

AOP

本章目标

- 掌握AOP基本概念
- 掌握@AspectJ支持
- 掌握基于XML的AOP配置
- 掌握动态代理机制

AOP介绍

- AOP介绍

- *Aspect-Oriented Programming* (AOP) , 切面编程
- AOP与*OOP(Object-Oriented Programming)*编程模式不同，提供了一种不同的编程思想。
- Spring IOC容器不依赖AOP，如果不需要可以不导入AOP相关包。
- Spring AOP提供了两种模式：
 - 基于XML的模式
 - 基于@AspectJ注解模式
- 基于Spring的AOP，重要应用有：
 - 用AOP声明性事务代替EJB的企业服务
 - 用AOP做日志处理
 - 用AOP做权限控制，如Spring Security

AOP介绍

- AOP中的专业术语

- **切面 (Aspect)** : 模块化关注多个类的共性处理。事务管理是J2EE应用中一个关于横切关注的很好的例子。
- **连接点 (Joinpoint)** : 在程序执行过程中某个特定的点, 比如某方法调用的时候或者处理异常的时候。在Spring AOP中, 一个连接点总是表示一个方法的执行。
- **通知 (Advice)** : 在切面的某个特定的连接点上执行的动作。其中包括了 “around” 、 “before” 和 “after” 等不同类型的通知。许多AOP框架 (包括Spring) 都是以拦截器做通知模型, 并维护一个以连接点为中心的拦截器链。
- **切入点 (Pointcut)** : 它由切入点表达式和签名组成。切入点如何与连接点匹配是AOP的核心, Spring缺省使用AspectJ切入点语法。

AOP介绍

- **引入 (Introduction)** : 用来给一个类型声明, 添加额外的方法或属性。Spring允许引入新的接口给任何被织入的对象。例如, 你可以通过引入, 使一个bean实现IsModified接口, 以便简化缓存机制。
- **目标对象 (Target Object)** : 被一个或者多个切面所通知的对象。也被称做被通知对象。既然Spring AOP是通过动态代理实现的, 这个对象永远是一个被代理对象。
- **AOP代理 (AOP Proxy)** : 动态代理, 在Spring中, AOP代理可以是JDK的Proxy或者CGLIB.
- **织入 (Weaving)** : 创建一个通知者, 把切面连接到其它的应用程序类型或者对象上。这些可以在编译时, 类加载时和运行时完成。Spring和其他纯Java AOP框架一样, 在运行时完成织入。

- advice的通知类型

- **前置通知 (Before advice)** : 在某连接点之前执行的通知 , 但这个通知不能阻止连接点之前的执行流程 (除非它抛出一个异常) 。
- **后置通知 (After returning advice)** : 在某连接点正常完成后执行的通知 : 例如 , 一个方法没有抛出任何异常 , 正常返回。
- **异常通知 (After throwing advice)** : 在方法抛出异常退出时执行的通知。
- **最终通知 (After (finally) advice)** : 当某连接点退出的时候执行的通知 (不论是正常返回还是异常退出) 。
- **环绕通知 (Around Advice)** : 包围一个连接点的通知 , 如方法调用。这是最强大的一种通知类型。环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回它自己的返回值或抛出异常来结束执行。

AOP介绍

- 通过切入点匹配连接点的概念是AOP的关键，这使得AOP不同于其它仅仅提供拦截功能的旧技术。
- 环绕通知是最常用的通知类型。和AspectJ一样，Spring提供所有类型的通知，推荐使用尽可能简单的通知类型来实现需要的功能。用最合适的通知类型可以使得编程模型变得简单，并且能够避免很多潜在的错误。比如，不需要在JoinPoint上调用用于环绕通知的proceed()方法，就不会有调用的问题。

- AOP动态代理选择

- Spring缺省使用标准JDK *动态代理 (dynamic proxies)* 来作为AOP的代理，这种模式的代理对象返回类型只能是接口。
- Spring也可以使用CGLIB代理. 对于需要代理类但是没有接口的时候，CGLIB代理是很有必要的。如果一个业务对象并没有实现一个接口，默认就会使用CGLIB。

AOP介绍

- JDK代理或CGLIB代理对象，都是通过ProxyFactoryBean创建而成：

```
public class ProxyFactoryBean extends ProxyCreatorSupport
    implements FactoryBean<Object>, BeanClassLoaderAware, BeanFactoryAware {

    /**
     * Return a proxy. Invoked when clients obtain beans from this factory bean.
     * Create an instance of the AOP proxy to be returned by this factory.
     * The instance will be cached for a singleton, and create on each call to
     * {@code getObject()} for a proxy.
     * @return a fresh AOP proxy reflecting the current state of this factory
     */
    @Override
    public Object getObject() throws BeansException {
        initializeAdvisorChain();
        if (isSingleton()) {
            return getSingletonInstance();
        }
        else {
            if (this.targetName == null) {
                logger.warn("Using non-singleton proxies with singleton targets is often undesirable. " +
                    "Enable prototype proxies by setting the 'targetName' property.");
            }
            return newPrototypeInstance();
        }
    }
}
```

注意：在Spring4.3中，已经打包了 CGLIB 3.2.4库，不要再额外导入其它版本的cglib库。在与hibernate或mybatis集成时，注意解决cglib的版本冲突问题。

@AspectJ支持

- @AspectJ介绍

- @AspectJ是一种风格样式，可以把Java的普通类，声明为一个切面。
- 在Spring4.3中使用@AspectJ，需要导入spring-aspects-4.3.19.RELEASE.jar，还需要导入依赖包aspectjweaver.jar(1.6.8 及以上版)

@AspectJ支持

- autoproxying配置
- @AspectJ 需要autoproxying配置，如下两种配置模式均可：

- Java类配置模式：

```
@Configuration
@EnableAspectJAutoProxy
public class AppConfig {

}
```

- XML配置模式：

```
<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"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-4.3.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.3.xsd">

  <aop:aspectj-autoproxy/>
```

@AspectJ支持

- 声明Aspect
 - A. 定义一个bean
 - B. 使用@Aspect声明一个切面

```
@Component  
@Aspect  
public class DbProxy {
```

@Aspect声明的类，和普通类一样可以添加属性和方法，还可以包含切入点、通知、引入。

@AspectJ支持

- 声明 Pointcut

- 切入点声明包含两个部分：

- 签名：由一个名字和多个参数组成
 - 切入点表达式

```
@Component
@Aspect
public class DbProxy {
```

```
    @Pointcut("execution(* transfer(..))")
```

```
    private void anyOldTransfer() {
```

```
}
```

//切入点表达式

//切入点签名，必须返回void

@AspectJ支持

- pointcut表达式

- pointcut中的指示符

- Spring AOP支持在切入点表达式中使用如下的AspectJ切入点指示符：

- execution* - 匹配方法执行的连接点，这是你将会用到的Spring的最主要的切入点指示符。
- within* - 限定匹配特定类型的连接点（在使用Spring AOP的时候，在匹配的类型中定义的方法的执行）。
- this* - 限定匹配特定的连接点（使用Spring AOP的时候方法的执行），其中bean reference（Spring AOP 代理）是指定类型的实例。
- target* - 限定匹配特定的连接点（使用Spring AOP的时候方法的执行），其中目标对象（被代理的应用对象）是指定类型的实例。
- args* - 限定匹配特定的连接点（使用Spring AOP的时候方法的执行），其中参数是指定类型的实例。
- @target* - 限定匹配特定的连接点（使用Spring AOP的时候方法的执行），其中正执行对象的类持有指定类型的注解。
- @args* - 限定匹配特定的连接点（使用Spring AOP的时候方法的执行），其中实际传入参数的运行时类型持有指定类型的注解。
- @within* - 限定匹配特定的连接点，其中连接点所在类型已指定注解（在使用Spring AOP的时候，所执行的方法所在类型已指定注解）。
- @annotation* - 限定匹配特定的连接点（使用Spring AOP的时候方法的执行），其中连接点的主题持有指定的注解

注意：对于JDK动态代理，只有public的接口方法调用能被拦截；对于cglib动态代理，public和protected方法调用，可以被拦截。对于Pointcut定义，可以应用于任何non-public的方法。

