Spring\_day02总结

今日内容

* AOP的概述
* AOP 的底层实现
* Spring 的AOP
* 使用AspectJ 实现AOP
* Spring JdbcTemplate 使用

## 上次课的内容回顾:

第一天:Spring的IOC.Spring就是一个大的工厂,通过工厂对Bean进行管理.

\* Spring的概述:

\* Spring的环境搭建:

\* Spring中的IOC:

\* IOC:控制反转.将对象的创建权交给Spring管理.

\* DI:依赖注入.在由Spring创建的对象的时候,将对象依赖的对象注入进来.

\* IOC的Bean装配(XML):

\* 定义类:

\* <bean id=”标识” class=”类的全路径”>

\* 其他的属性:

\* id和name:

\* id:满足XMLid约束语法,里面不能出现特殊的字符.

\* name:可以出现特殊的字符.

\*\*\*\*\* 如果配置<bean>标签的时候,如果只配置了name,就可以将这个name的值作为id.

\* scope:类的作用范围.

\* Bean的生命周期:

\* init-method:

\* destroy-method:

**\* Bean的完整生命周期:**

**\* 后处理Bean.增强**

\* 注入属性:

\* <property name=”” value=””>

\* <property name=”” ref=””>

\* IOC的Bean装配(注解):

\* 定义类:

\* 注入属性:

\* Spring框架与web整合.

\* Spring整合JUnit单元测试.

第二天Spring的AOP.JdbcTemplate.

## AOP的概述:

### 什么是AOP:

AOP Aspect Oriented Programing 面向切面编程

AOP采取横向抽取机制，取代了传统纵向继承体系重复性代码（性能监视、事务管理、安全检查、缓存）

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织入增强代码

AspecJ是一个基于Java语言的AOP框架，Spring2.0开始，Spring AOP引入对Aspect的支持，AspectJ扩展了Java语言，提供了一个专门的编译器，在编译时提供横向代码的织入

### AOP底层原理;

就是代理机制:

\* 动态代理:(JDK中使用)

\* JDK的动态代理,对实现了接口的类生成代理.

### Spring的AOP代理:

JDK动态代理 :对实现了接口的类生成代理

CGLib代理机制 :对类生成代理

### AOP的术语:

Joinpoint(连接点) :所谓连接点是指那些被拦截到的点。在spring中,这些点指的是方法,因为spring只支持方法类型的连接点.

Pointcut(切入点) :所谓切入点是指我们要对哪些Joinpoint进行拦截的定义.

Advice(通知/增强) :所谓通知是指拦截到Joinpoint之后所要做的事情就是通知.通知分为前置通知,后置通知,异常通知,最终通知,环绕通知(切面要完成的功能)

Introduction(引介):引介是一种特殊的通知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.

Target(目标对象) :代理的目标对象

Weaving(织入) :是指把增强应用到目标对象来创建新的代理对象的过程.

spring采用动态代理织入，而AspectJ采用编译期织入和类装在期织入

Proxy（代理） :一个类被AOP织入增强后，就产生一个结果代理类

Aspect(切面) :是切入点和通知（引介）的结合

## AOP的底层实现

### JDK动态代理:

public class JDKProxy implements InvocationHandler{

private UserDao userDao;

public JDKProxy(UserDao userDao) {

super();

this.userDao = userDao;

}

public UserDao createProxy() {

UserDao proxy = (UserDao) Proxy.newProxyInstance(userDao.getClass()

.getClassLoader(), userDao.getClass().getInterfaces(), this);

return proxy;

}

// 调用目标对象的任何一个方法 都相当于调用invoke();

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

throws Throwable {

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

// 记录日志:

System.out.println("日志记录=================");

Object result = method.invoke(userDao, args);

return result;

}

return method.invoke(userDao, args);

}

}

### CGLIB动态代理:

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。 Hibernate支持它来实现PO(Persistent Object 持久化对象)字节码的动态生成

Hibernate生成持久化类的javassist.

CGLIB生成代理机制:其实生成了一个真实对象的子类.

下载cglib的jar包.

