共性问题：

1. 服务器启动报错，什么原因？

\* jar包缺少、jar包冲突

1) 先检查项目中是否缺少jar包引用

2) 服务器: 检查jar包有没有发布到服务器下；

**用户库jar包，需要手动发布到tomcat. (每次新建项目)**

3) 重新发布项目

\* 配置文件错误

(web.xml / struts.xml /bean.xml /hibernate.xml / \*.hbm.xml)

明确的提示

\* 端口占用

\* webapps项目过多

当前项目没有问题，有可能是其他项目出错，这样启动也会报错！但这个错误不是当前错误报的，所以不影响当前项目运行.

注意：

一般开发中，一个tomcat下只有一个项目。

2. 一般启动报错： ClassNotFoundException异常

少jar包。

3. 访问404，什么原因？

1）客户端路径写错，或跳转错误。

2）启动报错

Web项目中，启动后一定要看下，是否报错。

4. 点击某个功能报错。

后台代码错误：

🡪 找到页面，点击哪个操作，提交到哪个地址

--》后台在提交地址的第一行打断点

Spring提供了一站式解决方案：

1） Spring Core spring的核心功能： IOC容器, 解决对象创建及依赖关系

2） Spring Web Spring对web模块的支持。

-🡪 可以与struts整合,让struts的action创建交给spring

-🡪 spring mvc模式

***3） Spring DAO Spring 对jdbc操作的支持 【JdbcTemplate模板工具类】***

4） Spring ORM spring对orm的支持：

🡪 既可以与hibernate整合，【session】

🡪 也可以使用spring的对hibernate操作的封装

***5）Spring AOP 切面编程***

6）SpringEE spring 对javaEE其他模块的支持

目标：

1） 代理模式

静态代理

动态代理

Cglib代理

2）手动实现AOP编程 【代理模式】

3）AOP编程

\* 注解方式实现

\* XMl配置方式实现

4） *Spring DAO*

*Spring对jdbc操作的支持*

# 1. 代理模式

## 1.1 概述

代理（Proxy）是一种设计模式， 提供了对目标对象另外的访问方式；即通过代理访问目标对象。 这样好处： 可以在目标对象实现的基础上，增强额外的功能操作。(扩展目标对象的功能)。

举例：明星（邓紫棋）🡨---经纪人<-------用户

目标 （代理）



代理模式的关键点： 代理对象与目标对象。

## 1.2 静态代理

静态代理，

1） 代理对象，要实现与目标对象一样的接口；

2） 举例:

保存用户(模拟)

Dao , 直接保存

DaoProxy, 给保存方法添加事务处理

|  |
| --- |
| package cn.itcast.a\_static;  public class App {  public static void main(String[] args) {  // 目标对象  IUserDao target = new UserDao();    // 代理  IUserDao proxy = new UserDaoProxy(target); 目标对象传入代理对象 执行代理方法  proxy.save(); // 执行的是，代理的方法  }  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*代理对象(静态代理)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\* 代理对象，要实现与目标对象一样的接口 \*/  public class UserDaoProxy implements IUserDao{  // 接收保存目标对象  private IUserDao target;  public UserDaoProxy(IUserDao target) {  this.target = target;  }    @Override  public void save() {  System.out.println("开始事务...");    target.save(); // 执行目标对象的方法    System.out.println("提交事务...");  }  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*目标对象\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  public class UserDao implements IUserDao{  @Override  public void save() {  System.out.println("-----已经保存数据！！！------");  }  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*都要实现的接口\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  package cn.itcast.a\_static;  // 接口  public interface IUserDao {  void save();  } |

总结静态代理：

1）可以做到在不修改目标对象的功能前提下，对目标对象功能扩展。

2）缺点：

--》 因为代理对象，需要与目标对象实现一样的接口。所以会有很多代理类，类太多。

--》 一旦接口增加方法，目标对象与代理对象都要维护。

解决：

代理工厂？ 可以使用动态代理。

## 1.3 动态代理

动态代理，

1）代理对象，不需要实现接口；

2）代理对象的生成，是利用JDKAPI， 动态的在内存中构建代理对象(需要我们指定创建 代理对象/目标对象 实现的接口的类型；);

