框架概述：

Struts2: 基于mvc模式的应用层框架！

主要是作为控制层组件，javabean， jsp！

Hibernate: 持久层组件，简化jdbc操作！

Spring: 主要有六大模块功能

事务管理！

与其他框架整合！

Mvc开发模式：

基于mvc模式的项目分层!

# 1. Spring框架

1. 框架理解

框架，都会预先实现一些功能，给开发人员开发提供便利！

提高开发效率、提升程序可扩展性、健壮！

1. 专业术语

(1)高内聚、低耦合：

类内部的关系越紧密越好，类与类的关系越少越好！

(2)非侵入式设计：

1)侵入式设计：

引入的组件对现有的类的结构会有影响，这种就是“侵入式设计”的组件！

Struts2: 侵入式设计！

2)非侵入式设计：

引入组件对现有的类的结构没有影响！

Hibernate: 非侵入式设计！

Spring, 非侵入式设计

(3)IOC 容器:Inversion of control 控制反转

1)控制反转容器! -🡪 对象创建的问题！

2)解释：

User user = new User(); 自己控制对象的创建

现在需要对象，自己不创建，交给外部的容器创建，叫控制反转！

IOC容器= bean.xml配置 + ApplicationContext容器类

3) 例子

public class Girl {

private Boy boy =new Boy();

public void kiss() {

System.out.println(boy.getName());

}

}

boy是在应用内部创建及维护的。控制反转就是应用本身不负责依赖对象的创建及维护，依赖对象的创建及维护是由外部容器负责的。控制权就由应用转移到了外部容器，控制权的转移就是所谓反转，目的是为了获得更好的扩展性和良好的可维护性

(4)DI, dependency injection 依赖注入

1）创建对象后，处理对象的依赖关系！

User user = new User();

user.setAddress(..) ;// 需要DI（依赖注入）

2）当我们把依赖对象交给外部容器负责创建，那么Girl类可以改成如下：

public class Girl {

private Boy boy;

public void kiss() {

System.out.println(boy.getName());

}

}

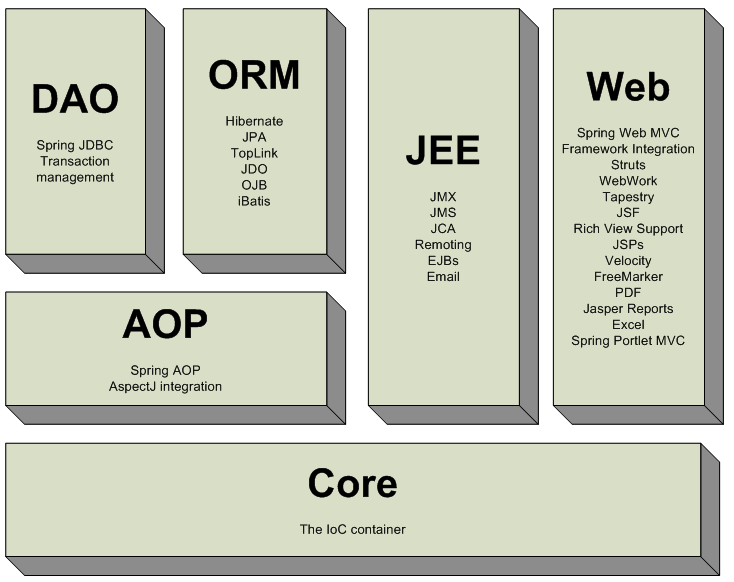
依赖注入：在运行期，由外部容器动态地将依赖对象注入到另一个对象中。

(5)Aop…

1. Spring框架

Spring 开源框架，提供的是一站式解决方案！

(1)六大模块:



(2) spring好处:

* 降低组件之间的耦合度,实现软件各层之间的解耦。

1)IOC(依赖注入)：包含并管理应用对象的配置和生命周期，可配置每个bean如何被创建，也可配置每个bean是只有一个实例，还是每次需要时都生成一个新的实例，以及它们是如何相互关联的

2)AOP(面向切面)：采用了面向切面编程来实现很多基础但是与业务逻辑无关的功能的解耦,比如：事务管理、日志、权限验证.....

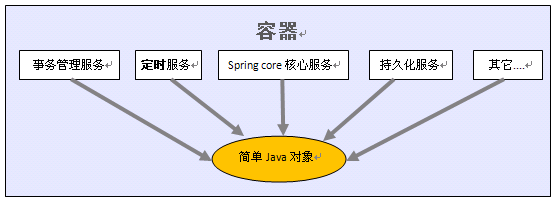
3)DAO:Spring 还提供了对数据库JDBC的封装,使用JdbcTemplate来简化数据操作

4)ORM：提供了对主流的对象关系映射框架的支持

5)JEE: 对Java企业级开发提供了一些解决方案,例如EJB、JMS等

6)WEB: 提供了自己的Spring MVC和对显示层框架的集合支持

* 可使用容器提供的众多服务



1. Spring 第一个案例， 解决项目中如何创建对象

(1) 引入jar文件 (3.2版本)

commons-logging-1.1.3.jar 【单独下载】

spring-beans-3.2.5.RELEASE.jar 【spring源码, bean节点管理】

spring-context-3.2.5.RELEASE.jar 【spring上下文类】

spring-core-3.2.5.RELEASE.jar 【IOC容器】

spring-expression-3.2.5.RELEASE.jar 【spring表达式】

注意：使用的版本Spring3.2

在这个版本中，只有spring自身的核心功能，spring依赖的其他组件，需要单独下载！ 例如：日志jar文件，就需要单独下载！

(2) 新建applicationContext.xml , 源码中拷贝约束

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  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">  </beans> |

(3) 配置

|  |
| --- |
| <!-- 创建Dao实例 -->  <bean id="userDao" class="cn.itcast.dao.UserDao"></bean>  <!-- 创建Service实例 -->  <bean id="userService" class="cn.itcast.service.UserService">  <property name="userDao" ref="userDao"></property>  </bean>    <!-- 创建Action实例 -->  <bean id="userAction" class="cn.itcast.action.UserAction">  <property name="userService" ref="userService"></property>  </bean> |

(4) UserDao.java

|  |
| --- |
| package cn.itcast.dao;  public class UserDao {  public void save() {  System.out.println("UserDao.save()");  }  } |

(5) UserService.java

|  |
| --- |
| package cn.itcast.service;  import cn.itcast.dao.UserDao;  public class UserService {  // 创建dao对象 : 单例，启动创建  private UserDao userDao;  public void setUserDao(UserDao userDao) {  this.userDao = userDao;  }  public void save() {  userDao.save();  }  } |

(6) UserAction.java

|  |
| --- |
| package cn.itcast.action;  import cn.itcast.service.UserService;  /\*\*  \* 多例Action实例：在访问的时候创建对象！  \* @heyutong  \* 2017年6月2日  \*/  public class UserAction {  // 判断一个类是单例还是多例，主要看有没有维护成员变量、且对象成员变量进行修  改！ 如果有，这个类就应该是多例！  // 创建Service:单例，启动时候创建  private UserService userService;//= new UserService();  // 提供set方法，给外部容器注入  public void setUserService(UserService userService) {  this.userService = userService;  }    public String execute() {  userService.save();  return "success";  }  } |

(4) 测试

|  |
| --- |
| package junit.test;  import org.junit.Test;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import cn.itcast.action.UserAction;  public class ActionTest {  //不从IOC获取对象  //报空指针异常  /\*@Test  public void test1() {  UserAction userAction = new UserAction();  userAction.execute();  }\*/    //从IOC容器获取对象  @Test  public void test2() {  // 容器对象(加载applicationContext.xml配置文件)  //调用了setUserDao和setUserService方法  ClassPathXmlApplicationContext ac = new  ClassPathXmlApplicationContext("applicationContext.xml");  // 获取对象  UserAction userAction = (UserAction)ac.getBean("userAction");  userAction.execute();  }  } |

# 2. Spring IOC 容器

## 2.1创建bean细节：

1. bean在容器中的写法

2. bean声明周期

|  |
| --- |
| <?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:context=*"http://www.springframework.org/schema/context"*  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"*>  <!-- 把对象加入IOC容器 -->  <!--  细节1：  id 与name：  id 不能以数字开头，不能含有特殊符号, 不能有空格、逗号等； id 不能重复！  name 可以数字开头，可以有特殊符号，如果name值重复，编译没问题但运行报错！  <bean id="user" name="user2,user3 user4" class="cn.itcast.a\_config.User"></bean>  <bean name="user5" class="cn.itcast.a\_config.User"></bean>  <bean id="user6" class="cn.itcast.a\_config.User"></bean>  <bean id="user6" class="cn.itcast.a\_config.User"></bean>  -->    <!-- 细节2： (单例/多例) -->  <!--  scope="singleton" 默认表示单例！  prototype 多例  init-method="" 在创建完对象之后执行初始化方法  destroy-method="" 在调用容器类的destroy()方法时候，对单例的对象有效！  lazy-init="true" 延迟初始化/设置只对单例有影响，对多例没有任何影响！  单例对象默认是在创建容器时创建所有单例对象，如希望在第一次访问时创建单例  对象，就设置延迟初始化  Bean生命周期：  bean在xml中配置，  1.singleton 单例  1) 创建对象  如果有配置延迟初始化，  lazy-init=true 如单例对象有配置延迟初始化， 在创建容器之后，在第一次从容器  获取对象时创建单例的对象！  如没有配置或延迟初始化为默认值, 单例对象会在创建容器时创建对象  2) 执行初始化方法 , init-method配置的方法会执行  3) 调用容器destroy() 方法时，容器在销毁单例对象实例时，会调用destroy-method  对应的方法 此时bean对象会被销毁！    2.prototype 多例  1) 每次从容器获取对象时，都会创建新的对象  2) 每次创建完对象后，就执行初始化方法  3) java回回收不用资源(jvm gc)  -->  <bean id=*"user"*  class=*"cn.itcast.a\_config.User"*  init-method=*"init"*  destroy-method=*"destroy\_"*  lazy-init=*"false"*  ~~scope=~~*~~"prototype"~~*>  </bean>  </beans> |