\* 现在做cglib的开发,可以不用直接引入cglib的包.已经在spring的核心中集成cglib.

public class CGLibProxy implements MethodInterceptor{

private ProductDao productDao;

public CGLibProxy(ProductDao productDao) {

super();

this.productDao = productDao;

}

public ProductDao createProxy(){

// 使用CGLIB生成代理:

// 1.创建核心类:

Enhancer enhancer = new Enhancer();

// 2.为其设置父类:

enhancer.setSuperclass(productDao.getClass());

// 3.设置回调:

enhancer.setCallback(this);

// 4.创建代理:

return (ProductDao) enhancer.create();

}

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

MethodProxy methodProxy) throws Throwable {

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

System.out.println("日志记录==============");

Object obj = methodProxy.invokeSuper(proxy, args);

return obj;

}

return methodProxy.invokeSuper(proxy, args);

}

}

**结论:Spring框架,如果类实现了接口,就使用JDK的动态代理生成代理对象,如果这个类没有实现任何接口,使用CGLIB生成代理对象.**

## Spring中的AOP

### Spring的传统AOP :

AOP:不是由Spring定义.AOP联盟的组织定义.

Spring中的通知:(增强代码)

前置通知 org.springframework.aop.MethodBeforeAdvice

\* 在目标方法执行前实施增强

后置通知 org.springframework.aop.AfterReturningAdvice

\* 在目标方法执行后实施增强

环绕通知 org.aopalliance.intercept.MethodInterceptor

\* 在目标方法执行前后实施增强

异常抛出通知 org.springframework.aop.ThrowsAdvice

\* 在方法抛出异常后实施增强

引介通知 org.springframework.aop.IntroductionInterceptor(课程不讲.)

\* 在目标类中添加一些新的方法和属性

### Spring中的切面类型:

Advisor : Spring中传统切面.

\* Advisor :都是有一个切点和一个通知组合.

\* Aspect :多个切点和多个通知组合.

Advisor : 代表一般切面，Advice本身就是一个切面，对目标类所有方法进行拦截(\* 不带有切点的切面.针对所有方法进行拦截)

PointcutAdvisor : 代表具有切点的切面，可以指定拦截目标类哪些方法(带有切点的切面,针对某个方法进行拦截)

IntroductionAdvisor : 代表引介切面，针对引介通知而使用切面（不要求掌握）

### Spring的AOP的开发:

#### 针对所有方法的增强:(不带有切点的切面)

第一步:导入相应jar包.

\* spring-aop-3.2.0.RELEASE.jar

\* com.springsource.org.aopalliance-1.0.0.jar

第二步:编写被代理对象:

\* CustomerDao接口

\* CustoemrDaoImpl实现类

第三步:编写增强的代码:

public class MyBeforeAdvice implements MethodBeforeAdvice{

/\*\*

\* method:执行的方法

\* args:参数

\* target:目标对象

\*/

public void before(Method method, Object[] args, Object target)

throws Throwable {

System.out.println("前置增强...");

}

}

第四步:生成代理:(配置生成代理:)

\* 生成代理Spring基于ProxyFactoryBean类.底层自动选择使用JDK的动态代理还是CGLIB的代理.

\* 属性:

target : 代理的目标对象

proxyInterfaces : 代理要实现的接口

如果多个接口可以使用以下格式赋值

<list>

<value></value>

....

</list>

proxyTargetClass : 是否对类代理而不是接口，设置为true时，使用CGLib代理

interceptorNames : 需要织入目标的Advice

singleton : 返回代理是否为单实例，默认为单例

optimize : 当设置为true时，强制使用CGLib

<!-- 定义目标对象 -->

<bean id="customerDao" class="cn.itcast.spring3.demo3.CustomerDaoImpl"></bean>

<!-- 定义增强 -->

<bean id="beforeAdvice" class="cn.itcast.spring3.demo3.MyBeforeAdvice"></bean>

<!-- Spring支持配置生成代理: -->

<bean id="customerDaoProxy" class="org.springframework.aop.framework.ProxyFactoryBean">

<!-- 设置目标对象 -->

<property name="target" ref="customerDao"/>

<!-- 设置实现的接口 ,value中写接口的全路径 -->

<property name="proxyInterfaces" value="cn.itcast.spring3.demo3.CustomerDao"/>

<!-- 需要使用value : 要拦截的名称 -->

<property name="interceptorNames" value="beforeAdvice"/>

</bean>

**\*\*\*\*\* 注入的时候要注入代理对象:**

@Autowired

// @Qualifier("customerDao")// 注入是真实的对象,必须注入代理对象.