3) 动态代理， JDK代理， 接口代理；

JDK中生成代理对象的API：

|-- Proxy

static Object newProxyInstance(

ClassLoader loader, 指定当前目标对象使用类加载器

Class<?>[] interfaces, 目标对象实现的接口的类型

InvocationHandler h 事件处理器

)

|  |
| --- |
| **public** **class** App {  **public** **static** **void** main(String[] args) {  // 目标对象  IUserDao target = **new** UserDao();  // 【原始的类型 class cn.itcast.b\_dynamic.UserDao】  System.***out***.println(target.getClass());    // 给目标对象，创建代理对象  IUserDao proxy = (IUserDao) **new** ProxyFactory(target).getProxyInstance();  // class $Proxy0 内存中动态生成的代理对象  System.***out***.println(proxy.getClass());    // 执行方法 【代理对象】  proxy.save();  }  }  /\*\*给所有的dao创建代理对象【动态代理】\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\*\* 代理对象，不需要实现接口\*\*/  **public** **class** ProxyFactory {  // 维护一个目标对象  **private** Object target;  **public** ProxyFactory(Object target){  **this**.target = target;  }    // 给目标对象，生成代理对象  **public** Object getProxyInstance() {  **return**  Proxy.newProxyInstance(  //目标对象类的加载  target.getClass().getClassLoader(),  //目标对象实现的接口类型  target.getClass().getInterfaces(),    //事件处理器  **new** InvocationHandler() {  @Override  **public** Object invoke(Object proxy, Method method, Object[] args)  **throws** Throwable {  System.***out***.println("开启事务");    // 执行目标对象方法  Object returnValue = method.invoke(target, args);    System.***out***.println("提交事务");  **return** returnValue;  }  });    }  } |

动态代理总结：

代理对象不需要实现接口，但是目标对象一定要实现接口；否则不能用动态代理！  
 (class $Proxy0 implements IuserDao)

思考：

有一个目标对象，想要功能扩展，但目标对象没有实现接口，怎样功能扩展？

Class UserDao{}

// 子类的方式

Class subclass extends UserDao{}

以子类的方式实现(cglib代理)

## 1.4 Cglib代理

Cglib代理，也叫做子类代理。在内存中构建一个子类对象从而实现对目标对象功能的扩展。

* **JDK的动态代理有一个限制，就是使用动态代理的对象必须实现一个或多个接口。如果想代理没有实现接口的类，就可以使用CGLIB实现。**
* **CGLIB是一个强大的高性能的代码生成包，它可以在运行期扩展Java类与实现Java接口。它广泛的被许多AOP的框架使用，例如Spring AOP和dynaop，为他们提供方法的interception（拦截）。**
* **CGLIB包的底层是通过使用一个小而快的字节码处理框架ASM，来转换字节码并生成新的类。不鼓励直接使用ASM，因为它要求你必须对JVM内部结构包括class文件的格式和指令集都很熟悉。**

Cglib子类代理：

1) 需要引入cglib – jar文件， 但是spring的核心包中已经包括了cglib功能，所以直接引入spring-core-3.2.5.jar即可。

2）引入功能包后，就可以在内存中动态构建子类

3）代理的类不能为final， 否则报错。

4） 目标对象的方法如果为final/static, 那么就不会被拦截，即不会执行目标对象额外的业务方法。

在Spring的AOP编程中，

如果加入容器的目标对象有实现接口，用JDK代理；

如果目标对象没有实现接口，用Cglib代理；

# 2. 手动实现AOP编程

AOP 面向切面的编程，

AOP可以实现“业务代码”与“关注点代码”分离

|  |
| --- |
| **// 保存一个用户**  **public void add(User user) {**  **Session session = null;**  **Transaction trans = null;**  **try {**  **session = HibernateSessionFactoryUtils.getSession(); // 【关注点代码】**  **trans = session.beginTransaction(); // 【关注点代码】**    **session.save(user); // 核心业务代码**    **trans.commit(); //…【关注点代码】**  **} catch (Exception e) {**  **e.printStackTrace();**  **if(trans != null){**  **trans.rollback(); //..【关注点代码】**  **}**  **} finally{**  **HibernateSessionFactoryUtils.closeSession(session); ////..【关注点代码】**  **}**  **}** |

