** (User user : list) {

System.***out***.println(user.getUname());

}

}

**public** **void** findAll2(){

String sql = "select uname from t\_user";

//返回集合数据

//错误写法

// List<User> list = jdbcTemplate.queryForList(sql, User.class);

List<String> list = jdbcTemplate.queryForList(sql, String.**class**);

**for** (String user : list) {

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

}

}

**public** **void** findAll3(){

String sql = "select \* from t\_user";

//返回集合数据,每个元素转为Map结构

List<Map<String, Object>> list = jdbcTemplate.queryForList(sql);

**for** (Map<String, Object> map : list) {

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

}

}

**public** **void** find(){

String sql = "select \* from t\_user where uid=2";

//查询单个数据，没有queryForInt

User user = jdbcTemplate.queryForObject(sql, **new** BeanPropertyRowMapper<User>(User.**class**));

System.***out***.println(user.getUname());

}

**public** **void** count(){

String sql = "select count(\*) from t\_user";

Integer count = jdbcTemplate.queryForObject(sql, Integer.**class**);

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

}

**public** **void** customer(){

String sql = "select \* from t\_user";

//自定义封装

List<User> list = jdbcTemplate.query(sql, **new** RowMapper<User>(){

@Override

**public** User mapRow(ResultSet rs, **int** rowNum) **throws** SQLException {

// **TODO** Auto-generated method stub

User u = **new** User();

u.setUid(rs.getInt("uid"));

u.setUname(rs.getString("uname"));

**return** u;

}

});

System.***out***.println(list.get(0).getUname());

}

}