@Qualifier("**customerDaoProxy**")

private CustomerDao customerDao;

#### 带有切点的切面:(针对目标对象的某些方法进行增强)

PointcutAdvisor 接口:

DefaultPointcutAdvisor 最常用的切面类型，它可以通过任意Pointcut和Advice 组合定义切面

**RegexpMethodPointcutAdvisor** 构造正则表达式切点切面

第一步:创建被代理对象.

\* OrderDao

第二步:编写增强的类:

public class MyAroundAdvice implements MethodInterceptor{

public Object invoke(MethodInvocation methodInvocation) throws Throwable {

System.out.println("环绕前增强...");

Object result = methodInvocation.proceed();// 执行目标对象的方法

System.out.println("环绕后增强...");

return result;

}

}

第三步:生成代理:

<!-- 带有切点的切面 -->

<!-- 定义目标对象 -->

<bean id="orderDao" class="cn.itcast.spring3.demo4.OrderDao"></bean>

<!-- 定义增强 -->

<bean id="aroundAdvice" class="cn.itcast.spring3.demo4.MyAroundAdvice"></bean>

<!-- 定义切点切面: -->

<bean id="myPointcutAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">

<!-- 定义表达式,规定哪些方法执行拦截 -->

<!-- . 任意字符 \* 任意个 -->

<!-- <property name="pattern" value=".\*"/> -->

<!-- <property name="pattern" value="cn\.itcast\.spring3\.demo4\.OrderDao\.add.\*"/> -->

<!-- <property name="pattern" value=".\*add.\*"></property> -->

<property name="patterns" value=".\*add.\*,.\*find.\*"></property>

<!-- 应用增强 -->

<property name="advice" ref="aroundAdvice"/>

</bean>

<!-- 定义生成代理对象 -->

<bean id="orderDaoProxy" class="org.springframework.aop.framework.ProxyFactoryBean">

<!-- 配置目标 -->

<property name="target" ref="orderDao"></property>

<!-- 针对类的代理 -->

<property name="proxyTargetClass" value="true"></property>

<!-- 在目标上应用增强 -->

<property name="interceptorNames" value="myPointcutAdvisor"></property>

</bean>

### 自动代理:

前面的案例中，每个代理都是通过ProxyFactoryBean织入切面代理，在实际开发中，非常多的Bean每个都配置ProxyFactoryBean开发维护量巨大

自动创建代理(\*\*\*\*\*基于后处理Bean.在Bean创建的过程中完成的增强.生成Bean就是代理.)