3. User.java

|  |
| --- |
| package cn.itcast.a\_config;  public class User {  private int id;  private String name;    public User() {  System.out.println("User.User(), 创建User对象...");  }  public int getId() {  return id;  }  public void setId(int id) {  this.id = id;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }    //初始化时候调用  public void init() {  System.out.println("初始化...");  }    //销毁对象实例时候调用  public void destroy\_() {  System.out.println("销毁...");  }  } |

4. App.java

|  |
| --- |
| package cn.itcast.a\_config;  import org.junit.Test;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class App {  @Test  public void test() {  ClassPathXmlApplicationContext ac = new  ClassPathXmlApplicationContext("cn/itcast/a\_config/bean.xml");  System.out.println("----------");    //获取/创建对象  User user = (User) ac.getBean("user");  System.out.println(user);  User user2 = (User) ac.getBean("user");  System.out.println(user2);  // 销毁容器实例  ac.destroy();  }  } |

1. 运行测试结果:
2. 单例：未配置scope=*"prototype"*
3. new ClassPathXmlApplicationContext("cn/itcast/a\_config/bean.xml");时，打印:

User.User(), 创建User对象... //new User对象

初始化... //执行init-method="init"

2)user 和user2 对象一样，例如都是cn.itcast.a\_config.User@1147d1a

3) 销毁容器实例,即执行ac.destroy();时执行destroy-method="destroy\_",打印:

销毁...

1. 多例: 配置scope=*"prototype"*
2. User user = (User) ac.getBean("user");时,打印:

|  |
| --- |
| User.User(), 创建User对象... //创建一个User对象  初始化... //执行init-method="init"  cn.itcast.a\_config.User@111cccd |

1. User user2 = (User) ac.getBean("user");时,打印:

|  |
| --- |
| User.User(), 创建User对象... //创建另一个User对象  初始化... //执行init-method="init"  cn.itcast.a\_config.User@18441c1 |

3) 销毁容器实例,即执行ac.destroy();时不执行destroy-method="destroy\_",

## 2.2创建对象的几种方式

**2.2.1.调用无参数构造器**

**2.2.2.调用有参数构造器**

1.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:context="http://www.springframework.org/schema/context"  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">  <!--IOC容器， 创建对象 -->  <!—1) 调用无参数构造器 -->  <bean id="user1" class="cn.itcast.b\_create\_obj.User"></bean>    <!—2) 调用有参数构造器, 创建对象 -->  <!-- String str = new String("Jack"); -->  <bean id="str" class="java.lang.String">  <constructor-arg value="Jack"></constructor-arg>  </bean>    <!-- 创建user对象，且赋值 -->  <!--  value 当直接给属性值的时候使用value赋值  ref 当引用的是IOC容器中的对象的时候，使用ref  -->  <bean id="user" class="cn.itcast.b\_create\_obj.User">  <constructor-arg index="0" type="int" value="10000"></constructor-arg>  <constructor-arg index="1" type="String" ref="str"></constructor-arg>  </bean>  </beans> |

**2.** User.java

|  |
| --- |
| package cn.itcast.b\_create\_obj;  public class User {  private int id;  private String name;    public User() {  System.out.println("User.User(), 创建User对象...");  }  public User(int id, String name) {  super();  this.id = id;  this.name = name;  }  public int getId() {  return id;  }  public void setId(int id) {  this.id = id;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  @Override  public String toString() {  return "User [id=" + id + ", name=" + name + "]";  }    // 初始化时候调用  public void init() {  System.out.println("初始化...");  }    // 销毁对象实例时候调用  public void destroy\_() {  System.out.println("销毁...");  }  } |

**3.** App.java

|  |
| --- |
| package cn.itcast.b\_create\_obj;  import org.junit.Test;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class App {  @Test  public void test() {  // 方式1：创建容器对象  /\*ClassPathXmlApplicationContext ac = new ClassPathXmlApplicationContext("cn/itcast/b\_create\_obj/bean.xml");//打印:User.User(), 创建User对象...  User user = (User) ac.getBean("user1");  System.out.println(user);\*/    // 方式2： 会从当前类所在的包下找bean.xml 【测试方便】  /\*ClassPathXmlApplicationContext ac = new  ClassPathXmlApplicationContext("bean.xml", this.getClass());  //打印:User.User(), 创建User对象...  String str = (String)ac.getBean("str");  System.out.println(str);//打印:Jack\*/    // 方式2： 会从当前类所在的包下找bean.xml 【测试方便】  ClassPathXmlApplicationContext ac = new  ClassPathXmlApplicationContext("bean.xml", this.getClass());  //打印:User.User(), 创建User对象...  User user = (User) ac.getBean("user");  System.out.println(user);//打印:User [id=10000, name=Jack]  }  } |

**2.2.3. 工厂**

1. 静态方法:<bean id=”’ class=”” factory-method=””/>

(1)nean2.xml

|  |
| --- |
| <!-- \* 工厂静态方法,创建对象 -->  <!--  class 指定工厂的类型；  factory-method: 工厂类的静态方法  -->  <bean id="user1" class="cn.itcast.b\_create\_obj.UserFactory"  factory-method="getStaticInstace"></bean> |

(2) App.java

|  |
| --- |
| package cn.itcast.b\_create\_obj;  import org.junit.Test;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class App {  @Test  public void test() {  //方式2： 会从当前类所在的包下找bean2.xml 【测试方便】  //打印:(1)调用getStaticInstace方法 (2)调用User(int id, String name)构造方法  ClassPathXmlApplicationContext ac = new ClassPathXmlApplicationContext("bean2.xml", this.getClass());  User user = (User) ac.getBean("user1");  System.out.println(user);//User [id=101, name=工厂静态方法，创建对象]  }  } |

(3) UserFactory.java

|  |
| --- |
| package cn.itcast.b\_create\_obj;  // 创建user对象的工厂  public class UserFactory {  // 非静态  public User getInstace() {  return new User(100,"工厂实例方法，创建对象");  }    // 静态  public static User getStaticInstace() {  System.out.println("调用getStaticInstace方法");  return new User(101,"工厂静态方法，创建对象");  }  } |

2. 非静态方法:

<bean id=”factory” class=”..”>

<bean id=”” factory-bean=” factory” factory-method=”实例方法” />

(1)nean2.xml

|  |
| --- |
| <!-- \* 非静态方法创建对象 -->  <!-- 先创建工厂实例 -->  <bean id="factory" class="cn.itcast.b\_create\_obj.UserFactory"></bean>  <bean id="user" factory-bean="factory" factory-method="getInstace"></bean> |

(2)

|  |
| --- |
| package cn.itcast.b\_create\_obj;  import org.junit.Test;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class App {  @Test  public void test() {  //方式2： 会从当前类所在的包下找bean2.xml 【测试方便】  //打印:(1)调用getInstace方法 (2)调用User(int id, String name)构造方法  ClassPathXmlApplicationContext ac = new  ClassPathXmlApplicationContext("bean2.xml", this.getClass());  User user = (User) ac.getBean("user");  System.out.println(user);//打印:User [id=100, name=工厂实例方法，创建对象]  }  } |

**2.24. 反射**

## 2.3 处理对象的依赖关系

给对象属性赋值(DI, 依赖注入)方式：

**2.3.1构造函数赋值**

**2.3.2 set方法注入值**

\* 普通字段赋值

\* 集合属性 (list/map/property)

1. 例1

(1)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:context="http://www.springframework.org/schema/context"  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">  <!--IOC容器， 给对象属性赋值 -->  <bean id="user" class="cn.itcast.c\_di.User">  <property name="id" value="1000"></property>  <property name="name" value="Jacky"></property>  <!-- list集合赋值 -->  <property name="list">  <list>  <value>cn</value>  <value>usa</value>  </list>  </property>  <!-- map 集合赋值 -->  <property name="map">  <map>  <entry key="cn" value="China"></entry>  <entry key="usa" value="1234"></entry>  </map>  </property>  <!-- Properties对象赋值 -->  <property name="prop">  <props>  <prop key="cn">China</prop>  <prop key="usa">America</prop>  </props>  </property>  </bean>  </beans> |

(2) User.java

|  |
| --- |
| package cn.itcast.c\_di;  import java.util.List;  import java.util.Map;  import java.util.Properties;  public class User {  private int id;  private String name;  // list集合  private List<String> list;  // Map集合  private Map<String,Object> map;  // Properties 对象  private Properties prop;  public int getId() {  return id;  }  public void setId(int id) {  this.id = id;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public List<String> getList() {  return list;  }  public void setList(List<String> list) {  this.list = list;  }  public Map<String, Object> getMap() {  return map;  }  public void setMap(Map<String, Object> map) {  this.map = map;  }  public Properties getProp() {  return prop;  }  public void setProp(Properties prop) {  this.prop = prop;  }  @Override  public String toString() {  return "User [id=" + id + ", list=" + list + ", map=" + map + ", name="  + name + ", prop=" + prop + "]";  }  } |

(3) App.java

|  |
| --- |
| package cn.itcast.c\_di;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class App {  @Test  public void testApp() throws Exception {  // 方式2： 会从当前类所在的包下找bean.xml 【测试方便】  //调用了setId、setName、setList、setMap、setProp方法  ApplicationContext ac = new  ClassPathXmlApplicationContext("bean.xml",this.getClass());  User user = (User) ac.getBean("user");  //打印:User [id=1000, list=[cn, usa],  //map={cn=China, usa=1234}, name=Jacky, prop={usa=America, cn=China}]  System.out.println(user);  }  } |