@AspectJ支持

- 联合使用Pointcut表达式

- 可以使用'&&' , '||' , '!' , 把多个Pointcut表达式 , 通过名字联合使用。示例如下 :

```
@Pointcut("execution(public * *(..))")  
private void anyPublicOperation() {}
```

```
@Pointcut("within(com.xyz.someapp.trading..*)")  
private void inTrading() {}
```

- 把上面的两个Pointcut联合使用如下 :

```
@Pointcut("anyPublicOperation() && inTrading()")  
private void tradingOperation() {}
```

@AspectJ支持

- 共享通用的Pointcut表达式

```
@Component
@Aspect
public class SystemArchitecture {
    /**
     * A join point is in the web layer if the method is defined
     * in a type in the com.xyz.someapp.web package or any sub-package under that.
     */
    @Pointcut("within(com.xyz.someapp.web..*)")
    public void inWebLayer() {}

    @Pointcut("within(com.xyz.someapp.service..*)")
    public void inServiceLayer() {}

    @Pointcut("within(com.xyz.someapp.dao..*)")
    public void inDataAccessLayer() {}

    /**
     * A business service is the execution of any method defined on a service
     * interface. This definition assumes that interfaces are placed in the
     * "service" package, and that implementation types are in sub-packages.
     */
    @Pointcut("execution(* com.xyz.someapp..service.*(..))")
    public void businessService() {}

    @Pointcut("execution(* com.xyz.someapp.dao.*(..))")
    public void dataAccessOperation() {}
}
```


@AspectJ支持

- execution指示器

- Spring AOP使用最多的就是execution指示器，格式如下：
- *execution(修饰符 返回类型 方法名(参数) 异常)*
- ret-type-pattern：使用最频繁返回类型模式是*，它代表了匹配任意的返回类型
- name-pattern：指方法名，可以使用*，通配全部或部分名字
- param-pattern：方法参数，（ ）表示无参；（..）表示0或任意个参数；（*）表示一个参数，类型任意；(*,String)，表示两个参数，第一个参数类型任意

@AspectJ支持

- 示例：

- 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接口定义的任何方法的执行：

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

- 在service包中定义的任何方法的执行：

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

- 在service包或其子包中定义的任何方法的执行：

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

@AspectJ支持

- 示例：

- 目标对象中有一个 @Transactional 注解的任意连接点（在Spring AOP中只是方法执行）

```
@target (org.springframework.transaction.annotation.Transactional)
```

'@target'在绑定表单中更加常用：- 请参见后面的通知一节中了解如何使得注解对象在通知体内可用。

- 任何一个目标对象声明的类型有一个 @Transactional 注解的连接点（在Spring AOP中只是方法执行）：

```
@within (org.springframework.transaction.annotation.Transactional)
```

'@within'在绑定表单中更加常用：- 请参见后面的通知一节中了解如何使得注解对象在通知体内可用。

- 任何一个执行的方法有一个 @Transactional 注解的连接点（在Spring AOP中只是方法执行）

```
@annotation (org.springframework.transaction.annotation.Transactional)
```

'@annotation'在绑定表单中更加常用：- 请参见后面的通知一节中了解如何使得注解对象在通知体内可用。

- 任何一个只接受一个参数，并且运行时所传入的参数类型具有@Classified 注解的连接点（在Spring AOP中只是方法执行）

```
@args (com.xyz.security.Classified)
```

'@args'在绑定表单中更加常用：- 请参见后面的通知一节中了解如何使得注解对象在通知体内可用。

- 任何一个在名为'tradeService'的Spring bean之上的连接点（在Spring AOP中只是方法执行）：

```
bean (tradeService)
```

- 任何一个在名字匹配通配符表达式'*Service'的Spring bean之上的连接点（在Spring AOP中只是方法执行）：

```
bean (*Service)
```

@AspectJ支持

- 示例：

- 在service包中的任意连接点（在Spring AOP中只是方法执行）：

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

- 在service包或其子包中的任意连接点（在Spring AOP中只是方法执行）：

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

- 实现了AccountService接口的代理对象的任意连接点（在Spring AOP中只是方法执行）：

```
this (com.xyz.service.AccountService)
```

'this'在绑定表单中更加常用：- 请参见后面的通知一节中了解如何使得代理对象在通知体内可用。

- 实现AccountService接口的目标对象的任意连接点（在Spring AOP中只是方法执行）：

```
target (com.xyz.service.AccountService)
```

'target'在绑定表单中更加常用：- 请参见后面的通知一节中了解如何使得目标对象在通知体内可用。

- 任何一个只接受一个参数，并且运行时所传入的参数是Serializable 接口的连接点（在Spring AOP中只是方法执行）

```
args (java.io.Serializable)
```

@AspectJ支持

- 声明advice:通知与Pointcut表达式关联，在Pointcut表达式关联匹配的方法前、后、环绕执行。你可以直接使用Pointcut表达式，也可以通过名字，引用已定义好的Pointcut表达式。

- 前置通知

@Before中，通过名字，引用公用的Pointcut表达式

```
@Aspect
public class BeforeExample {

    @Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
    public void doAccessCheck() {

    }

}
```

@Before中直接定义Pointcut表达式

```
@Aspect
public class BeforeExample {

    @Before("execution(* com.xyz.myapp.dao.*.*(..))")
    public void doAccessCheck() {

    }

}
```

@AspectJ支持

- 后置返回通知

当Pointcut匹配的代理方法，正常执行结束后，触发通知。它声明为@AfterReturning

```
@Aspect
public class AfterReturningExample {

    @AfterReturning("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
    public void doAccessCheck() {
    }
}
```

@AspectJ支持

- 后置异常通知

当Pointcut匹配的代理方法，异常结束后，触发通知。它声明为@AfterThrowing

@Aspect

public class AfterThrowingExample {

@AfterThrowing("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")

public void doRecoveryActions() {
}

@AspectJ支持

- 后置通知

后置通知，翻译成最终通知更贴切，即finally advice。
不论一个代理方法是如何结束的，最终通知都会运行。使用@After注解来声明。最终通知必须准备处理正常返回和异常返回两种情况。
通常用@After来释放资源，如数据库连接。

@Aspect

public class AfterFinallyExample {

@After("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")

public void doReleaseLock() {

//在此处释放数据库连接

}

@AspectJ支持

- 环绕通知


环绕通知在一个方法执行之前和之后执行。
环绕通知使用@Around注解来声明。
通知的第一个参数必须是 ProceedingJoinPoint类型。调用 ProceedingJoinPoint的proceed()方法，触发JoinPoint方法执行。

```
@Aspect
public class AroundExample {

    @Around("com.xyz.myapp.SystemArchitecture.businessService()")
    public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {

        //开始观察
        Object retVal = pjp.proceed();    //执行动态方法
        //停止观察

        return retVal;
    }
}
```



@AspectJ支持

- 给advice传递参数

- 给通知传递参数，org.aspectj.lang.JoinPoint常作为第一个参数使用。
- org.aspectj.lang.ProceedingJoinPoint是JoinPoint的子类，它只能用于环绕通知。
- JoinPoint中的方法：
 - getArgs()：返回代理对象的方法参数
 - getTarget()：返回目标对象
 - getSignature()：返回被通知方法的信息
 - toString()：打印被通知方法的有用信息

@AspectJ支持

```
@Component
@Aspect
public class BeforeExample {

    @Before("execution(* com.icss.biz.*(..))")
    public void doAccessCheck(JoinPoint jp) {
        System.out.println("taget:" + jp.getTarget().toString());
        System.out.println("tostring:" + jp.toString());
        System.out.println("args:" + jp.getArgs()[0].toString());
        System.out.println("signature:" + jp.getSignature().getName()
    }
}
```

```
public static void main(String[] args) {
    UserBiz iuser = (UserBiz)BeanFactory.getBean("userBiz");
    try {
        User user = iuser.login("admin", "123");
        if(user != null) {
            System.out.println("登录成功, 身份是" + user.getRole());
        }else {
            System.out.println("登录失败");
        }
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
```

信息: Loading XML bean definitions from class path resource [beans.xml]

taget:com.icss.biz.UserBiz@1c911a1

tostring::execution(User com.icss.biz.UserBiz.login(String,String))

args:admin

signature:login

UserBizMysql....login....

UserDaoMysql....login....