BeanNameAutoProxyCreator 根据Bean名称创建代理

DefaultAdvisorAutoProxyCreator 根据Advisor本身包含信息创建代理

\* AnnotationAwareAspectJAutoProxyCreator 基于Bean中的AspectJ 注解进行自动代理

#### BeanNameAutoProxyCreator :按名称生成代理

<!-- 定义目标对象 -->

<bean id="customerDao" class="cn.itcast.spring3.demo3.CustomerDaoImpl"></bean>

<bean id="orderDao" class="cn.itcast.spring3.demo4.OrderDao"></bean>

<!-- 定义增强 -->

<bean id="beforeAdvice" class="cn.itcast.spring3.demo3.MyBeforeAdvice"></bean>

<bean id="aroundAdvice" class="cn.itcast.spring3.demo4.MyAroundAdvice"></bean>

<!-- 自动代理:按名称的代理 基于后处理Bean,后处理Bean不需要配置ID-->

**<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">**

**<property name="beanNames" value="\*Dao"/>**

**<property name="interceptorNames" value="beforeAdvice"/>**

**</bean>**

#### DefaultAdvisorAutoProxyCreator :根据切面中定义的信息生成代理

<!-- 定义目标对象 -->

<bean id="customerDao" class="cn.itcast.spring3.demo3.CustomerDaoImpl"></bean>

<bean id="orderDao" class="cn.itcast.spring3.demo4.OrderDao"></bean>

<!-- 定义增强 -->

<bean id="beforeAdvice" class="cn.itcast.spring3.demo3.MyBeforeAdvice"></bean>

<bean id="aroundAdvice" class="cn.itcast.spring3.demo4.MyAroundAdvice"></bean>

**<!-- 定义一个带有切点的切面 -->**

**<bean id="myPointcutAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">**

**<property name="pattern" value=".\*add.\*"/>**

**<property name="advice" ref="aroundAdvice"/>**

**</bean>**

**<!-- 自动生成代理 -->**

**<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"></bean>**

**区分基于ProxyFattoryBean的代理与自动代理区别?**

**\*\*\*\*\* ProxyFactoryBean:先有被代理对象,将被代理对象传入到代理类中生成代理.**

**自动代理基于后处理Bean.在Bean的生成过程中,就产生了代理对象,把代理对象返回.生成Bean已经是代理对象.**

## Spring的AspectJ的AOP(\*\*\*\*\*)

AspectJ是一个面向切面的框架，它扩展了Java语言。AspectJ定义了AOP语法所以它有一个专门的[编译器](http://baike.baidu.com/view/487018.htm" \t "http://baike.baidu.com/_blank)用来生成遵守Java字节编码规范的Class文件。

AspectJ是一个基于Java语言的AOP框架

Spring2.0以后新增了对AspectJ切点表达式支持

@AspectJ 是AspectJ1.5新增功能，通过JDK5注解技术，允许直接在Bean类中定义切面

新版本Spring框架，建议使用AspectJ方式来开发AOP

AspectJ表达式:

\* 语法:execution(表达式)

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

\* execution(“\* cn.itcast.spring3.demo1.dao.\*(..)”) ---只检索当前包

\* execution(“\* cn.itcast.spring3.demo1.dao..\*(..)”) ---检索包及当前包的子包.

\* execution(\* cn.itcast.dao.GenericDAO+.\*(..)) ---检索GenericDAO及子类

AspectJ增强:

@Before 前置通知，相当于BeforeAdvice

@AfterReturning 后置通知，相当于AfterReturningAdvice

@Around 环绕通知，相当于MethodInterceptor

@AfterThrowing抛出通知，相当于ThrowAdvice

@After 最终final通知，不管是否异常，该通知都会执行

@DeclareParents 引介通知，相当于IntroductionInterceptor (不要求掌握)

### 基于注解:

第一步:引入相应jar包.

\* aspectj依赖aop环境.

\* spring-aspects-3.2.0.RELEASE.jar

\* com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar

第二步:编写被增强的类:

\* UserDao

第三步:使用AspectJ注解形式:

@Aspect

public class MyAspect {

@Before("execution(\* cn.itcast.spring3.demo1.UserDao.add(..))")

public void before(){

System.out.println("前置增强....");

}

}

第四步:创建applicationContext.xml

\* 引入aop的约束:

\* <aop:aspectj-autoproxy /> --- 自动生成代理:

\* 底层就是AnnotationAwareAspectJAutoProxyCreator

<aop:aspectj-autoproxy />

<bean id="userDao" class="cn.itcast.spring3.demo1.UserDao"></bean>

<bean id="myAspect" class="cn.itcast.spring3.demo1.MyAspect"></bean>

#### AspectJ的通知类型:

@Before 前置通知，相当于BeforeAdvice

\* 就在方法之前执行.没有办法阻止目标方法执行的.

@AfterReturning 后置通知，相当于AfterReturningAdvice

\* 后置通知,获得方法返回值.