2.例2

(1)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"  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">  <!-- 配置： 需要创建的对象、对象依赖关系处理 -->  <!-- 方式1：通过set方法注入，这种方式最常用！-->  <bean id="userdao" class="cn.itcast.d\_di2.UserDao"></bean>  <bean id="userService" class="cn.itcast.d\_di2.UserService">  <property name="userDao" ref="userdao"></property>  </bean>  <!-- 多例 -->  <bean id="userAction" class="cn.itcast.d\_di2.UserAction" scope="prototype">  <property name="userService" ref="userService"></property>  </bean>  </beans> |

(2) UserDao.java

|  |
| --- |
| package cn.itcast.d\_di2;  public class UserDao implements IUserDao {  @Override  public void save() {  System.out.println("UserDao.save()");  }  } |

IUserDao.java

|  |
| --- |
| package cn.itcast.d\_di2;  public interface IUserDao {  // 模拟：保存对象  //接口中的方法都是public abstract,平时习惯abstract不写,public也不写  public abstract void save();  } |

(3) IUserService.java

|  |
| --- |
| package cn.itcast.d\_di2;  public interface IUserService {  public abstract void save();  } |

UserService.java

|  |
| --- |
| package cn.itcast.d\_di2;  public class UserService implements IUserService{  private UserDao userDao;  // 接收IOC容器注入值  public void setUserDao(UserDao userDao) {  this.userDao = userDao;  }  @Override  public void save() {  userDao.save();  }  } |

(4) UserAction.java

|  |
| --- |
| package cn.itcast.d\_di2;  public class UserAction {  private IUserService userService;  //接收IOC容器注入值  public void setUserService(IUserService userService) {  this.userService = userService;  }  public String execute(){  userService.save();  return "success";  }  } |

(5)

|  |
| --- |
| package cn.itcast.d\_di2;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class App {  private ApplicationContext ac = new  ClassPathXmlApplicationContext("bean.xml",App.class);  @Test  public void testApp() throws Exception {  //从容器中获取Action实例  UserAction userAction = (UserAction) ac.getBean("userAction");  System.out.println(userAction);//cn.itcast.d\_di2.UserAction@898802  // 执行方法  userAction.execute();//打印:UserDao.save()  System.out.println("----------------");  userAction =  (UserAction)ac.getBean("userAction");//cn.itcast.d\_di2.UserAction@1405b61  System.out.println(userAction);  userAction.execute();//打印:UserDao.save()  }  } |

**2.3.3内部bean**

(1)bean.xml

|  |
| --- |
| <!-- 方式2： 内部bean -->  <!-- 总结： 当创建的对象，不被容器其他地方引用的时候，可以这样写！ 这样写不通用，内部对象只能用一次 -->  <!--action如是单例，内部bean默认是单例; action如是多例，内部bean就是多例 -->  <bean id="userAction" class="cn.itcast.d\_di2.UserAction" scope="prototype">  <property name="userService">  <bean class="cn.itcast.d\_di2.UserService">  <property name="userDao">  <bean class="cn.itcast.d\_di2.UserDao"></bean>  </property>  </bean>  </property>  </bean> |

(2)其它同2.3.2节-例2

**2.3.4 p名称空间**

|  |
| --- |
| <!-- 方式3： p名称空间，给属性注入值 -->  <bean id="userDao" class="cn.itcast.d\_di2.UserDao"></bean>  <bean id="userService" class="cn.itcast.d\_di2.UserService" p:userDao-ref="userDao"></bean>  <bean id="userAction" class="cn.itcast.d\_di2.UserAction"  p:userService-ref="userService"></bean> |

**2.3.5自动装配**

1.(1)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"  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"  default-autowire="byType">  <!-- 配置： 需要创建的对象、对象依赖关系处理 -->  <!-- 方式4：自动装配 (了解) -->  <!--  a. default-autowire="byType" 配置到全局  当前所有的bean都采用”根据类型自动装配“  b. 配置到bean节点  autowire="byName" 根据名称自动装配， 会去容器找指定名称的对象，  注入到set方法的参数中！  autowire="byType" 根据类型自动装配, 要确保改类型对应的对象在IOC  容器中唯一，否则报错！  总结：简化配置，但是维护麻烦！默认配置是no，推荐用这种  -->  <bean id="userDao" class="cn.itcast.e\_autowire.UserDao"></bean>  <bean id="userService" class="cn.itcast.e\_autowire.UserService"></bean>  <bean id="userAction" class="cn.itcast.e\_autowire.UserAction"></bean>  </beans> |

(2)其它同2.3.2节-例2

2. default-autowire自动装配类型

自动装配可使配置与java代码同步更新。例如，如果你需要给一个java类增加一个依赖，那么该依赖将被自动实现而不需要修改配置。因此强烈推荐在开发过程中采用自动装配，而在系统趋于稳定的时候改为显式装配的方式。

(1)no 默认.必须显示使用"<ref />"标签明确地指定bean合作者，对于部署给予更大的

控制和明了。

(2)byName 根据属性名自动装配。此选项将检查容器并根据名字查找与属性完全一致的bean，并将其与属性自动装配。若变量与id名称不一样，就报错。

例如，在bean定义中将 autowire设置为by name，而该bean包含master属性（同时提供setMaster(..)方法），Spring就会查找名为master的bean定义，并用它来装配给master属性。

|  |  |  |
| --- | --- | --- |
| 例:   |  | | --- | | Service.java  public class Service  {  Source source;  public void setSource(Source source)  {  this.source = source;  }  } |  |  | | --- | | applicationContext.xml  <beans  ...  default-autowire="byName">  <bean id="source" class="cn.hh.spring.DBCPSource" scope="prototype"/>  <bean id="service" class="cn.hh.spring.Service" scope="prototype">  </bean>  </beans> |   applicationContext.xml中并没给bean id="service"配Source属性，但在beans中设置了autowire="byName".配置文件会自动根据 cn.hh.spring.Service 中的setSource找bean id="Source"的bean ，然后自动配上去，如果没找到就不装配。  注意：byName的name是java中setXxxx 的Xxxx, 和上面设置的Source source中source拼写毫无关系，完全可以是  public class Service  {  Source source1;  public void setSource(Source source1)  {  this.source1 = source1;  }  }  结果相同。 |

1. byType 如果容器中存在一个与指定属性类型相同的bean，那么将与该属性自动装配。

如存在多个该类型的bean，将会抛出异常，并指出不能使用byType方式进行自动装配。

若没有找到相匹配的bean，则什么事都不发生，属性也不会被设置。

如不希望这样，可通过设置dependency-check="objects"让Spring抛出异常。

若变量与id不匹配，也没关系

(4)constructor 与byType的方式类似，不同之处在于它应用于构造器参数。如果在容器中没有找到与构造器参数类

型一致的bean，那么将会抛出异常。

(5)autodetect 通过bean类的自省机制（introspection）来决定是使用constructor还是byType方式进行自动装配。

如果发现默认的构造器，那么将使用byType方式。

**2.3.6注解**

2.3.6.1 @Repository、@Service、@Controller 和 @Component 将类标识为Bean

自 spring 2.0 引入注解。

1. @Repository

将数据访问层(DAO 层)的类标识为 Spring Bean。同时，为了让 Spring 能够扫描类路径中的类并识别出 @Repository 注解，需在 XML 配置文件中启用Bean 的自动扫描功能，可通过<context:component-scan/>实现：

|  |
| --- |
| //首先使用 @Repository 将 DAO 类声明为 Bean  package bookstore.dao;  @Repository  public class UserDaoImpl implements UserDao{ …… }  // 其次，在 XML 配置文件中启动 Spring 的自动扫描功能  <beans … >  ……  <context:component-scan base-package=”bookstore.dao” />  ……  </beans> |

就不再需要在 XML 中显式使用 <bean/> 进行Bean 的配置。Spring 在容器初始化时将自动扫描 base-package 指定的包及其子包下的所有 class文件，所有标注了 @Repository 的类都将被注册为 Spring Bean。

@Repository只能标注在DAO 类上:因为该注解的作用不只是将类识别为Bean,还能将所标注的类中抛出的数据访问异常封装为 Spring 的数据访问异常类型。Spring本身提供了一个丰富的并且是与具体的数据访问技术无关的数据访问异常结构，用于封装不同的持久层框架抛出的异常，使得异常独立于底层的框架。

1. @Component :泛化的概念，仅表示一个组件 (Bean) ，可作用在任何层次。
2. @Service 常作用在业务层，但目前该功能与 @Component 相同。
3. @Constroller 常作用在控制层，但目前该功能与 @Component 相同。

通过在类上使用 @Repository、@Component、@Service 和 @Constroller 注解，Spring会自动创建相应的 BeanDefinition 对象，并注册到 ApplicationContext 中。这些类就成了 Spring受管组件。这三个注解除了作用于不同软件层次的类，其使用方式与 @Repository 是完全相同的。

5.另外,用户可以创建自定义注解，然后在注解上标注@Component，那么,该自定义注解便具有了与所@Component 相同的功能。不过这个功能并不常用。

当一个 Bean 被自动检测到时，会根据那个扫描器的 BeanNameGenerator 策略生成它的 bean名称。默认情况下，对于包含 name 属性的 @Component、@Repository、 @Service 和@Controller，会把 name 取值作为 Bean 的名字。如果这个注解不包含 name值或是其他被自定义过滤器发现的组件，默认 Bean 名称会是小写开头的非限定类名。如果你不想使用默认 bean命名策略，可以提供一个自定义的命名策略。首先实现 BeanNameGenerator接口，确认包含了一个默认的无参数构造方法。然后在配置扫描器时提供一个全限定类名，如下所示：

|  |
| --- |
| <beans ...>  <context:component-scan  base-package="a.b" name-generator="a.SimpleNameGenerator"/>  </beans> |