登录成功, 身份是1

注意：导包时很容易出错，应该使用org.aspectj包，下面的两个JoinPoint是有大小写区别的：
import org.aopalliance.intercept.Joinpoint;
import org.aspectj.lang.JoinPoint;

@AspectJ支持

• StaffUser日志管理案例

@Component

@Aspect

public class LogProxy {

@Around("execution(public * com.icss.biz.*(..))")

public Object logging(ProceedingJoinPoint pjp) throws Throwable{

Object obj = null;

try {

Log.Logger.info(pjp.getSignature() + new Date().toString());

obj = pjp.proceed();

Log.Logger.info(pjp.toString() + new Date().toString());

} catch (Throwable e) {

throw e;

return obj;

}

使用环绕通知配置日志代理

//注意必须要有返回值

//必须要抛出异常

INFO - User com.icss.biz.UserBiz.login(String,String)Sun Nov 10 19:49:06 CST 2019

UserBizMysql....login....

UserDaoMysql....login....

INFO - execution(User com.icss.biz.UserBiz.login(String,String))Sun Nov 10 19:49:06 CST 2019

登录成功, 身份是1

public static void main(String[] args) {

UserBiz iuser = (UserBiz)BeanFactory.getBean("userBiz");

try {

User user = iuser.login("admin", "123");

if(user != null) {

System.out.println("登录成功, 身份是" + user.getRole());

}else {

System.out.println("登录失败");

}

} catch (Exception e) {

System.out.println(e.getMessage());

}

登录测试

测试结果

@AspectJ支持

• StaffUser数据库连接管理

- 使用AOP，实现所有业务层方法，自动打开数据库，自动关闭数据库

```
@Component
@Aspect
public class DbProxy {

    @Pointcut("execution(public * com.icss.biz.*(..))")
    private void businessOperate() {

    }

    @Before("businessOperate()")
    public void openDataBase(JoinPoint jp) {
        System.out.println("-----打开数据库-----");
    }

    @After("businessOperate()")
    public void closeDataBase(JoinPoint jp) {
        System.out.println(jp.toString());
        System.out.println("-----关闭数据库-----");
    }
}
```

使用@Before打开数据库，使用
@After关闭数据库

```
public static void main(String[] args) {
    UserBiz iuser = (UserBiz)BeanFactory.getBean("userBiz");
    try {
        User user = iuser.login("admin", "123");
        if(user != null) {
            System.out.println("登录成功，身份是" + user.getRole());
        } else {
            System.out.println("登录失败");
        }
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
```

```
INFO - Loading XML bean definitions from class path resource [beans.xml]
-----打开数据库-----
INFO - User com.icss.biz.UserBiz.login(String,String)Sun Nov 10 21:32:03 CST 2019
UserBizMysql....login....
UserDaoMysql....login....
execution(User com.icss.biz.UserBiz.login(String,String))
-----关闭数据库-----
异常测试...
```

测试，即使出现异常，也要确保关闭
数据库

基于XML的AOP配置

- 说明
 - 如果你喜欢XML格式，Spring通过"aop" 命名标签，提供了切面编程支持。
 - 与@AspectJ风格一致的Pointcut表达式和Advice，在XML中同样支持。
 - 在Spring的配置文件中，所有的切面和通知都必须定义在<aop:config>元素内部（context可以包含多个 <aop:config> ）。
 - 一个<aop:config>可以包含pointcut，advisor和aspect元素（注意这三个元素必须按照这个顺序进行声明）。


警告：

- 1、<aop:config>风格配置使得Spring的auto-proxying机制**变得很笨重**。
- 2、如果你已经通过 BeanNameAutoProxyCreator或类似的东西显式使用auto-proxying，它可能会导致问题（例如通知没有被织入）。
- 3、推荐的使用模式是**仅仅使用<aop:config>风格，或者仅仅使用AutoProxyCreator风格**。

基于XML的AOP配置

- 声明Aspect

使用<aop:aspect>来声明切面，通过ref可以引用支撑bean



```
<aop:config>
  <aop:aspect id="myAspect" ref="aBean">
    ...
  </aop:aspect>
</aop:config>
<bean id="aBean" class="...">
  ...
</bean>
```

注意：切面中要配置通知，通知需要代码实现，因此必须要有一个支撑bean

基于XML的AOP配置

• 声明Pointcut

```
<aop:config>
  <aop:pointcut id="businessService"
    expression="execution(* com.xyz.myapp.service.*(..))" />
</aop:config>
```

<aop:pointcut>应该声明在<aop:config>内，这样其它aspect和advisor就可以共享这个Pointcut。

XML中的pointcut表达式，与我们前面在@AspectJ中使用的pointcut表达式语法完全一致。
你也可以通过名字引用其它pointcut：

```
<aop:config>
  <aop:pointcut id="businessService"
    expression="com.xyz.myapp.SystemArchitecture.businessService()" />
</aop:config>
```


基于XML的AOP配置

• 声明Advice

```
<aop:config>
  <aop:aspect id="example" ref="aBean">
    <aop:before pointcut-ref="dataAccessOperation"
      method="doAccessCheck" />
    <aop:after-returning
      pointcut-ref="dataAccessOperation" method="doAccessCheck" />
    <aop:after-throwing
      pointcut-ref="dataAccessOperation" method="doRecoveryActions" />
    <aop:after pointcut-ref="dataAccessOperation"
      method="doReleaseLock" />
    <aop:around pointcut-ref="businessService"
      method="doBasicProfiling" />
  </aop:aspect>
</aop:config>

<bean id="aBean" class="com.icss.biz.ABean" ></bean>
```

```
public class ABean {
```

```
    public void doAccessCheck(JoinPoint jp) {
    }
```

```
    public void doRecoveryActions(JoinPoint jp ) {
    }
```

```
    public void doReleaseLock() {
    }
```

```
    public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable{
        return pjp.proceed();
    }
}
```

<在@AspectJ中定义的5个通知类型，XML中都支持，而且语意相同

基于XML的AOP配置

• 使用Advisor

```
<aop:config>
  <aop:pointcut id="businessService"
    expression="execution(* com.xyz.myapp.service.*(..))" />
  <aop:advisor pointcut-ref="businessService"
    advice-ref="tx-advice" />
</aop:config>
<tx:advice id="tx-advice">
  <tx:attributes>
    <tx:method name="*" propagation="REQUIRED" />
  </tx:attributes>
</tx:advice>
```

示例：Advisor与事务通知策略协同使用

Advisor(通知器)这个概念源于Spring对AOP的支持，在@AspectJ中没有等价内容。一个Advisor就是一个自包含的切面，Advisor中只能有一个通知。Aspect需要一个bean的支持，Advisor不需要，这样配置更加简单。

```
<bean id="txManager"
  class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource" />
</bean>
<aop:config>
  <aop:pointcut id="fooServiceOperation"
    expression="execution(* com.icss.biz.*(..))" />
  <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceOperation" />
</aop:config>
<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
    <tx:method name="add*" propagation="REQUIRED" rollback-for="Throwable" />
    <tx:method name="*" read-only="true" />
  </tx:attributes>
</tx:advice>
```

基于XML的AOP配置

- StaffUser日志管理案例
 - 步骤一：编写日志处理代码

```
public class LogProxy {  
  
    @Around("execution(public * com.icss.biz.*(..))")  
    public Object logging(ProceedingJoinPoint pjp) throws Throwable{  
        Object obj = null;  
        try {  
            Log.logger.info(pjp.getSignature() + new Date().toString());  
            obj = pjp.proceed();  
            Log.logger.info(pjp.toString() + new Date().toString());  
        } catch (Throwable e) {  
            throw e;  
        }  
  
        return obj;  
    }  
}
```

//注意必须要有返回值

//必须要抛出异常

基于XML的AOP配置

- 步骤三：测试

```
public static void main(String[] args) {  
    UserBiz iuser = (UserBiz)BeanFactory.getBean("userBiz");  
    try {  
        User user = iuser.login("admin", "123");  
        if(user != null) {  
            System.out.println("登录成功, 身份是" + user.getRole());  
        }else {  
            System.out.println("登录失败");  
        }  
    } catch (Exception e) {  
        System.out.println(e.getMessage());  
    }  
}
```

INFO - User com.icss.biz.UserBiz.login(String,String)Mon Nov 11 11:43:16 CST 2019

UserBizMysql....login....