分析总结：

关注点代码，就是指重复执行的代码。

**业务代码与关注点代码分离，好处？**

--🡪 关注点代码写一次即可；

-🡪开发者只需要关注核心业务；

-🡪运行时期，执行核心业务代码时候动态植入关注点代码； 【代理】

如何分离？

过程式/对象式/代理模式分离

# 3. AOP编程

## 3.1 概述：

**Aop，** aspect object programming 面向切面编程

功能： 让关注点代码与业务代码分离！

**关注点,**

重复代码就叫做关注点；

**切面，**

关注点形成的类，就叫切面(类)！

面向切面编程，就是指 对很多功能都有的重复的代码抽取，再在运行的时候网业务方法上动态植入“切面类代码”。

**切入点，**

执行目标对象方法，动态植入切面代码。

可以通过切入点表达式，指定拦截哪些类的哪些方法； 给指定的类在运行的时候植入切面类代码。

AOP原理 ： 用动态代理 实现

## 3.2 注解方式实现AOP编程

步骤：

1） 先引入aop相关jar文件 （aspectj aop优秀组件）

spring-aop-3.2.5.RELEASE.jar 【spring3.2源码】

aopalliance.jar 【spring2.5源码/lib/aopalliance】

aspectjweaver.jar 【spring2.5源码/lib/aspectj】或【aspectj-1.8.2\lib】

aspectjrt.jar 【spring2.5源码/lib/aspectj】或【aspectj-1.8.2\lib】

注意： 用到spring2.5版本的jar文件，如果用jdk1.7可能会有问题。

需要升级aspectj组件，即使用aspectj-1.8.2版本中提供jar文件提供。

2） bean.xml中引入aop名称空间

3） 开启aop注解

4) 使用注解 （）

@Aspect 指定一个类为切面类

@Pointcut("execution(\* cn.itcast.e\_aop\_anno.\*.\*(..))") 指定切入点表达式

@Before("pointCut\_()") 前置通知: 目标方法之前执行

@After("pointCut\_()") 后置通知：目标方法之后执行（始终执行）

@AfterReturning("pointCut\_()") 返回后通知： 执行方法结束前执行(异常不执行)

@AfterThrowing("pointCut\_()") 异常通知: 出现异常时候执行

@Around("pointCut\_()") 环绕通知： 环绕目标方法执行

|  |
| --- |
| **1. IUserDao.java** |
| // 接口  **public** **interface** IUserDao {  **void** save();  } |
| **2. UserDao.java** |
| /\*\*  \* 目标对象  \* **@author** Jie.Yuan  \*  \*/  @Component // 加入容器  **public** **class** UserDao **implements** IUserDao{  @Override  **public** **void** save() {  System.*out*.println("-----核心业务：保存！！！------");  }  } |
| **3. Aop.java 切面类** |
| @Component //注意这一步一定要有  @Aspect // 指定当前类为切面类  **public** **class** Aop {  // 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  所有包 所有方法  @Pointcut("execution(\* cn.itcast.e\_aop\_anno.\*.\*(..))")  **public** **void** pointCut\_(){  }    // 前置通知 : 在执行目标方法之前执行  @Before("pointCut\_()")  **public** **void** begin(){  System.*out*.println("开始事务/异常");  }    // 后置/最终通知：在执行目标方法之后执行 【无论是否出现异常最终都会执行】  @After("pointCut\_()")  **public** **void** after(){  System.*out*.println("提交事务/关闭");  }    // 返回后通知： 在调用目标方法结束后执行 【出现异常不执行】  @AfterReturning("pointCut\_()")  **public** **void** afterReturning() {  System.*out*.println("afterReturning()");  }    // 异常通知： 当目标方法执行异常时候执行此关注点代码  @AfterThrowing("pointCut\_()")  **public** **void** afterThrowing(){  System.*out*.println("afterThrowing()");  }    // 环绕通知：环绕目标方式执行  @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....");  }    } |
| **4. bean.xml** |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:p=*"http://www.springframework.org/schema/p"*  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"*>    <!-- 开启注解扫描 -->  <context:component-scan base-package=*"cn.itcast.e\_aop\_anno"*></context:component-scan>    <!-- 开启aop注解方式 -->  <aop:aspectj-autoproxy></aop:aspectj-autoproxy>  </beans> |

|  |
| --- |
| App.java |
| **public** **class** App {    ApplicationContext ac =  **new** ClassPathXmlApplicationContext("cn/itcast/e\_aop\_anno/bean.xml");  // 目标对象有实现接口，spring会自动选择“JDK代理”  @Test  **public** **void** testApp() {  IUserDao userDao = (IUserDao) ac.getBean("userDao");  System.*out*.println(userDao.getClass());  userDao.save();  }    // 目标对象没有实现接口， spring会用“cglib代理”  @Test  **public** **void** testCglib() {  OrderDao orderDao = (OrderDao) ac.getBean("orderDao");  System.*out*.println(orderDao.getClass());  orderDao.save();  }  } |