与通过 XML 配置的 Spring Bean 一样，通过上述注解标识的Bean，其默认作用域是"singleton"，为了配合这四个注解，在标注 Bean 的同时能够指定 Bean 的作用域，Spring2.5 引入了 @Scope 注解。使用该注解时只需提供作用域的名称就行了，如下所示：

|  |
| --- |
| @Scope("prototype")  @Repository  public class Demo { … } |

如果你想提供一个自定义的作用域解析策略而不使用基于注解的方法，只需实现 ScopeMetadataResolver接口，确认包含一个默认的没有参数的构造方法。然后在配置扫描器时提供全限定类名：

|  |
| --- |
| <context:component-scan base-package="a.b"  scope-resolver="footmark.SimpleScopeResolver" /> |

6.@Autowired与@Resource的区别

(1)都可用来装配bean. 都可写在字段上,或写在setter方法上。

(2)@Autowired默认按类型装配（属于spring的），默认要求依赖对象必须存在，如要允许null 值，可设置@Autowired(required=false) ，如想使用名称装配可以结合@Qualifier注解进行使用，如下：

|  |
| --- |
| @Autowired() @Qualifier("baseDao")  private BaseDao baseDao; |

(3)@Resource(属于J2EE)，默认按名称进行装配，名称可通过name属性进行指定，

如没有指定name属性，当注解写在字段上时，默认取字段名进行按照名称查找，如果注解写在setter方法上默认取属性名进行装配。 当找不到与名称匹配的bean时才按照类型进行装配。但是需要注意的是，如果name属性一旦指定，就只会按照名称进行装配。

总结:name指定，只按名称;

~未指定，先按名称，再按类型

|  |
| --- |
| @Resource(name="baseDao")  private BaseDao baseDao; |

减少了与spring的耦合

2.3.6.2 例子

1.bean.xml

|  |
| --- |
| <!-- 注解的方式，实现springIOC容器配置 -->  <!-- 1. 开启注解扫描 -->  <context:component-scan base-package="cn.itcast.f\_anno"></context:component-scan>  <bean class="cn.itcast.f\_anno.UserDao"></bean> |

2. IUserDao.java

|  |
| --- |
| package cn.itcast.f\_anno;  public interface IUserDao {  // 模拟：保存对象  public abstract void save();  } |

UserDao.java

|  |
| --- |
| package cn.itcast.f\_anno;  import org.springframework.stereotype.Component;  import org.springframework.stereotype.Repository;  // 把userDao对象加入IOC容器  //@Component("userdao") // <bean id=userDao class="...">  //@Repository("userdao") // <bean id=userDao class="...">  @Repository //默认名称是类名，且第一个字母小写  public class UserDao implements IUserDao {  // 模拟：保存对象  public void save() {  System.out.println("UserDao.save()");  }  } |

3. IUserService.java

|  |
| --- |
| package cn.itcast.f\_anno;  public interface IUserService {  public abstract void save();  } |

UserService.java

|  |
| --- |
| package cn.itcast.f\_anno;  import javax.annotation.Resource;  import org.springframework.stereotype.Component;  import org.springframework.stereotype.Service;  //@Component("userService") // 把当前类加入IOC容器  //@Service("userService") // 把当前类加入IOC容器  @Service  public class UserService implements IUserService {  //@Resource(name = "userDao") //根据”userDao“去ioc容器找对象，  //找到后注入到当前方法参数  @Resource // 默认会根据private IUserDao userDao 这里的名称去容器找； 如果没有  //找到，再根据类型找，再没有找到就报错！  private IUserDao userDao;  public void save() {  userDao.save();  }  } |

4. UserAction.java

|  |
| --- |
| package cn.itcast.f\_anno;  import javax.annotation.Resource;  import org.springframework.context.annotation.Scope;  import org.springframework.stereotype.Controller;  //@Component("userAction") // 加入IOC容器  //@Controller("userAction") // 加入IOC容器  @Scope("prototype") // 指定加入ioc容器的对象为多例  @Controller  public class UserAction {  // @Resource(name = "userService") // ioc容器找名称是userService的对象并注入  @Resource  private IUserService userService;    public String execute(){  userService.save();  return "success";  }  } |

3.案例

Dao/service/action实例，处理依赖关系

\* 常用的通过set方法注入

\* 内部bean

\* p名称空间

\* 自动装配

\* 注解

注解总结：

@Component 表示一个组件(类)，把当前组件加入ioc容器

加入容器的组件的名称默认是类名第一个字母小写

@Component(“”) 指定加入ioc容器的组件类的类名

@Repository 标识是一个持久层的组件

@Service 标识是一个业务逻辑层的组件

@Controller 标识是一个控制层的组件

@Scope("prototype") 指定对象单例/多例

@Resource 1. 默认根据修饰的字段名称会取ioc容器找对象自动注入

找到后注入

2. 如果名称没有找到，再根据类型查找 找到后就立刻注入

如果改类型在ioc容器中有多个对象，报错！

3. 根据类型也没有找到对象，报错！

@Resource(name =””) 会根据指定的名称去容器找对象自动注入

配置方式与注解方式:

1. 配置， 便于维护

(配置过多，比较繁琐)

2. 注解， 开发方便

(简化配置，不利于后期维护，如果修改对象创建、关系处理，需要改代码！)

# 3. Spring 与Struts整合

Spring与struts整合，关键点:

把action的创建，交给spring的ioc容器！

所以，需要引入jar文件：

**struts2-spring-plugin-2.3.4.1.jar 【struts源码】**

**spring-web-3.2.5.RELEASE.jar 【spring源码】**

整合完整步骤：

1. 引入jar

Struts核心jar

Spring

Core 核心 （5个）

Web 对web支持 (2个)

2. 配置

bean.xml

struts.xml

web.xml

struts2核心过滤器

启动时候，配置加载springIOC容器

3. Action 类

任务：

1. 写简单案例，练习spring 、

\* 搭建spring环境

\* 测试OK

\* dao/service/action

然后处理对象依赖关系，

\* set方法注入值, 几种方式

\* 注解

2. spring与struts整合，小案例

3. 优化前面项目，

融入spring功能！

目标：

1.代理模式

2. Aop编程

\* 手动实现aop编程

\* spring Aop 注解方式

\* Spring Aop XML 配置

\* 切入点表达式语法详解

3. Spring 对jdbc模块的支持

# 4. 代理模式

## 4.1概念

1. Proxy-代理:通过代理访问**目标对象**！

2. Java中代理：

(1)静态代理

(2)动态代理

1)Jdk 代理

2)Cglib 代理

3)Spring默认支持

## 4.2静态代理

1.特点：

(1)目标对象必须要实现接口

(2)代理对象，要实现与目标对象一样的接口

2.需求：

(1)IUserDao.java dao接口

|  |
| --- |
| package cn.itcast.a\_static;  public interface IUserDao {  void save();  void find();  } |

(2) UserDao.java dao实现

|  |
| --- |
| package cn.itcast.a\_static;  public class UserDao implements IUserDao {  @Override  public void save() {  System.out.println("模拟： 保存用户！");  }  @Override  public void find() {  System.out.println("查询");  }  } |

(3)UserDaoProxy.java dao代理类，对UserDao中功能进行扩展！

|  |
| --- |
| package cn.itcast.a\_static;  /\*\*  \* 静态代理  \* 特点：  \* 1. 目标对象必须要实现接口  \* 2. 代理对象，要实现与目标对象一样的接口  \* @heyutong  \* 2017年6月12日  \*/  public class UserDaoProxy implements IUserDao{  // 代理对象，需要维护一个目标对象  private IUserDao target = new UserDao();  @Override  public void save() {  System.out.println("代理操作： 开启事务...");  target.save();// 执行目标对象的方法  System.out.println("代理操作：提交事务...");  }  @Override  public void find() {  target.find();  }  } |

(4)App

|  |
| --- |
| package cn.itcast.a\_static;  public class App {  public static void main(String[] args) {  // 代理对象  UserDaoProxy userDaoProxy = new UserDaoProxy();  //private IUserDao target = new UserDao();  // 执行代理方法  userDaoProxy.save();//1.代理操作： 开启事务... 2.模拟： 保存用户！  // 3.代理操作：提交事务...  }  } |

3.缺点：

(1)代理对象，需要依赖目标对象的接口！

如果接口功能变化，目标对象变化，会引入代理对象的变化!

(2)对每一个目标对象，都要分别写一个代理类，麻烦！

(代理工厂)

## 4.3动态代理

1.动态代理：

(1)通常说的动态代理，就是指jdk代理!

因为是通过jdk的api在运行时期，动态的生成代理对象的！

(2)目标对象一定要实现接口, 代理对象不用实现接口!

2. JDK 生成代理对象的Api

|-- Proxy

|  |  |
| --- | --- |
| static [Object](mk:@MSITStore:E:\API\JDK_API_1_6_zh_CN.CHM::/java/lang/Object.html) | [newProxyInstance](mk:@MSITStore:E:\API\JDK_API_1_6_zh_CN.CHM::/java/lang/reflect/Proxy.html#newProxyInstance(java.lang.ClassLoader, java.lang.Class[], java.lang.reflect.InvocationHandler))([ClassLoader](mk:@MSITStore:E:\API\JDK_API_1_6_zh_CN.CHM::/java/lang/ClassLoader.html) loader, [Class](mk:@MSITStore:E:\API\JDK_API_1_6_zh_CN.CHM::/java/lang/Class.html)<?>[] interfaces, [InvocationHandler](mk:@MSITStore:E:\API\JDK_API_1_6_zh_CN.CHM::/java/lang/reflect/InvocationHandler.html) h) |

参数loader : 当前目标对象使用的类加载器！

参数interfaces : 当前目标对象实现的接口

参数 h: 接口类型，事件处理器.