UserDaoMysql....login....

INFO - execution(User com.icss.biz.UserBiz.login(String,String))Mon Nov 11 11:43:16 CST 2019

登录成功, 身份是1

基于XML的AOP配置

- 步骤四：给服务层和持久层方法都打日志（在XML中使用& & &表示&&，也可以使用and连接）

```
<aop:config>
  <aop:pointcut id="allLayer"
    expression="execution(* com.icss.biz.*(..)) or execution(* com.icss.dao.*(..))" />
  <aop:aspect id="logAspect" ref="logBean">
    <aop:around method="logging" pointcut-ref="allLayer"/>
  </aop:aspect>
</aop:config>
```

```
<bean id="logBean" class="com.icss.biz.LogProxy"></bean>
```

INFO - User com.icss.biz.UserBiz.login(String,String)Mon Nov 11 14:59:02 CST 2019

UserBizMysql....login....

INFO - User com.icss.dao.IUserDao.login(String,String)Mon Nov 11 14:59:02 CST 2019

UserDaoMysql....login....

INFO - execution(User com.icss.dao.IUserDao.login(String,String))Mon Nov 11 14:59:02 CST 2019

INFO - execution(User com.icss.biz.UserBiz.login(String,String))Mon Nov 11 14:59:02 CST 2019

登录成功, 身份是1

基于XML的AOP配置

- StaffUser数据库连接管理案例

- 步骤一：编写数据库打开关闭的代码

```
public class DbProxy {  
  
    public void openDataBase(JoinPoint jp) {  
        System.out.println("-----打开数据库-----");  
    }  
  
    public void closeDataBase(JoinPoint jp) {  
        System.out.println(jp.toString());  
        System.out.println("-----关闭数据库-----");  
    }  
}
```

基于XML的AOP配置

- 步骤二：编写XML配置信息

```
<aop:config>
    <aop:pointcut expression="execution(public * com.icss.biz.*.*(..))" id="businessOperate"/>
    <aop:aspect id="dbAspect" ref="dbBean">
        <aop:before method="openDataBase" pointcut-ref="businessOperate"/>
        <aop:after method="closeDataBase" pointcut-ref="businessOperate"/>
    </aop:aspect>
</aop:config>

<bean id="dbBean" class="com.icss.biz.DbProxy"></bean>

<aop:config>
    <aop:aspect id="logAspect" ref="logBean">
        <aop:around method="logging" pointcut-ref="businessOperate"/>
    </aop:aspect>
</aop:config>
<bean id="logBean" class="com.icss.biz.LogProxy"></bean>
```

基于XML的AOP配置

- 步骤三：测试

-----打开数据库-----

```
INFO - User com.icss.biz.UserBiz.login(String,String)Mon Nov 11 15:30:07 CST 2019
```

```
UserBizMysql....login....
```

```
UserDaoMysql....login....
```

```
INFO - execution(User com.icss.biz.UserBiz.login(String,String))Mon Nov 11 15:30:07 CST 2019
```

```
execution(User com.icss.biz.UserBiz.login(String,String))
```

-----关闭数据库-----

登录成功，身份是1

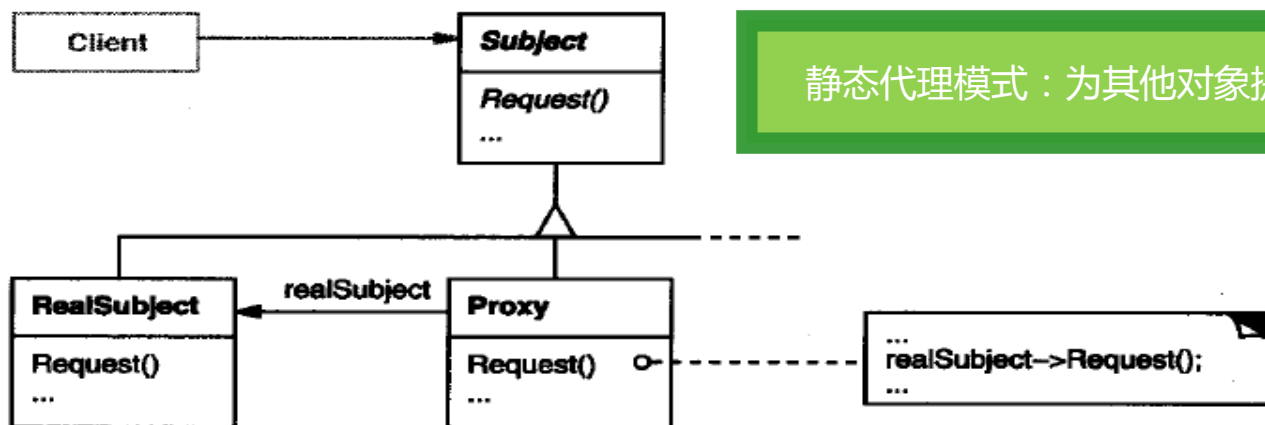
动态代理机制

- 说明：

- Spring AOP部分使用JDK动态代理，部分使用CGLIB来为目标对象创建代理。
- 如果被代理的目标对象实现了至少一个接口，则会使用JDK动态代理。所有该目标类型实现的接口都将被代理。若该目标对象没有实现任何接口，则创建一个CGLIB代理。

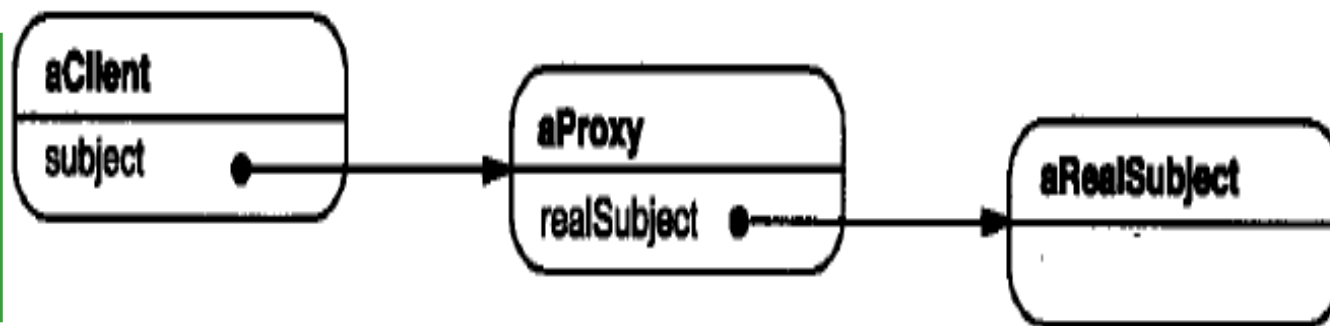
动态代理机制

- 静态代理之设计模式之静态代理



静态代理模式：为其他对象提供一种代理以控制对这个对象的访问。

客户向代理发出请求，真正做事的是被代理者；代理模式被广泛应用，如hibernate的懒加载机制。



动态代理机制

- 静态代理之明星与经纪人案例

- 步骤一：接口设计

```
public interface IStar {  
  
    public void playMovie();           // 演电影  
    public void playAd();              // 拍广告  
    public void playActive();          // 参加活动  
}
```

动态代理机制

- 步骤二：业务类

```
public class MovieStar implements IStar{
    private String name;

    public String getName() {
        return name;
    }
    public MovieStar(String name) {
        this.name = name;
    }

    @Override
    public void playMovie() {
        System.out.println(this.name + " is 拍电影...");
    }

    @Override
    public void playAd() {
        System.out.println(this.name + " is 拍广告...");
    }

    @Override
    public void playActive() {
        System.out.println(this.name + " is 参加活动...");
    }
}
```

动态代理机制

- 步骤三：静态代理类

```
public class StarProxy implements IStar{

    private String name;
    private MovieStar star;

    public StarProxy(String name,MovieStar star) {
        this.name = name;
        this.star = star;
        System.out.println(this.name + "代理" + star.getName() + "工作");
    }
    @Override
    public void playMovie() {
        star.playMovie();
        System.out.println(this.name + "揩油15万, 主动照顾" + star.getName() + "家属...");
    }
    @Override
    public void playAd() {
        star.playAd();
        System.out.println(this.name + "揩油3万....");
    }
    @Override
    public void playActive() {
        star.playActive();
        System.out.println(this.name + "揩油8万....");
    }
}
```

动态代理机制

- 步骤四：代码测试

```
public static void main(String[] args) {  
    MovieStar star = new MovieStar("王宝强");  
    IStar proxy = new StarProxy("宋喆", star);  
    proxy.playActive();  
    proxy.playAd();  
    proxy.playMovie();  
}
```

宋喆代理王宝强工作

王宝强 is 参加活动...