## 3.3 XML方式实现AOP编程

Xml实现aop编程：

1） 引入jar文件 【aop 相关jar， 4个】

2） 引入aop名称空间

3）aop 配置

\* 配置切面类 （重复执行代码形成的类）

\* aop配置

拦截哪些方法 / 拦截到方法后应用通知代码

|  |
| --- |
| <?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:p=*"http://www.springframework.org/schema/p"*  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"*>    <!-- dao 实例 -->  <bean id=*"userDao"* class=*"cn.itcast.f\_aop\_xml.UserDao"*></bean>  <bean id=*"orderDao"* class=*"cn.itcast.f\_aop\_xml.OrderDao"*></bean>    <!-- 切面类 -->  <bean id=*"aop"* class=*"cn.itcast.f\_aop\_xml.Aop"*></bean>    <!-- Aop配置 -->  <aop:config>  <!-- 定义一个切入点表达式： 拦截哪些方法 -->  <aop:pointcut expression=*"execution(\* cn.itcast.f\_aop\_xml.\*.\*(..))"* id=*"pt"*/>  <!-- 切面 -->  <aop:aspect ref=*"aop"*>  <!-- 前置通知： 在目标方法调用前执行 -->  <aop:before method=*"begin"* pointcut-ref=*"pt"*/>  <!-- 后置通知： -->  <aop:after method=*"after"* pointcut-ref=*"pt"*/>  <!-- 返回后通知 -->  <aop:after-returning method=*"afterReturning"* pointcut-ref=*"pt"*/>  <!-- 异常通知 -->  <aop:after-throwing method=*"afterThrowing"* pointcut-ref=*"pt"*/>  <!-- 环绕通知 -->  <aop:around method=*"around"* pointcut-ref=*"pt"*/>  </aop:aspect>  </aop:config>  </beans> |

## 3.4 切入点表达式

切入点表达式,

可以对指定的“方法”进行拦截； 从而给指定的方法所在的类生成代理对象。

|  |
| --- |
| <?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:p=*"http://www.springframework.org/schema/p"*  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"*>    <!-- dao 实例 -->  <bean id=*"userDao"* class=*"cn.itcast.g\_pointcut.UserDao"*></bean>  <bean id=*"orderDao"* class=*"cn.itcast.g\_pointcut.OrderDao"*></bean>    <!-- 切面类 -->  <bean id=*"aop"* class=*"cn.itcast.g\_pointcut.Aop"*></bean>    <!-- Aop配置 -->  <aop:config>    <!-- 定义一个切入点表达式： 拦截哪些方法 第一个\* 是返回值 （固定要有）-->  <!--<aop:pointcut expression="execution(\* cn.itcast.g\_pointcut.\*.\*(..))" id="pt"/>-->    <!-- 【拦截所有public方法】 -->  <!--<aop:pointcut expression="execution(public \* \*(..))" id="pt"/>-->    <!-- 【拦截所有save开头的方法 】 -->  <!--<aop:pointcut expression="execution(\* save\*(..))" id="pt"/>-->    <!-- 【拦截指定类的指定方法, 拦截时候一定要定位到方法】 -->  <!--<aop:pointcut expression="execution(public \* cn.itcast.g\_pointcut.OrderDao.save(..))" id="pt"/>-->    <!-- 【拦截指定类的所有方法】 -->  <!--<aop:pointcut expression="execution(\* cn.itcast.g\_pointcut.UserDao.\*(..))" id="pt"/>-->    <!-- 【拦截指定包，以及其自包下所有类的所有方法】 -->  <!--<aop:pointcut expression="execution(\* cn..\*.\*(..))" id="pt"/>-->    <!-- 【多个表达式】 -->  <!--<aop:pointcut expression="execution(\* cn.itcast.g\_pointcut.UserDao.save()) || execution(\* cn.itcast.g\_pointcut.OrderDao.save())" id="pt"/>-->  <!--<aop:pointcut expression="execution(\* cn.itcast.g\_pointcut.UserDao.save()) or execution(\* cn.itcast.g\_pointcut.OrderDao.save())" id="pt"/>-->  <!-- 下面2个且关系的，没有意义 -->  <!--<aop:pointcut expression="execution(\* cn.itcast.g\_pointcut.UserDao.save()) &amp;&amp; execution(\* cn.itcast.g\_pointcut.OrderDao.save())" id="pt"/>-->  <!--<aop:pointcut expression="execution(\* cn.itcast.g\_pointcut.UserDao.save()) and execution(\* cn.itcast.g\_pointcut.OrderDao.save())" id="pt"/>-->    <!-- 【取非值】 -->  <!--<aop:pointcut expression="!execution(\* cn.itcast.g\_pointcut.OrderDao.save())" id="pt"/>-->  <aop:pointcut expression=*" not execution(\* cn.itcast.g\_pointcut.OrderDao.save())"* id=*"pt"*/>    <!-- 切面 -->  <aop:aspect ref=*"aop"*>  <!-- 环绕通知 -->  <aop:around method=*"around"* pointcut-ref=*"pt"*/>  </aop:aspect>  </aop:config>  </beans> |

# 4. Spring对jdbc支持

使用步骤：

1）引入jar文件

spring-jdbc-3.2.5.RELEASE.jar

spring-tx-3.2.5.RELEASE.jar

2） 优化