当执行目标对象方法的时候，会触发事件； 把当前执行的方法(method对象)，传入事件处理器方法参数中, 这样就可以根据业务逻辑，判断是否执行目标对象方法或扩展功能！

3.例子

(1) IUserDao.java

|  |
| --- |
| package cn.itcast.b\_dynamic;  public interface IUserDao {  void save();  void find();  } |

(2) UserDao.java

|  |
| --- |
| package cn.itcast.b\_dynamic;  public class UserDao implements IUserDao{  @Override  public void save() {  System.out.println("模拟： 保存用户！");  }  @Override  public void find() {  System.out.println("查询");  }  } |

(3) ProxyFactory.java

|  |
| --- |
| package cn.itcast.b\_dynamic;  import java.lang.reflect.InvocationHandler;  import java.lang.reflect.Method;  import java.lang.reflect.Proxy;  /\*\*  \* 动态代理：代理工厂，给多个目标对象生成代理对象！  \* @heyutong  \* 2017年6月12日  \*/  public class ProxyFactory {  // 接收一个目标对象  private Object target;    public ProxyFactory(Object target) {  this.target = target;  }    // 返回对目标对象(target)代理后的对象(proxy)  public Object getProxyInstance() {  Object proxy = Proxy.newProxyInstance(  target.getClass().getClassLoader(), // 目标对象使用的类加载器  target.getClass().getInterfaces(), // 目标对象实现的所有接口  new InvocationHandler() {// 执行代理对象方法时触发  @Override  public Object invoke(Object proxy, Method method, Object[] args)  throws Throwable {  // 获取当前执行的方法的方法名  String methodName = method.getName();//find  // 方法返回值  Object result = null;    // 判断  if("find".equals(methodName)) {  // 直接调用目标对象方法  method.invoke(target, args);  } else {  System.out.println("开启事务...");  // 执行目标对象方法  result = method.invoke(target, args);  System.out.println("提交事务...");  }    return result;  }  });  return proxy;  }  } |

1. App.java

|  |
| --- |
| package cn.itcast.b\_dynamic;  public class App {  public static void main(String[] args) {  // 创建目标对象  UserDao target = new UserDao();  System.out.println("目标对象：" + target.getClass());  //目标对象：class cn.itcast.b\_dynamic.UserDao    // 代理对象  IUserDao proxy = (IUserDao)new ProxyFactory(target).getProxyInstance();  System.out.println("代理对象： " + proxy.getClass());  //代理对象： class com.sun.proxy.$Proxy0    // 执行代理对象的方法  //proxy.find();//执行public Object invoke方法 1.查询  proxy.save();//执行public Object invoke方法 1.开启事务...  //2.模拟： 保存用户！ 3.提交事务...  }  } |

## 4.4 Cglib代理

1.也叫”子类代理”

当目标对象没有实现接口，就不能使用jdk提供的代理，可以子类的方式实现！

在运行时期动态在内存中构建一个子类对象的方法，从而对目标对象扩展

2.Spring也支持cglib代理，核心包中已经包含此功能！

* JDK的动态代理限制:使用动态代理的对象必须实现一个或多个接口。用CGlib生成 代理类不需要接口
* CGLIB:强大的高性能的代码生成包，可在运行期扩展Java类与实现Java接口。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包底层: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。
* 用CGlib生成代理类是目标类的子类。
* 用CGLib生成的代理类重写了父类的方法。
* 拦截器中的intercept方法内容正好就是代理类中的方法体

3.例子

(1) IUserDao.java

|  |
| --- |
| package cn.itcast.c\_cglib;  public interface IUserDao {  void save();    void find();  } |

(2) UserDao.java

|  |
| --- |
| package cn.itcast.c\_cglib;  //目标对象  public class UserDao {  public static void save() {  System.out.println("模拟： 保存用户！");  }    public void find() {  System.out.println("查询");  }  } |

(3) ProxyFactory.javas

|  |
| --- |
| package cn.itcast.c\_cglib;  import java.lang.reflect.Method;  import org.springframework.cglib.proxy.Enhancer;  import org.springframework.cglib.proxy.MethodInterceptor;  import org.springframework.cglib.proxy.MethodProxy;  /\*\*  \* cglib代理：代理工厂，给多个目标对象生成代理对象！  \* @heyutong  \* 2017年6月12日  \*/  public class ProxyFactory implements MethodInterceptor{  // 接收一个目标对象  private Object target;    public ProxyFactory(Object target) {  this.target = target;  }    // 返回目标对象代理后的子类对象  public Object getProxyInstance() {  // 对target生成子类对象  // 字节码生成工具类  Enhancer enhancer = new Enhancer();  // 设置父类  enhancer.setSuperclass(target.getClass());  // 设置回调函数  enhancer.setCallback(this);  // 创建子类对象  return enhancer.create();  }    // 事件处理器，执行目标方法时候触发  @Override  public Object intercept(Object obj, Method method, Object[] args,  MethodProxy methodProxy) throws Throwable {  System.out.println("开启事务...");  Object result = method.invoke(target, args);  System.out.println("提交事务...");  return result;  }  } |

(4) App.java

|  |
| --- |
| package cn.itcast.c\_cglib;  public class App {  public static void main(String[] args) {  // 创建目标对象  UserDao target = new UserDao();  System.out.println("目标对象：" + target.getClass());  //目标对象：class cn.itcast.c\_cglib.UserDao    // 代理对象  UserDao proxy = (UserDao) new ProxyFactory(target).getProxyInstance();  System.out.println("代理对象： " + proxy.getClass());  //代理对象： class cn.itcast.c\_cglib.UserDao$$EnhancerByCGLIB$$f0dc6d19    // 执行代理对象的方法  proxy.save(); //打印:模拟： 保存用户！  }  } |

总结：

1. 目标对象可以不实现接口

2. 目标类不能为final， 如果为final报错

3. 方法如果为final/static, 不会被代理拦截！ 会直接执行目标对象方法 ！

## 4.5 代理总结

1.在SpringAop编程中，

(1)若目标对象实现了接口，spring使用JDK的java.lang.reflect.Proxy类代理。

优点：因为有接口，所以使系统更加松耦合

缺点：为每一个目标类创建接口

(2)若目标对象未实现任何接口，用CGLIB库生成目标对象的子类。

优点：因代理类与目标类是继承关系，不需要有接口的存在。

缺点：因没有使用接口，系统的耦合性没有使用JDK的动态代理好。

(3)如果目标没有实现接口、且为final , 不能进行aop编程，报错！不能生成代理！

2. JDK和CGLIB的区别:

(1)Java动态代理:利用反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用

InvokeHandler来处理。

(2)cglib动态代理:利用asm开源包，对代理对象类的class文件加载进来，通过修改其字节

码生成子类来处理。

(3)如目标对象实现了接口，默认会采用JDK的动态代理实现AOP

如目标对象实现了接口，可强制使用CGLIB实现AOP

如目标对象没有实现接口,必须用CGLIB库,spring会自动在JDK动态代理和CGLIB间转换

(4)如何强制使用CGLIB实现AOP:

1）添加CGLIB库，SPRING\_HOME/cglib/\*.jar

2）在spring配置文件中加入<aop:aspectj-autoproxy proxy-target-class="true"/>

(5)JDK动态代理和CGLIB字节码生成的区别？

1）JDK动态代理只能对实现了接口的类生成代理，而不能针对类

2）CGLIB是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法

因为是继承，所以该类或方法最好不要声明成final

# Aop编程

|  |
| --- |
| **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); // 业务**    **trans.commit(); 【关注点代码】**  **} catch (Exception e) {**  **e.printStackTrace();**  **if(trans != null){**  **trans.rollback();**  **}**  **} finally{**  **HibernateSessionFactoryUtils.closeSession(session);**  **}**  **}** |

Aop 编程，

主要是分离业务代码与关注点代码！

关注点代码，写一次，在执行业务代码时候**动态植入关注点代码**！

## 5.1自己实现aop编程

1.例1

1. jar包

|  |
| --- |
| commons-logging-1.1.3.jar  spring-aop-3.2.5.RELEASE.jar  spring-beans-3.2.5.RELEASE.jar  spring-context-3.2.5.RELEASE.jar //@Component  spring-core-3.2.5.RELEASE.jar  spring-expression-3.2.5.RELEASE.jar  spring-jdbc-3.2.5.RELEASE.jar |

1. applicationContext.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  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">  <!-- 开启注解扫描 -->  <context:component-scan base-package="cn.itcast.d\_my\_aop"></context:component-scan>  </beans> |

(3) TransactionAop.java

|  |
| --- |
| package cn.itcast.d\_my\_aop;  import org.springframework.stereotype.Component;  //重复代码  @Component("aop")  public class TransactionAop {  public void beginTransaction() {  System.out.println("开启事务..");  }    public void commit() {  System.out.println("提交事务..");  }  } |

(4) UserDao.java

|  |
| --- |
| package cn.itcast.d\_my\_aop;  import javax.annotation.Resource;  import org.springframework.stereotype.Repository;  @Repository // 把对象加入ioc容器  public class UserDao {  @Resource  private TransactionAop aop;    public void save() {  aop.beginTransaction(); //开启事务..  System.out.println("保存..."); //保存...  aop.commit(); //提交事务..  }  } |

(5) App.java

|  |
| --- |
| package cn.itcast.d\_my\_aop;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class App {  private ApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml", getClass());  @Test  public void test() {  UserDao userDao = (UserDao) ac.getBean("userDao");  userDao.save();  }  } |

2.例2

(1) applicationContext.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  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">  <!-- 开启注解扫描 -->  <context:component-scan  base-package="cn.itcast.d\_my\_aop2"></context:component-scan>    <!-- 实例化工厂 -->  <bean id="proxyFactory" class="cn.itcast.d\_my\_aop2.ProxyFactory"></bean>  <!-- 调用工厂方法，返回代理后的对象 -->  <bean id="userDaoProxy" factory-bean="proxyFactory"  factory-method="getProxyInstance">  <constructor-arg index="0" ref="userDao"></constructor-arg>  <constructor-arg index="1" ref="aop"></constructor-arg>  </bean>  </beans> |