宋喆揩油8万.....

王宝强 is 拍广告...

宋喆揩油3万.....

王宝强 is 拍电影...

宋喆揩油15万，主动照顾宝强家属...

动态代理机制

• JDK动态代理Proxy之静态代理的日志实现案例

代理类

```
public interface IBook {  
    public void getBookInfo();  
    public void buyBook();  
    public void updateBook();  
}
```



```
public class BookBiz implements IBook{  
  
    @Override  
    public void getBookInfo() {  
        System.out.println("getBookInfo...");  
    }  
  
    @Override  
    public void buyBook() {  
        System.out.println("buyBook...");  
    }  
  
    @Override  
    public void updateBook() {  
        System.out.println("updateBook...");  
    }  
}
```



```
public class BookLog implements IBook{  
  
    private BookBiz bookBiz;  
  
    public BookLog(BookBiz bookBiz) {  
        this.bookBiz = bookBiz;  
    }  
    @Override  
    public void getBookInfo() {  
        Log.Logger.info("getBookInfo...begin...");  
        bookBiz.getBookInfo();  
        Log.Logger.info("getBookInfo...end...");  
    }  
    @Override  
    public void buyBook() {  
        Log.Logger.info("buyBook...begin...");  
        bookBiz.buyBook();  
        Log.Logger.info("buyBook...end...");  
    }  
    @Override  
    public void updateBook() {  
        Log.Logger.info("updateBook...begin...");  
        bookBiz.updateBook();  
        Log.Logger.info("updateBook...end...");  
    }  
}
```



动态代理机制

- 步骤四：测试

```
public static void main(String[] args) {  
    BookBiz bookBiz = new BookBiz();  
    IBook ibook = new BookLog(bookBiz);  
    ibook.buyBook();  
}
```

```
INFO - buyBook...begin...  
buyBook...  
INFO - buyBook...end...
```


动态代理机制

- 提问：接口IUser中的方法，也要实现日志，怎么办？还使用静态代理的话，是不是太繁琐了？

```
public interface IUser {  
  
    public void regist(User user) throws Exception;  
  
    public void login(String uname,String pwd) throws Exception;
```

动态代理机制

- JDK动态代理Proxy之Proxy类和InvocationHandler接口案例

```
public class Proxy  
extends Object  
implements Serializable
```

Proxy提供了创建动态代理类和实例的静态方法，它也是由这些方法创建的所有动态代理类的超类。

为某个接口创建代理Foo：

```
InvocationHandler handler = new MyInvocationHandler(...);  
Class<?> proxyClass = Proxy.getProxyClass(Foo.class.getClassLoader(), Foo.class);  
Foo f = (Foo) proxyClass.getConstructor(InvocationHandler.class).  
    newInstance(handler);
```

或更简单地：

```
Foo f = (Foo) Proxy.newProxyInstance(Foo.class.getClassLoader(),  
    new Class<?>[] { Foo.class },  
    handler);
```

动态代理机制

- JDK动态代理Proxy之JDK动态代理日志案例

- 步骤一：编写动态代理类

```
public class LogDynamic implements InvocationHandler{
```

```
    private Object target;
```

```
    /**
```

```
     * 传入被代理对象，返回代理对象
```

```
     * @param target
```

```
     * @return
```

```
     */
```

```
    public Object bind(Object target) {
```

```
        this.target = target;
```

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

```
    }
```

```
    /**
```

```
     * 当调用被代理对象上的方法时，这个invoke方法会被自动激活
```

```
     */
```

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

```
        Log.Logger.info(method.getName() + " beging...");
```

```
        //动态调用被代理对象的方法
```

```
        Object result = method.invoke(target, args);
```

```
        Log.Logger.info(method.getName() + " ending...");
```

```
        return result;
```

```
    }
```

动态代理机制

- 步骤二：调用测试

```
public static void main(String[] args) {  
    LogDynamic logd = new LogDynamic();  
  
    IUser proxy = (IUser)logd.bind(new UserBiz());  
    try {  
        proxy.login("tom", "123445");  
        User user = new User();  
        user.setUname("王五");  
        proxy.regist(user);  
    } catch (Exception e) {  
    }  
    IBook bookProxy = (IBook)logd.bind(new BookBiz());  
    bookProxy.buyBook();  
    bookProxy.getBookInfo();  
}
```

动态代理机制

- 步骤四：测试结果，IUser和IBook都可以动态绑定，动态打印日志

```
INFO - login beging....  
tom is logging....  
INFO - login ending....  
INFO - regist beging....  
王五 is regist....  
INFO - regist ending....  
INFO - buyBook beging....  
buyBook...  
INFO - buyBook ending....  
INFO - getBookInfo beging....  
getBookInfo...  
INFO - getBookInfo ending....
```

动态代理机制

- 自动管理StaffUser数据库连接案例之配置mysql数据库环境

- A、安装mysql8数据库

- B、创建库： create database staff;打开库： use staff;

- C、创建表：

```
create table TStaff
(
    sno                varchar(9) not null,
    name               varchar(30),
    birthday            date,
    address             varchar(180),
    tel                 varchar(18),
    primary key (sno)
);

create table TUser
(
    uname              varchar(30) not null,
    sno                 varchar(9) not null,
    pwd                 varchar(20),
    role                int,
    primary key (sno)
);

alter table TUser add constraint FK_Reference_1 foreign key (sno)
references TStaff (sno) ;
```

动态代理机制

- D、StaffUser系统导入mysql驱动包：mysql-connector-java-8.0.11.jar
- E、配置数据库的连接信息，新建db.properties文件

```
driver=com.mysql.cj.jdbc.Driver
```

```
url=jdbc:mysql://localhost:3306/book2?useSSL=false&serverTimezone=UTC&allowPublicKeyRetrieval=true
```

```
username=root
```

```
password=123456
```

动态代理机制

- F、封装单例类DbInfo，读取数据库连接信息

```
public class DbInfo {  
    private static DbInfo info;    //单例  
    private String driver;  
    private String url;  
    private String uname;  
    private String pwd;  
  
    public String getDriver() {  
        return driver;  
    }  
    public String getUrl() {  
        return url;  
    }  
    public String getUname() {  
        return uname;  
    }  
    public String getPwd() {  
        return pwd;  
    }  
    private DbInfo() {  
        //构造函数私有化  
    }  
}
```

```
static {  
    info = new DbInfo();  
    InputStream fis = null;  
    try {  
        String fname = DbInfo.class.getResource("/").getPath() + "db.properties";  
        Properties prop = new Properties();  
        fis = new FileInputStream(new File(fname));  
        prop.load(fis);  
        info.driver = prop.getProperty("driver");  
        info.url = prop.getProperty("url");  
        info.uname = prop.getProperty("username");  
        info.pwd = prop.getProperty("password");  
    } catch (Exception e) {  
        Log.Logger.error(e.getMessage());  
    } finally {  
        if (fis != null) {  
            try {  
                fis.close();  
            } catch (IOException e) {  
                Log.Logger.error(e.getMessage());  
            }  
        }  
    }  
}  
  
public static DbInfo instance(){  
    return info;  
}
```