@Around 环绕通知，相当于MethodInterceptor

\* 在可以方法之前和之后来执行的,而且可以阻止目标方法的执行.

@AfterThrowing抛出通知，相当于ThrowAdvice

@After 最终final通知，不管是否异常，该通知都会执行

@DeclareParents 引介通知，相当于IntroductionInterceptor (不要求掌握)

#### 切点的定义:

@Pointcut("execution(\* cn.itcast.spring3.demo1.UserDao.find(..))")

private void myPointcut(){}

再次使用该切点

@Before("myPointcut()")

public void update(User user) {

.....

}

面试:

\* Advisor和Aspect的区别?

\* Advisor :Spring传统意义上的切面:支持一个切点和一个通知的组合.

\* Aspect :可以支持多个切点和多个通知的组合.

### 基于XML:

第一步:编写被增强的类:

\* ProductDao

第二步:定义切面

第三步:配置applicationContext.xmll

前置通知:

\* 代码:

public void before(){

System.out.println("前置通知...");

}

\* 配置:

<aop:config>

<!-- 定义切点: -->

<aop:pointcut expression="execution(\* cn.itcast.spring3.demo2.ProductDao.add(..))" id="mypointcut"/>

<aop:aspect ref="myAspectXML">

<!-- 前置通知 -->

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

</aop:aspect>

</aop:config>

后置通知:

\* 代码:

public void afterReturing(Object returnVal){

System.out.println("后置通知...返回值:"+returnVal);

}

\* 配置:

<aop:config>

<!-- 定义切点: -->

<aop:pointcut expression="execution(\* cn.itcast.spring3.demo2.ProductDao.add(..))" id="mypointcut"/>

<aop:aspect ref="myAspectXML">

<!-- 后置通知 -->

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

</aop:aspect>

</aop:config>

环绕通知:

\* 代码:

public Object around(ProceedingJoinPoint proceedingJoinPoint) throws Throwable{

System.out.println("环绕前增强....");

Object result = proceedingJoinPoint.proceed();

System.out.println("环绕后增强....");

return result;

}

\* 配置:

<aop:config>

<!-- 定义切点: -->

<aop:pointcut expression="execution(\* cn.itcast.spring3.demo2.ProductDao.add(..))" id="mypointcut"/>

<aop:aspect ref="myAspectXML">

<!-- 前置通知 -->

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

<!-- 后置通知 -->

<!-- <aop:after-returning method="afterReturing" pointcut-ref="mypointcut" returning="returnVal"/> -->

<!-- 环绕通知 -->

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

</aop:aspect>

</aop:config>

异常通知:

\* 代码;

public void afterThrowing(Throwable e){

System.out.println("异常通知..."+e.getMessage());

}

\* 配置;

<aop:config>

<!-- 定义切点: -->

<aop:pointcut expression="execution(\* cn.itcast.spring3.demo2.ProductDao.add(..))" id="mypointcut"/>

<aop:aspect ref="myAspectXML">

<!-- 异常通知 -->

<aop:after-throwing method="afterThrowing" pointcut-ref="mypointcut" throwing="e"/>

</aop:aspect>

</aop:config>

最终通知:

\* 代码:

public void after(){

System.out.println("最终通知....");

}

\* 配置:

<aop:config>

<!-- 定义切点: -->

<aop:pointcut expression="execution(\* cn.itcast.spring3.demo2.ProductDao.add(..))" id="mypointcut"/>

<aop:aspect ref="myAspectXML">

<!-- 最终通知 -->

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

</aop:aspect>

</aop:config>

## Spring的JdbcTemplate

JdbcTemplate模板与DbUtils工具类比较类似.

### Spring对持久层技术支持:

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

### 开发JDBCTemplate入门:

第一步:引入相应jar包:

\* spring-tx-3.2.0.RELEASE.jar

\* spring-jdbc-3.2.0.RELEASE.jar

\* mysql驱动.