(2) IUserDao.java

|  |
| --- |
| package cn.itcast.d\_my\_aop2;  public interface IUserDao {  void save();  } |

(3) UserDao.java

|  |
| --- |
| package cn.itcast.d\_my\_aop2;  import org.springframework.stereotype.Repository;  @Repository  public class UserDao implements IUserDao{  @Override  public void save() {  System.out.println("保存...");  }  } |

(4) TransactionAop.java

|  |
| --- |
| package cn.itcast.d\_my\_aop2;  import org.springframework.stereotype.Component;  @Component("aop")  public class TransactionAop {  public void beginTransaction() {  System.out.println("开启事务..");  }    public void commit() {  System.out.println("提交事务..");  }  } |

(5)

|  |
| --- |
| package cn.itcast.d\_my\_aop2;  import java.lang.reflect.InvocationHandler;  import java.lang.reflect.Method;  import java.lang.reflect.Proxy;  public class ProxyFactory {  /\*\*  \* 生成代理对象  \* @heyutong  \* 2017年6月13日  \* @param target 目标对象  \* @param aop 给目标对象动态注入的重复的代码(关注点代码)  \* @return  \*/  public Object getProxyInstance(final Object target,final TransactionAop aop){  return Proxy.newProxyInstance(target.getClass().getClassLoader(),  target.getClass().getInterfaces(),  new InvocationHandler() {//jdk代理    //userDao.save();触发  @Override  public Object invoke(Object proxy, Method method, Object[] args)  throws Throwable {  aop.beginTransaction(); // 执行重复代码 打印："开启事务.."  Object result = method.invoke(target, args);// 执行目标对象方法：  //userDao.save();打印:"保存..."  aop.commit(); // 打印:"提交事务.."  return result;  }  });  }  } |

(5)

|  |
| --- |
| package cn.itcast.d\_my\_aop2;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class App {  private ApplicationContext ac = new  ClassPathXmlApplicationContext("applicationContext.xml",getClass());  @Test  public void testApp() throws Exception {  IUserDao userDao = (IUserDao) ac.getBean("userDaoProxy");  userDao.save();  }  } |

## 5.2 aop编程, 注解方式

1.概念：

(1)Aop编程：关注点代码与业务代码分离！(jdk/cglib代理)

(2)关注点：重复执行的代码, 也叫关注点代码!

(3)切面：**关注点代码形成的类，叫切面**

(4)springAop编程，也叫面向切面编程！

Aop: Aspect Object Programming 面向切面编程!

(5)切面:事务，权限控制, 日志…

(6)切入点表达式

1)拦截方法，给方法所在的类，生成代理对象！

2)Spring在初始化容器时，会根据切入点表达式的规则，会符合拦截规则的方法所在的类生成代理对象！

2.使用Aop开发步骤：

(1)引入aop 相关 jar文件

(aspectj 在spring之前，面向切面开发的公用组件)

aopalliance.jar 【spring-framework-2.5.6\lib\aopalliance】

aspectjrt.jar 【spring-framework-2.5.6\lib\aspectj】

aspectjweaver.jar 【spring-framework-2.5.6\lib\aspectj】

spring-aop-3.2.5.RELEASE.jar 【Spring3.2源码】

(2)引入aop名称空间 applicationContext.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  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 -->  <aop:aspectj-autoproxy></aop:aspectj-autoproxy>  </beans> |

(3) IUserDao.java

|  |
| --- |
| package cn.itcast.e\_aop\_anno;  public interface IUserDao {  void save();  } |

(4) UserDao.java

|  |
| --- |
| package cn.itcast.e\_aop\_anno;  import org.springframework.stereotype.Repository;  @Repository  public class UserDao implements IUserDao{  @Override  public void save() {  System.out.println("保存...");  }  } |

(5) OrderDao.java

|  |
| --- |
| package cn.itcast.e\_aop\_anno;  import org.springframework.stereotype.Repository;  @Repository  public class OrderDao {  public void save() {  System.out.println("保存订单..");  }  } |

(6) TransactionAop.java

|  |
| --- |
| package cn.itcast.e\_aop\_anno;  import org.aspectj.lang.ProceedingJoinPoint;  import org.aspectj.lang.annotation.After;  import org.aspectj.lang.annotation.AfterReturning;  import org.aspectj.lang.annotation.AfterThrowing;  import org.aspectj.lang.annotation.Around;  import org.aspectj.lang.annotation.Aspect;  import org.aspectj.lang.annotation.Before;  import org.aspectj.lang.annotation.Pointcut;  import org.springframework.stereotype.Component;  //重复代码  @Component("aop")  @Aspect // 指定一个类为切面类  public class TransactionAop {  // 定义一个切入点表达式变量  //（后面使用这个切入点表达式的时候，直接引用方法名即可）  // execution-执行，\*:任意返回值，然后是包名，.\*意思是包下面的所有子包。(..)代  //表各种方法.  @Pointcut("execution(\* cn.itcast.e\_aop\_anno.UserDao.\*(..))")  public void pointcut\_(){  }    //【前置通知】  // 在执行业务方法，之前执行  @Before("pointcut\_()")  public void beginTransaction() {  System.out.println("[前置通知] 开启事务..");  }    //【后置通知】  // 在执行业务方法，之后执行  @After("pointcut\_()")  public void commit() {  System.out.println("[后置通知] 提交事务..");  }    // 【返回后通知】 在执行目标方法结束后执行, 出现异常不会执行  @AfterReturning("pointcut\_()")  public void afterReturing(){  System.out.println("[返回后通知]");  }    // 【异常通知】 在执行目标方法的时候出现异常时执行  @AfterThrowing("pointcut\_()")  public void afterThrowing(){  System.out.println("[异常通知]");  }    // 【环绕通知】 会环绕目标方法执行  @Around("pointcut\_()")  public void arround(ProceedingJoinPoint point) throws Throwable {  System.out.println("[环绕前：]");  point.proceed(); // 执行目标方法  System.out.println("[环绕后：]");  }  } |

(7) App.java

|  |
| --- |
| package cn.itcast.e\_aop\_anno;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class App {  private ApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml",getClass());    //1.springIOC容器中获取对象，用实现接收？ 报错！！！！！！！！！  @Test  public void testApp() throws Exception {  //java.lang.ClassCastException: $Proxy12 cannot be cast to cn.itcast.e\_aop\_anno.UserDao  // 总结：在spring的aop编程中，符合切入点表达式的目标类，如果目标对象有实现接口，从容器获取对象的时候，一定要通过接口接收！  // 否则，包类型转换错误！  /\*UserDao userDao = (UserDao) ac.getBean("userDao");  System.out.println(userDao.getClass());  userDao.save();\*/  }    //cglib代理  //2.springIOC容器中获取对象，用接口接收！  @Test  public void testApp\_cglib() throws Exception {  IUserDao userDao = (IUserDao) ac.getBean("userDao");  System.out.println(userDao.getClass());//打印：class com.sun.proxy.$Proxy12  userDao.save();//1:arround():打印"[环绕前：]"  //2:执行beginTransaction() 2:UserDao.save  //3：[环绕后：] 4:commit() 5:afterReturing()  }    // 没有生成代理对象，因为没有被切入点表达式拦截  // execution(\* cn.itcast.e\_aop\_anno.UserDao.\*(..))  /\*@Test  public void testApp\_save\_order() throws Exception {  OrderDao orderDao = (OrderDao) ac.getBean("orderDao");  System.out.println(orderDao.getClass());//打印:class cn.itcast.e\_aop\_anno.OrderDao  orderDao.save();//打印:"保存订单.."  }\*/  } |

\*使用Aop相关注解

|  |
| --- |
| @Aspect 指定一个类为切面类  (切面类也需实例化）  (切面类中方法,也叫通知）  @Before 前置通知 【在执行目标对象方法之前执行】  @After 后置通知 【在执行目标对象方法之后执行】  @AfterReturning 返回后通知 【在执行目标对象方法结束后执行,出现异常不执行】  @AfterThrowing 异常通知 【在执行目标对象方法出现异常时执行】  @Around 环绕通知 【环绕目标方法执行】  @Pointcut 定义一个切入点表达式变量  （后面使用这个切入点表达式的时候，直接引用方法名即可） |

3.Spring生成代理对象的过程:

(1)创建容器对象时,根据“切入点表达式”拦截的类，生成代理对象；

(2)如果目标对象有实现接口，使用jdk代理！

(3)如果目标对象没有实现接口，使用cglib代理！

(4)从容器获取代理后的对象

(5)执行代理对象的方法，在运行时期，动态植入“切面”类中的“通知”!