动态代理机制

• 自动管理StaffUser数据库连接案例之DbFactory封装数据库Connection

```
public class DbFactory {
    private static ThreadLocal<Connection> tlocal = new ThreadLocal<>(); //单例对象,存储所有客户的Connection对象

    public static Connection openConnection() throws ClassNotFoundException, SQLException {
        Connection conn = tlocal.get(); //获得当前线程原来存入的Connection对象
        try {
            if(conn == null || conn.isClosed()) {
                DbInfo db = DbInfo.instance();
                Class.forName(db.getDriver()); //加载驱动
                conn = DriverManager.getConnection(db.getUrl(), db.getUserName(), db.getPwd());
                tlocal.set(conn); //当前线程,第一次生成的Connection对象,存入ThreadLocal
                Log.logger.info(Thread.currentThread().getId() + "打开数据库,生成一个新连接.....");
            } else {
                Log.logger.info(Thread.currentThread().getId() + "打开数据库,使用原有连接.....");
            }
        } catch (ClassNotFoundException e) {
            Log.logger.error(e.getMessage(), e);
            throw e; //二次抛出异常
        } catch (SQLException e) {
            Log.logger.error(e.getMessage(), e);
            throw e;
        }

        return conn;
    }

    public static void closeConnection() {
        Log.logger.info(Thread.currentThread().getId() + "关闭数据库.....");
        Connection conn = tlocal.get();
        tlocal.set(null); //当前线程tlocal中存储的是null
        if(conn != null) {
            try {
                conn.close();
            } catch (Exception e) {
                Log.logger.error(e.getMessage(), e);
            }
        }
    }
}
```

动态代理机制

- 自动管理StaffUser数据库连接案例之JDK动态代理实现数据库的自动打开和关闭

```
public class DbProxy implements InvocationHandler {  
  
    private Object target;  
  
    public Object bind(Object target) {  
        this.target = target;  
        return Proxy.newProxyInstance(target.getClass().getClassLoader(),  
                                       target.getClass().getInterfaces(), this);  
    }  
  
    @Override  
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
  
        Object result = null;  
  
        DbFactory.openConnection();                                //自动打开数据库，获得一个新连接  
        try {  
            result = method.invoke(target, args);  
        } finally {  
            DbFactory.closeConnection();                          //业务操作完毕，关闭数据库连接  
        }  
  
        return result;  
    }  
}
```

动态代理机制

- 代码测试：
 - A、服务层代码

```
public class UserBiz implements IUser{

    @Override
    public User login(String uname, String pwd) throws Exception {
        Log.Logger.info(Thread.currentThread().getId() + ":" + uname + " login ....");
        IUserDao dao = new UserDaoMysql();
        return dao.login(uname, pwd);
    }
}
```

动态代理机制

- B、持久层代码

```
public class UserDaoMysql implements IUserDao {  
  
    public User login(String name, String pwd) throws Exception {  
        User user = null;  
  
        DbFactory.openConnection();  
  
        // 用伪代码模拟用户登录  
        if (name.equals("admin") && pwd.equals("123")) {  
            user = new User();  
            user.setRole(IRole.ADMIN);  
            user.setUname(name);  
            user.setPwd(pwd);  
        } else if (name.equals("tom") && pwd.equals("123")) {  
            user = new User();  
            user.setRole(IRole.COMMON_USER);  
            user.setUname(name);  
            user.setPwd(pwd);  
        } else if (name.equals("jack") && pwd.equals("123")) {  
            user = new User();  
            user.setRole(IRole.VIP_USER);  
            user.setUname(name);  
            user.setPwd(pwd);  
        } else {  
            }  
        throw new Exception("异常测试...");  
        // return user;  
    }  
}
```

动态代理机制

- C、UI代码调用

```
//多线程模拟多用户并发访问
ExecutorService pool = Executors.newCachedThreadPool();
for(int i=0;i<3;i++) {
    pool.execute(new Runnable() {
        @Override
        public void run() {
            DbProxy proxy = new DbProxy();
            IUser userProxy = (IUser)proxy.bind(new UserBiz());
            try {
                userProxy.login("tom", "123456");
            } catch (Exception e) {
            }
        }
    });
}
pool.shutdown();
```

```
INFO - 8打开数据库,生成一个新连接.....
INFO - 10打开数据库,生成一个新连接.....
INFO - 10:tom login .....
INFO - 9打开数据库,生成一个新连接.....
INFO - 9:tom login .....
INFO - 8:tom login .....
INFO - 8打开数据库,使用原有连接.....
INFO - 9打开数据库,使用原有连接.....
INFO - 10打开数据库,使用原有连接.....
INFO - 9关闭数据库.....
INFO - 8关闭数据库.....
INFO - 10关闭数据库.....
```

测试发现，即使业务操作中出现异常，也不影响数据库连接的获得与释放。

动态代理机制

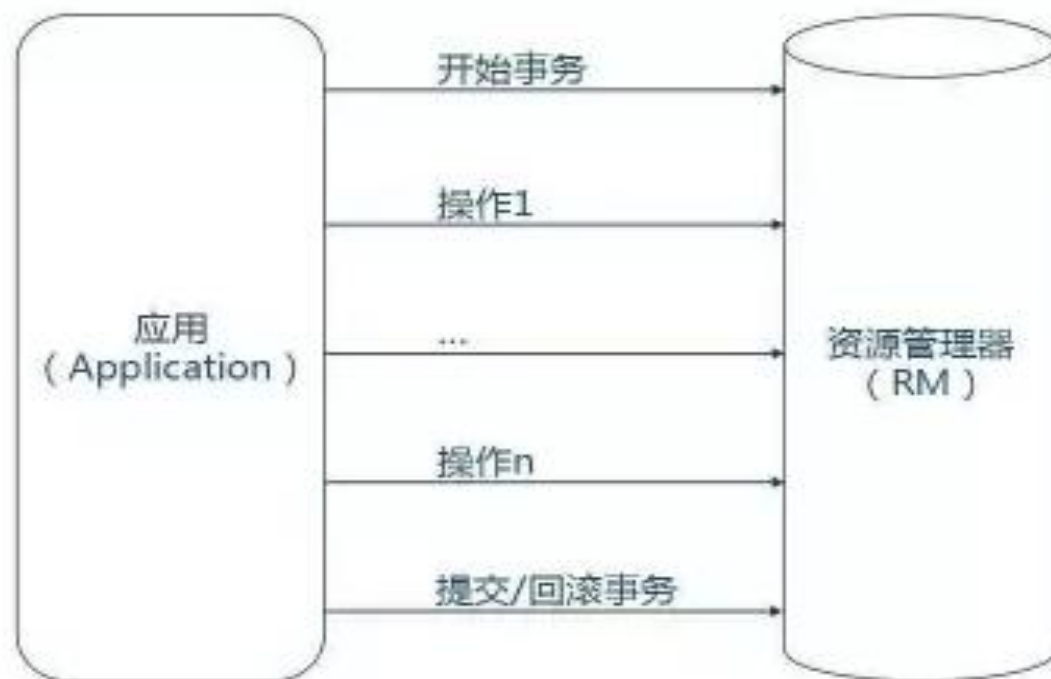
- StaffUser系统的事务处理案例之事务ACID特性

- 本地事务：local transaction，使用单一资源管理器，管理本地资源

- 本地事务与ACID:

- 关系数据库事务

- Atomicity (原子性)
 - Consistency (一致性)
 - Isolation (隔离性)
 - Durability (持久性)



动态代理机制

- StaffUser系统的事务处理案例之程式本地事务新增员工和用户
 - 添加新员工时，自动添加一个系统的登录用户。
 - 用户名默认为员工编号、密码默认为身份证后6位，用户第一次登录提示修改。
 - 用户名为用户的姓名、或昵称，用于显示，不做唯一性校验。
 - 用事务的ACID特性，保证员工和默认用户同时添加成功。

动态代理机制

- 步骤一： DbFactory中封装JDBC的事务操作

```
public class DbFactory {  
    private static ThreadLocal<Connection> tlocal = new ThreadLocal<>();  
    /**  
     * 开启事务  
     * @throws Exception  
     */  
    public static void beginTransaction() throws Exception{  
        Connection conn = openConnection();  
        conn.setAutoCommit(false);  
        Log.logger.info(Thread.currentThread().getId() + "开启事务");  
    }  
    /**  
     * 提交事务  
     * @throws Exception  
     */  
    public static void commit() throws Exception {  
        Connection conn = openConnection();  
        conn.commit();  
        Log.logger.info(Thread.currentThread().getId() + "提交事务");  
    }  
    /**  
     * 事务回滚  
     * @throws Exception  
     */  
    public static void rollback() throws Exception{  
        Connection conn = openConnection();  
        conn.rollback();  
        Log.logger.info(Thread.currentThread().getId() + "回滚事务");  
    }  
}
```