第二步:创建applicationContext.xml

第三步:编写一个测试类:

### 配置连接池:

#### Spring默认的连接池:

<!-- 配置Spring默认的连接池 -->

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

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

<property name="url" value="jdbc:mysql:///spring3\_day02"/>

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

<property name="password" value="123"/>

</bean>

#### DBCP连接池:

导入jar包:

\* com.springsource.org.apache.commons.dbcp-1.2.2.osgi.jar

\* com.springsource.org.apache.commons.pool-1.5.3.jar

<!-- 配置DBCP连接池 -->

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">

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

<property name="url" value="jdbc:mysql:///spring3\_day02"/>

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

<property name="password" value="123"/>

</bean>

#### C3P0连接池:

导入jar包:

\* com.springsource.com.mchange.v2.c3p0-0.9.1.2.jar

<!-- 配置c3p0连接池 -->

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

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

<property name="jdbcUrl" value="jdbc:mysql:///spring3\_day02"/>

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

<property name="password" value="123"/>

</bean>

### 参数设置到属性文件中:

在src下创建jdbc.properties

jdbc.driver = com.mysql.jdbc.Driver

jdbc.url = jdbc:mysql:///spring3\_day02

jdbc.user = root

jdbc.password = 123

需要在applicationContext.xml 中使用属性文件配置的内容.

\* 第一种写法:

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<property name="location" value="classpath:jdbc.properties"></property>

</bean>

\* 第二种写法:

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

### JdbcTemplate的CRUD的操作:

Spring框架中提供了对持久层技术支持的类:

JDBC :org.springframework.jdbc.core.support.JdbcDaoSupport

Hibernate 3.0 :org.springframework.orm.hibernate3.support.HibernateDaoSupport

iBatis :org.springframework.orm.ibatis.support.SqlMapClientDaoSupport

编写DAO的时候:

Public class UserDao extends **JdbcDaoSupport**{

}

进行CRUD的操作;

\* 保存:update(String sql,Object... args)

\* 修改:update(String sql,Object... args)

\* 删除:update(String sql,Object... args)

查询:

\* 简单查询:

\* select count(\*) from user; --- queryForInt(String sql);

\* select name from user where id = ?; --- queryForObject(String sql,Class clazz,Object... args);

\* 复杂查询:(返回对象,和对象集合)

\* select \* from user where id = ? --- queryForObjectString sql,RowMapper<T> rowMapper,Object... args);

\* select \* from user; --- query(String sql,RowMapper<T> rowMapper,Object... args);

今天的内容总结:

Spring AOP

\* AOP:面向切面编程.不是替代面向对象,对面向对象扩展,延伸.

\* AOP的思想:

\* 代理机制.(横向抽取).

\* Spring使用AOP的时候,根据类的情况,自动选择代理机制:

\* JDK :针对实现了接口的类生成代理.

\* CGLIB :针对没有实现接口的类生成代理.生成某个类的子类.

\* Spring的AOP的开发中:

\* 术语 :

\* JoinPoint:连接点.可以被增强的方法.

\* PointCut:切入点.真正被增强的方法.

\* Advice:通知.增强的代码.

\* Target:目标对象.被增强的类.

\* Weaving:

\* Aspect:切面.在切入点应用增强.

\* 传统的Spring AOP.

\* 不带切点的切面:

\* 带有切点的切面:

\* 生成代理:基于ProxyFactoryBean.

**\* 缺点;每个类都需要配置ProxyFactroyBean**.

\* 自动代理:(基于后处理Bean)

\* Bean名称自动代理:

\* 切面信息自动代理:

**\* Spring中AspectJ的支持.**

\* 为了简便开发引入AspectJ的支持.

\* 注解:

\* @Aspect:

\* @Before

\* @Around

\* @AfterReturing

\* @AfterThrowing

\* @After

\* PointCut

\* XML:

\* 引入aop约束:

<aop-config>

<aop:pointcut id=”” expression=””/>

<aop:aspect ref=””>

<aop:before />

</aop:aspect>

</aop-config>

Spring JDBCTemplate:

\* 配置连接池:

\* 默认的:

\* DBCP:

\* C3P0:(\*\*\*\*\*)

\* 提取了properties

\* JdbcTemplate的CRUD的操作.