## 5.3 aop编程, XML配置方式

1.步骤：

(1) 引入aop 相关jar文件

(2) Aop配置 applicationContext.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  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实例加入容器 -->  <bean id="userDao" class="cn.itcast.f\_aop\_xml.UserDao"></bean>    <!-- 实例化切面类 -->  <bean id="aop111" class="cn.itcast.f\_aop\_xml.TransactionAop"></bean>    <!-- Aop相关配置 -->  <aop:config>  <!-- 切入点表达式定义 -->  <aop:pointcut expression="execution(\* cn.itcast.f\_aop\_xml.UserDao.\*(..))" id="pt"/>    <!-- 切面配置 -->  <aop:aspect ref="aop111">    <!-- 【环绕通知】 -->  <aop:around method="arroud" pointcut-ref="pt"/>    <!-- 【前置通知】 在目标方法之前执行 -->  <aop:before method="beginTransaction" pointcut-ref="pt"/>    <!-- 【后置通知】 -->  <aop:after method="commit" pointcut-ref="pt"/>    <!-- 【返回后通知】 -->  <aop:after-returning method="afterReturing" pointcut-ref="pt"/>    <!-- 异常通知 -->  <aop:after-throwing method="afterThrowing" pointcut-ref="pt"/>  </aop:aspect>  </aop:config>  </beans> |

(3) IUserDao.java

|  |
| --- |
| package cn.itcast.f\_aop\_xml;  public interface IUserDao {  void save();  } |

(4) UserDao.java

|  |
| --- |
| package cn.itcast.f\_aop\_xml;  public class UserDao implements IUserDao {  @Override  public void save() {  System.out.println("保存...");  }  } |

(5) TransactionAop.java

|  |
| --- |
| package cn.itcast.f\_aop\_xml;  import org.aspectj.lang.ProceedingJoinPoint;  // 切面类  public class TransactionAop {    public void beginTransaction() {  System.out.println("[前置通知] 开启事务..");  }    public void commit() {  System.out.println("[后置通知] 提交事务..");  }    public void afterReturing(){  System.out.println("[返回后通知]");  }    public void afterThrowing(){  System.out.println("[异常通知]");  }    public void arroud(ProceedingJoinPoint pjp) throws Throwable{  System.out.println("[环绕前：]");  pjp.proceed(); // 执行目标方法  System.out.println("[环绕后：]");  }  } |

(6) App.java

|  |
| --- |
| package cn.itcast.f\_aop\_xml;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class App {  private ApplicationContext ac = new ClassPathXmlApplicationContext(  "applicationContext.xml", getClass());  @Test  public void testApp() throws Exception {  // springIOC容器中获取对象，用接口接收！  IUserDao userDao = (IUserDao) ac.getBean("userDao");  System.out.println(userDao.getClass());//class com.sun.proxy.$Proxy4  //1:(arroud)[环绕前：] 2:(beginTransaction)[前置通知] 开启事务.. 3:(save)保存...  //4:(arroud)[环绕后：] 5:(commit)[后置通知] 提交事务.. 6:(afterReturing)[返回后通知]  userDao.save();  }  } |

## 5.4切入点表达式语法详解

1.切入点表达式：拦截指定的类，生成代理对象！

2. execution(

modifiers-pattern? 拦截的方法的访问修饰符

ret-type-pattern 方法返回类型，必须指定

declaring-type-pattern? 拦截的方法所在的类

name-pattern(param-pattern)  **拦截的方法（以及方法的参数列表）**

throws-pattern?) 方法声明的异常

3.例： execution (\* com.sample.service.impl..\*. \*(..))

(1)execution(): 表达式主体。

(2)第一个\*号：表返回类型， \*号表所有类型。

(3)包名：表需要拦截的包名，..表当前包和当前包的所有子包，com.sample.service.impl包、子孙包下所有类的方法。

(4)第二个\*号：表示类名，\*号表示所有的类。

(5)\*(..):最后这个星号表示方法名，\*号表示所有的方法，后面括弧里面表示方法的参数，两个句点表示任何参数

4.

(1) applicationContext.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  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.g\_execution"></context:component-scan>    <!-- dao实例加入容器 -->  <bean id="userDao" class="cn.itcast.g\_execution.UserDao"></bean>    <!-- 实例化切面类 -->  <bean id="aop" class="cn.itcast.g\_execution.TransactionAop"></bean>    <!-- Aop相关配置 -->  <aop:config>  <!-- 切入点表达式定义 -->    <!-- 1. 拦截指定的方法(通用) -->  <!-- 1:[环绕前：] 2:保存... 3:[环绕后：] 4:保存订单..-->  <aop:pointcut expression="execution(\* cn.itcast.g\_execution.UserDao.save(..))" id="pt"/>    <!-- 2. 拦截指定的类下所有的方法 -->  <!-- 1:[环绕前：] 2:保存... 3:[环绕后：] 4:保存订单..-->  <!-- <aop:pointcut expression="execution(\* cn.itcast.g\_execution.UserDao.\*(..))" id="pt"/> -->    <!-- 3. 拦截指定包下所有的类的所有方法 -->  <!-- 1:[环绕前：] 2:保存... 3:[环绕后：] 4:[环绕前：] 5:保存订单.. 6:[环绕后：] -->  <!-- <aop:pointcut expression="execution(\* cn.itcast.g\_execution.\*.\*(..))" id="pt"/> -->    <!-- 3. 拦截指定包，以及其子包下所有类的所有方法 -->  <!-- 1:[环绕前：] 2:保存... 3:[环绕后：] 4:[环绕前：] 5:保存订单.. 6:[环绕后：] -->  <!-- <aop:pointcut expression="execution(\* cn..\*.\*(..))" id="pt"/> -->    <!-- 5. 拦截所有的public方法 -->  <!-- 1:[环绕前：] 2:保存... 3:[环绕后：] 4:[环绕前：] 5:保存订单.. 6:[环绕后：] -->  <!-- <aop:pointcut expression="execution(public \* \*(..))" id="pt"/> -->    <!-- 6. 拦截所有的包含save方法 -->  <!-- 1:[环绕前：] 2:保存... 3:[环绕后：] 4:[环绕前：] 5:保存订单.. 6:[环绕后：] -->  <!-- <aop:pointcut expression="execution(\* \*save\*(..))" id="pt"/> -->    <!-- 7. 拦截UserDao.save()方法与OrderDao.save() -->  <!-- 1:[环绕前：] 2:保存... 3:[环绕后：] 4:[环绕前：] 5:保存订单.. 6:[环绕后：] -->  <!-- <aop:pointcut expression="execution(\* cn..UserDao.save(..)) || execution(\* cn..OrderDao.save(..))" id="pt"/> -->  <!--<aop:pointcut expression="execution(\* cn..UserDao.save(..)) or execution(\* cn..OrderDao.save(..))" id="pt"/>-->    <!-- 8. 不拦截UserDao.save()方法 -->  <!--<aop:pointcut expression="!execution(\* cn..UserDao.save(..))" id="pt"/>-->  <!--<aop:pointcut expression=" not execution(\* cn..UserDao.save(..))" id="pt"/>-->    <!-- 9. 拦截UserDao.save()同时拦截OrderDao.save() -->  <!-- 注意： 这种很少用，一般都是或者的关系即: || 、 or -->  <!-- 都没被拦截到 -->  <!--<aop:pointcut expression="execution(\* cn..UserDao.save(..)) and execution(\* cn..OrderDao.save(..))" id="pt"/>-->  <!-- <aop:pointcut expression="execution(\* cn..UserDao.save(..)) &amp;&amp; execution(\* cn..OrderDao.save(..))" id="pt"/> -->    <!-- 切面配置 -->  <aop:aspect ref="aop">  <aop:around method="arroud" pointcut-ref="pt"/>  </aop:aspect>  </aop:config>  </beans> |

(2) IUserDao.java

|  |
| --- |
| package cn.itcast.g\_execution;  public interface IUserDao {  void save();  void get();  } |

(3) UserDao.java

|  |
| --- |
| package cn.itcast.g\_execution;  public class UserDao implements IUserDao {  @Override  public void save() {  System.out.println("保存...");  }    public void get() {  System.out.println("获取...");  }  } |

(4) OrderDao.java

|  |
| --- |
| package cn.itcast.g\_execution;  import org.springframework.stereotype.Repository;  @Repository  public class OrderDao {    public void save() {  System.out.println("保存订单..");  }  } |

(5) TransactionAop.java

|  |
| --- |
| package cn.itcast.g\_execution;  import org.aspectj.lang.ProceedingJoinPoint;  // 切面类  public class TransactionAop {    public void beginTransaction() {  System.out.println("[前置通知] 开启事务..");  }    public void commit() {  System.out.println("[后置通知] 提交事务..");  }    public void afterReturing(){  System.out.println("[返回后通知]");  }    public void afterThrowing(){  System.out.println("[异常通知]");  }    public void arroud(ProceedingJoinPoint pjp) throws Throwable{  System.out.println("[环绕前：]");  pjp.proceed(); // 执行目标方法  System.out.println("[环绕后：]");  }  } |

(6) App.java

|  |
| --- |
| package cn.itcast.g\_execution;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class App {  private ApplicationContext ac = new ClassPathXmlApplicationContext(  "applicationContext.xml", getClass());  @Test  public void testApp() throws Exception {  // springIOC容器中获取对象，用接口接收！  IUserDao userDao = (IUserDao) ac.getBean("userDao");  System.out.println(userDao.getClass());  userDao.save();  //userDao.get();    OrderDao orderDao = (OrderDao) ac.getBean("orderDao");  orderDao.save();  }  } |

总结：

拦截，一定要指定到方法！

共性问题：

1. @Override报错

Jdk1.5 只支持父类方法的重写，不支持接口

Jdk1.6 修复这个问题

2. xml配置没有提示

# 6. Spring 对jdbc模块的支持

1. Spring Core ioc容器

Spring Web 对web支持（与struts整合）

Spring Aop 面向切面编程

Spring Jdbc 对jdbc的支持

2. Spring对Jdbc的支持:

(1)简化对jdbc的操作

(2)提供了**JdbcTemplate**模板工具类，简化jdbc的操作！

3.jar文件：

spring-jdbc-3.2.5.RELEASE.jar 工具类包

spring-tx-3.2.5.RELEASE.jar 事务支持依赖包

mysql-connector-java-5.1.7-bin.jar 数据库驱动包！

c3p0-0.9.1.2.jar 连接池

4.步骤：

~~(1) 原始的jdbc操作代码~~

~~(2) 对连接管理~~

对jdbc操作进行简化

🡪 JdbcTemplate 工具类Api

🡪 JdbcTemplate与DataSource依赖关系

(1) jdbc.properties

|  |
| --- |
| driverClass=com.mysql.jdbc.Driver  jdbcUrl=jdbc:mysql://localhost:3306/day37?useUnicode=true&characterEncoding=UTF-8  user=root  password=123  initialPoolSize=5  maxPoolSize=8  acquireIncrement=2 |