动态代理机制

- 步骤二：服务层调用事务，添加员工和用户

```
public class StaffBiz implements IStaff {  
  
    /**  
     * 使用本地事务，保证员工和用户信息同时写入成功  
     */  
    public void addStaffUser(TStaff staff, TUser user) throws Exception {  
        //打开数据库  
        DbFactory.openConnection();  
        //开启事务  
        DbFactory.beginTransaction();  
        StaffDao staffDao = new StaffDao();  
        UserDao userDao = new UserDao();  
        try {  
            staffDao.addStaff(staff);  
            userDao.addUser(user);  
            //无异常，提交事务  
            DbFactory.commit();  
        } catch (Exception e) {  
            //异常，回滚事务  
            DbFactory.rollback();  
            //异常信息，要二次抛出  
            throw e;  
        }  
        //关闭数据库  
        DbFactory.closeConnection();  
    }  
}
```

动态代理机制

- 步骤三：在持久层分别添加员工和用户

```
public class StaffDao {
```

```
    /**
     * 添加员工
     * @param staff
     * @throws Exception
     */
    public void addStaff(TStaff staff) throws Exception{
        String sql = "insert into tstaff values(?,?,?,?,?)";
        Connection conn = DbFactory.openConnection();
        PreparedStatement ps = conn.prepareStatement(sql);
        ps.setString(1, staff.getSno());
        ps.setString(2, staff.getName());
        ps.setDate(3, new java.sql.Date(staff.getBirthDay().getTime()));
        ps.setString(4, staff.getAddress());
        ps.setString(5, staff.getTel());
        ps.executeUpdate();
        ps.close();
    }
```

```
public class UserDao {
```

```
    /**
     * 添加用户
     * @param user
     * @throws Exception
     */
    public void addUser(TUser user) throws Exception {
        String sql = "insert into tuser values(?,?,?,?,?)";
        Connection conn = DbFactory.openConnection();
        PreparedStatement ps = conn.prepareStatement(sql);
        ps.setString(1, user.getUsername());
        ps.setString(2, user.getSno());
        ps.setString(3, user.getPwd());
        ps.setInt(4, user.getRole());
        ps.executeUpdate();
        ps.close();
    }
```

动态代理机制

- 步骤四：视图层测试

```
public static void main(String[] args) {  
    StaffBiz biz = new StaffBiz();  
    TStaff staff = new TStaff();  
    staff.setSno("121000123");  
    staff.setName("李四");  
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");  
    try {  
        staff.setBirthday(sdf.parse("1995-10-1"));  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
    staff.setAddress("北京朝阳区建国门");  
    staff.setTel("13522454666");  
    TUser user = new TUser();  
    user.setSno("121000123");  
    user.setUname("jack");  
    user.setRole(2);  
    user.setPwd("123456");  
    try {  
        biz.addStaffUser(staff, user);  
        System.out.println(staff.getSno() + "创建成功....");  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

动态代理机制

- 步骤五：在addUser中主动抛出异常，测试事务回滚

```
public class UserDao {  
  
    /**  
     * 添加用户  
     * @param user  
     * @throws Exception  
     */  
    public void addUser(TUser user) throws Exception {  
        String sql = "insert into tuser values(?,?,?,?)";  
        Connection conn = DbFactory.openConnection();  
        PreparedStatement ps = conn.prepareStatement(sql);  
        ps.setString(1, user.getUname());  
        ps.setString(2, user.getSno());  
        ps.setString(3, user.getPwd());  
        ps.setInt(4, user.getRole());  
        ps.executeUpdate();  
        ps.close();  
        throw new RuntimeException("事务回滚测试...");  
    }  
}
```

动态代理机制

- StaffUser系统的事务处理案例之动态代理实现事务

- 步骤一：编写动态代理

```
public class TransacationProxy implements InvocationHandler {  
  
    private Object target;  
  
    public Object bind(Object target) {  
        this.target = target;  
        return Proxy.newProxyInstance(target.getClass().getClassLoader(),  
            target.getClass().getInterfaces(), this);  
    }  
}
```

```
@Override  
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
  
    Object result = null;  
  
    //打开数据库  
    DbFactory.openConnection();  
    //开启事务  
    DbFactory.beginTransaction();  
    try {  
        result = method.invoke(target, args);  
        //事务提交  
        DbFactory.commit();  
    } catch (Exception e) {  
        //事务回滚  
        DbFactory.rollback();  
        throw e; //二次抛出异常  
    } finally {  
        //关闭数据库  
        DbFactory.closeConnection();  
    }  
  
    return result;  
}
```

动态代理机制

- 步骤二：新增员工和用户的业务逻辑代码简化为自动控制

```
/**
 * 动态代理控制本地事务，保证员工和用户信息同时写入成功
 */
public void addStaffUser(TStaff staff, TUser user) throws Exception {
    Log.logger.info(Thread.currentThread().getId() + ":addStaffUser()");

    StaffDao staffDao = new StaffDao();
    UserDao userDao = new UserDao();
    staffDao.addStaff(staff);
    userDao.addUser(user);
}
```

动态代理机制

• 步骤三：代码测试

```
public static void main(String[] args) {
    TStaff staff = new TStaff();
    staff.setSno("121000152");
    staff.setName("马52");
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
    try {
        staff.setBirthday(sdf.parse("1995-10-1"));
    } catch (Exception e) {
    }
    staff.setAddress("北京朝阳区建国门");
    staff.setTel("13522454666");
    TUser user = new TUser();
    user.setSno("121000152");
    user.setUname("ma52");
    user.setRole(2);
    user.setPwd("1234");
    //动态代理绑定
    TransactionProxy proxy = new TransactionProxy();
    IStaff staffProxy = (IStaff)proxy.bind(new StaffBiz());
    try {
        staffProxy.addStaffUser(staff, user);
        System.out.println(staff.getSno() + "创建成功....");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

```
INFO - 1打开数据库,生成一个新连接.....
INFO - 1打开数据库,使用原有连接.....
INFO - 1开启事务
INFO - 1:addStaffUser()
INFO - 1:addStaff()
INFO - 1打开数据库,使用原有连接.....
INFO - 1:addUser()
INFO - 1打开数据库,使用原有连接.....
INFO - 1打开数据库,使用原有连接.....
INFO - 1提交事务
INFO - 1关闭数据库.....
121000152创建成功....
```

动态代理机制

• 步骤四：异常测试

```
public class UserDao {  
  
    /**  
     * 添加用户  
     * @param user  
     * @throws Exception  
     */  
    public void addUser(TUser user) throws Exception {  
        Log.logger.info(Thread.currentThread().getId() + ":addUser()");  
        String sql = "insert into tuser values(?,?,?,?)";  
        Connection conn = DbFactory.openConnection();  
        PreparedStatement ps = conn.prepareStatement(sql);  
        ps.setString(1, user.getUsername());  
        ps.setString(2, user.getSno());  
        ps.setString(3, user.getPwd());  
        ps.setInt(4, user.getRole());  
        ps.executeUpdate();  
        ps.close();  
        throw new RuntimeException("事务回滚测试...");  
    }  
}
```

INFO - 1打开数据库,生成一个新连接.....

INFO - 1打开数据库,使用原有连接.....

INFO - 1开启事务

INFO - 1:addStaffUser()

INFO - 1:addStaff()

INFO - 1打开数据库,使用原有连接.....

INFO - 1:addUser()

INFO - 1打开数据库,使用原有连接.....

INFO - 1打开数据库,使用原有连接.....

INFO - 1回滚事务

INFO - 1关闭数据库.....

动态代理机制

- StaffUser系统的事务处理案例之自定义注解@Transaction
 - 在StaffUser的业务操作中，addStaffUser()需要事务处理，而login()只需要数据库连接，不需要事务处理。如果所有的方法都做事务控制，对系统性能会有很大影响。
 - 如何识别一个业务方法，是否需要事务处理呢？
 - 可以采用非侵入式的XML配置，和注解方式识别。

动态代理机制

- 步骤一：新增注解

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Transaction {

}
```

动态代理机制

- 步骤二：使用注解

```
public interface IStaff {  
    /**  
     * 添加员工信息时，系统自动创建一个默认用户  
     * @param staff  
     * @param user  
     */  
    @Transaction  
    public void addStaffUser(Staff staff, User user) throws Exception;
```

动态代理机制

- 步骤三：TransactionProxy中增加注解判断

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
  
    Object result = null;  
    // 打开数据库  
    DbFactory.openConnection();  
    if (method.isAnnotationPresent(Transaction.class)) {  
        // 开启事务  
        DbFactory.beginTransaction();  
    }  
    try {  
        result = method.invoke(target, args);  
        if (method.isAnnotationPresent(Transaction.class)) {  
            // 事务提交  
            DbFactory.commit();  
        }  
    } catch (Exception e) {  
        if (method.isAnnotationPresent(Transaction.class)) {  
            // 事务回滚  
            DbFactory.rollback();  
        }  
        throw e; // 二次抛出异常  
    } finally {  
        // 关闭数据库  
        DbFactory.closeConnection();  
    }  
  
    return result;  
}
```

动态代理机制

- 步骤四：添加新员工测试

```
INFO - 1打开数据库,生成一个新连接.....
INFO - 1打开数据库,使用原有连接.....
INFO - 1开启事务
INFO - 1:addStaffUser()
INFO - 1:addStaff()
INFO - 1打开数据库,使用原有连接.....
INFO - 1:addUser()
INFO - 1打开数据库,使用原有连接.....
INFO - 1打开数据库,使用原有连接.....
INFO - 1提交事务
INFO - 1关闭数据库.....
121000154创建成功.....
```

动态代理机制

- 步骤五：用户登录测试

```
public static void main(String[] args) {  
    TransacationProxy proxy = new TransacationProxy();  
    IUser userProxy = (IUser) proxy.bind(new UserBiz());  
    try {  
        userProxy.login("tom", "123456");  
    } catch (Exception e) {  
    }  
}
```

INFO - 1打开数据库,生成一个新连接.....

INFO - 1:UserBiz-->>login()

INFO - 1:UserDao-->>login()

INFO - 1关闭数据库.....

动态代理机制

- StaffUser事务AspectJ方案之配置扫描信息

```
<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"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-4.3.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.3.xsd">

  <aop:aspectj-autoproxy />

  <context:component-scan
    base-package="com.icss.biz" />
```

动态代理机制

• StaffUser事务AspectJ方案之编写切面和通知

```
@Component
@Aspect
public class DbProxy {

    @Pointcut("execution(public * com.icss.biz.*.*(..))")
    private void businessOperate() {

    }

    @Before("businessOperate()")
    public void openDataBase(JoinPoint jp) throws Exception{
        Log.logger.info(Thread.currentThread().getId() + ":openDataBase..." + jp.toString());
        DbFactory.openConnection();
    }

    @After("businessOperate()")
    public void closeDataBase(JoinPoint jp) {
        Log.logger.info(Thread.currentThread().getId() + ":closeDataBase..." + jp.toString());
        DbFactory.closeConnection();
    }
}
```

```
@Around("businessOperate()")
public Object transaction(ProceedingJoinPoint pjp) throws Throwable{
    Object obj = null;

    Log.logger.info(pjp.toString());
    MethodSignature ms = (MethodSignature)pjp.getSignature();
    Method m = ms.getMethod();
    try {
        if(m.isAnnotationPresent(Transaction.class)) {
            DbFactory.beginTransaction(); //开启事务
        }
        obj = pjp.proceed(); //注意必须要有返回值
        if(m.isAnnotationPresent(Transaction.class)) {
            DbFactory.commit(); //提交事务
        }
    } catch (Throwable e) {
        if(m.isAnnotationPresent(Transaction.class)) {
            DbFactory.rollback(); //回滚事务
        }
        throw e; //必须要抛出异常
    }

    return obj;
}
```


动态代理机制

• StaffUser事务AspectJ方案之编写逻辑代码

```
public interface IStaff {  
    /**  
     * 添加员工信息时，系统自动创建一个默认用户  
     * @param staff  
     * @param user  
     */  
    @Transaction  
    public void addStaffUser(Staff staff, User user) throws Exception;
```

```
@Service  
public class StaffBiz implements IStaff{  
    /**  
     * 动态代理控制本地事务，保证员工和用户信息同时写入成功  
     */  
    public void addStaffUser(Staff staff, User user) throws Exception {  
        Log.logger.info(Thread.currentThread().getId() + ":addStaffUser()");  
  
        StaffDao staffDao = new StaffDao();  
        UserDao userDao = new UserDao();  
        staffDao.addStaff(staff);  
        userDao.addUser(user);  
    }  
}
```

动态代理机制

• StaffUser事务AspectJ方案之事务代码测试

```
public class TestAddStaffUser {  
    public static void main(String[] args) {  
        Staff staff = new Staff();  
        staff.setSno("121000155");  
        staff.setName("马55");  
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");  
        try {  
            staff.setBirthday(sdf.parse("1995-10-1"));  
        } catch (Exception e) {  
        }  
        staff.setAddress("北京朝阳区建国门");  
        staff.setTel("13522454666");  
        User user = new User();  
        user.setSno("121000155");  
        user.setUname("ma55");  
        user.setRole(2);  
        user.setPwd("1234");  
        //获得Spring的bean对象  
        IStaff staffProxy = (IStaff)BeanFactory.getBean(IStaff.class);  
        try {  
            staffProxy.addStaffUser(staff, user);  
            System.out.println(staff.getSno() + "创建成功....");  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
INFO - 1:打开数据库,生成一个新连接.....  
INFO - 1:开启事务  
INFO - 1:openDataBase...execution(void com.icss.biz.IStaff.addStaffUser(Staff,User))  
INFO - 1:打开数据库,使用原有连接.....  
INFO - 1:addStaffUser()  
INFO - 1:addStaff()  
INFO - 1:打开数据库,使用原有连接.....  
INFO - 1:addUser()  
INFO - 1:打开数据库,使用原有连接.....  
INFO - 1:打开数据库,使用原有连接.....  
INFO - 1:提交事务  
INFO - 1:closeDataBase...execution(void com.icss.biz.IStaff.addStaffUser(Staff,User))  
INFO - 1:关闭数据库.....  
121000155创建成功....
```

动态代理机制

• StaffUser事务AspectJ方案之事务异常测试

```
public void addUser(User user) throws Exception {  
    Log.logger.info(Thread.currentThread().getId() + ":addUser()");  
    String sql = "insert into tuser values(?,?,?,?)";  
    Connection conn = DbFactory.openConnection();  
    PreparedStatement ps = conn.prepareStatement(sql);  
    ps.setString(1, user.getUname());  
    ps.setString(2, user.getSno());  
    ps.setString(3, user.getPwd());  
    ps.setInt(4, user.getRole());  
    ps.executeUpdate();  
    ps.close();  
    throw new RuntimeException("事务回滚测试...");  
}
```

```
INFO - 1打开数据库,生成一个新连接.....  
INFO - 1开启事务  
INFO - 1:openDataBase...execution(void com.icss.biz.IStaff.addStaffUser(Staff,User))  
INFO - 1打开数据库,使用原有连接.....  
INFO - 1:addStaffUser()  
INFO - 1:addStaff()  
INFO - 1打开数据库,使用原有连接.....  
INFO - 1:addUser()  
INFO - 1打开数据库,使用原有连接.....  
INFO - 1打开数据库,使用原有连接.....  
INFO - 1回滚事务  
INFO - 1:closeDataBase...execution(void com.icss.biz.IStaff.addStaffUser(Staff,User))  
INFO - 1关闭数据库.....  
java.lang.RuntimeException: 事务回滚测试...  
    at com.icss.dao.UserDao.addUser(UserDao.java:37)  
    at com.icss.biz.impl.StaffBiz.addStaffUser(StaffBiz.java:25)  
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
```

动态代理机制

- StaffUser事务AspectJ方案之用户登录测试

```
public static void main(String[] args) {
```

```
    //获得Spring的bean对象
```

```
    IUser userProxy = (IUser)BeanFactory.getBean(IUser.class);
```

```
    try {
```

```
        userProxy.login("tom", "123456");
```

```
    } catch (Exception e) {
```

```
    }
```

```
}
```

INFO - 1:打开数据库,生成一个新连接.....

INFO - 1:UserBiz-->>login()

INFO - 1:UserDao-->>login()

INFO - 1:closeDataBase...execution(User com.icss.biz.IUser.login(String,String))

INFO - 1:关闭数据库.....