(2) applicationContext.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  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">  <context:property-placeholder location="classpath:cn/itcast/h\_jdbc/jdbc.properties"/>    <!-- 1. 实例化连接池 -->  <!-- 导入c3p0-0.9.1.2.jar -->  <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">  <property name="driverClass" value="${driverClass}"></property>  <property name="jdbcUrl" value="${jdbcUrl}"></property>  <property name="user" value="${user}"></property>  <property name="password" value="${password}"></property>  <property name="initialPoolSize" value="${initialPoolSize}"></property>  <property name="maxPoolSize" value="${maxPoolSize}"></property>  <property name="acquireIncrement" value="${acquireIncrement}"></property>  </bean>    <!-- 2. 创建JdbcTemplate对象 -->  <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">  <property name="dataSource" ref="dataSource"></property>  </bean>    <!-- 实例化dao -->  <bean id= "deptDao" class="cn.itcast.h\_jdbc.DeptDao">  <property name="jdbcTemplate" ref="jdbcTemplate"></property>  </bean>  </beans> |

(3) Dept.java

|  |
| --- |
| package cn.itcast.h\_jdbc;  public class Dept {  private int id;  private String name;  public int getId() {  return id;  }  public void setId(int id) {  this.id = id;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  } |

(4) IDeptDao.java

|  |
| --- |
| package cn.itcast.h\_jdbc;  import java.io.Serializable;  import java.util.List;  //接口  public interface IDeptDao {  public void save(Dept dept);  public void update(Dept dept);  public void delete(Serializable id);  public Dept findById(Serializable id);  public List<Dept> getAll();  } |

(5) DeptDao.java

|  |
| --- |
| package cn.itcast.h\_jdbc;  import java.io.Serializable;  import java.sql.ResultSet;  import java.sql.SQLException;  import java.util.List;  import org.springframework.jdbc.core.JdbcTemplate;  import org.springframework.jdbc.core.RowMapper;  public class DeptDao implements IDeptDao {    // 接收容器注入的JdbcTemplate对象  private JdbcTemplate jdbcTemplate;    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {  this.jdbcTemplate = jdbcTemplate;  }    // 1. 原始jdbc代码  @Override  public void save(Dept dept) {  //需导入spring-tx-3.2.5.RELEASE.jar包  jdbcTemplate.update("insert into t\_dept(deptName) values(?)", dept.getName());  }  @Override  public void update(Dept dept) {  jdbcTemplate.update("update t\_dept set deptName=? where id=?", dept.getName(), dept.getId());  }  @Override  public void delete(Serializable id) {  jdbcTemplate.update("delete from t\_dept where id=?", id);  }  @Override  public Dept findById(Serializable id) {  // queryForList 把每一行都封装为map对象，再添加到list中  //List<Map<String, Object>> list = jdbcTemplate.queryForList("select \* from t\_dept");    // 传入类型参数，表示查询的列的类型;这里只能查询一列  //List<String> list = jdbcTemplate.queryForList("select deptName from t\_dept", String.class);  List<Dept> list = jdbcTemplate.query("select \* from t\_dept where id=?", new MyRowMapper(), id);  return (list!=null&&list.size()>0)?list.get(0):null;  }  @Override  public List<Dept> getAll() {  List<Dept> list = jdbcTemplate.query("select \* from t\_dept", new MyRowMapper());  return list;  }    // 封装Springjdbc查询的结果集  //建立内部类实现RowMapper接口  // RowMapper可将数据中的每一行数据封装成用户定义的类.  // sping与hibernate相结合时基本用不到,大多是在spring单独用时用到.  class MyRowMapper implements RowMapper<Dept> {    // 如何解析一行  @Override  public Dept mapRow(ResultSet resultSet, int rowNum) throws SQLException {  Dept dept = new Dept();  dept.setId(resultSet.getInt("id"));  dept.setName(resultSet.getString("deptName"));  return dept;  }  }  } |

(6) App.java

|  |
| --- |
| package cn.itcast.h\_jdbc;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class App {  private ApplicationContext ac = new  ClassPathXmlApplicationContext("applicationContext.xml",App.class);  @Test  public void testApp() throws Exception {  IDeptDao deptDao = (IDeptDao) ac.getBean("deptDao");  //deptDao.save(new Dept());  //deptDao.delete(9);    Dept dept = new Dept();  dept.setId(3);  //jdbcUrl需加上"?useUnicode=true&characterEncoding=UTF-8",避免中文乱码  dept.setName("人事部");  deptDao.update(dept);    // 查询方法  /\*Dept dept = deptDao.findById(1);  System.out.println(dept);  System.out.println(dept.getName());\*/  //System.out.println(deptDao.getAll());  }  } |

目标：

1. Spring声明式事务管理

2. Spring 与 Hibernate 整合

\* Spring创建SessionFactory几种方式

\* Spring对dao操作的支持

3. SSH 初步整合

# 7. Spring声明式事务管理

## 7.1概念

1.事务:一组操作的执行单元，相对于数据库操作来讲，事务管理的是一组SQL指令，比如增加，修改，删除等，事务的一致性，要求，这个事务内的操作必须全部执行成功，如果在此过程种出现了差错，比如有一条SQL语句没有执行成功，那么这一组操作都将全部回滚

用来确保数据的完整性和一致性

2. 事务特性（ACID）

1)Atomic(原子性):要么都成功，要么都失败

2)Consistent(一致性):数据应该不被破坏. 一旦所有事务动作完成，事务就被提交。数据和资

源就处于一种满足业务规则的一致性状态中

3)Isolate(隔离性):用户间操作不相混淆. 可能有许多事务会同时处理相同的数据，因此每个

事物都应该与其他事务隔离开来，防止数据损坏

4)Durable(持久性):永久保存. 一旦事务完成，无论发生什么系统错误，它的结果都不应该受到影响。通常情况下，事务的结果被写到持久化存储器中

3.数据隔离级别

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 隔离级别 | 脏读 | 不可重复读 | 幻象读 | 第一类丢失更新 | 第二类丢失更新 |
| READ UNCOMMITED | 允许 | 允许 | 允许 | 不允许 | 允许 |
| READ COMMITED | 不允许 | 允许 | 允许 | 不允许 | 允许 |
| REPEATABLE READ | 不允许 | 不允许 | 允许 | 不允许 | 不允许 |
| SERIALIZABLE | 不允许 | 不允许 | 不允许 | 不允许 | 不允许 |

SqlServer2008R2的默认隔离级别是“READ COMMITED”，

MySQL的默认隔离级别是“REPEATABLE READ”。

2. 程序中两种事务管理方式

(1)**编程式事务管理**

1) 将事务管理代码嵌入到业务方法中来控制事务的提交和回滚，在编程式事务中，必须在每个业务操作中包含额外的事务管理代码

2)编写程序式的事务管理可清楚的定义事务边界，可实现**细粒度的事务控制**，比如可通过程序代码来控制事务何时开始，何时结束等，与后面介绍的声明式事务管理相比，它可实现细粒度的事务控制，例如jdbc,hibernate. spring中不提倡使用。

|  |
| --- |
| JDBC事务控制：  **con.setAutoCommite (false); 设置事务手动提交**  Hibernate中事务控制：  **session.beginTransaction(); 开启事务** |

3)优缺点：

事务控制精确

事务代码，与业务逻辑处理代码，耦合在一起!

事务代码，不能共用！ 重新写事务控制操作！

开发效率低，不便于维护！ (不想用事务，要改代码！)

**(2)声明式事务管理 (在Spring中使用)**

1) 大多数情况下比编程式事务管理更好用。它将事务管理代码从业务方法中分离出来，以声明的方式来实现事务管理。事务管理作为一种横切关注点，可以通过AOP方法模块化。Spring通过Spring AOP框架支持声明式事务管理。

2)如不需要细粒度的事务控制，可使用声明式事务，在Spring中，只需在Spring配置文件中做一些配置，即可将操作纳入到事务管理中，解除了和代码的耦合， 这是对应用代码影响最小的选择，从这一点再次验证了Spring关于**AOP的概念**。当不需要事务管理时，可以直接从Spring配置文件中移除该设置

3)特点：

Spring提供的声明式事务管理，用到Aop概念！

对指定的方法添加事务控制，这里只需要配置即可！

修改事务控制实现或删除事务控制操作，只需要移除xml事务相关配置！

3)注意：

只能对某个方法应用事务！ （因“切入点表达式”拦截的是方法，控制不了方法内部代码！）

所以，Spring声明式事务管理，即为**粗粒度的事务控制！**

声明式事务管理器类：

|  |
| --- |
| Jdbc:  DataSourceTransactionManager 管理jdbc中事务控制  Hibernate:  HibenateTransactionManager 管理hibernate中事务控制 |

3. Spring既支持编程式事务管理，也支持声明式的事务管理

(1)Spring事务的传播属性：

当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。

(2)事务的传播行为可以由传播属性指定。Spring定义了7种传播行为：

PROPAGATION\_REQUIRED 如当前没有事务，就新建一个事务，如已经存在一个事务，加入到这个事务中。最常见的选择。

PROPAGATION\_SUPPORTS 支持当前事务，如当前没有事务，就以非事务方式执行。

PROPAGATION\_MANDATORY 使用当前的事务，如当前没有事务，就抛出异常。

PROPAGATION\_REQUIRES\_NEW 新建事务，如当前存在事务，把当前事务挂起。

PROPAGATION\_NOT\_SUPPORTED 以非事务方式执行操作，如当前存在事务，就把当前事务挂起。

PROPAGATION\_NEVER 以非事务方式执行，如果当前存在事务，则抛出异常。

PROPAGATION\_NESTED 如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION\_REQUIRED类似的操作。

(3)Spring声明式事务让我们从复杂的事务处理中得到解脱，再不必去处理获得连接、关闭连接、事务提交和回滚等操作，也无需在与事务相关的方法中处理大量的try…catch…finally代码。

4.并发事务所导致的问题

在同一个应用程序或者不同应用程序中的多个事务在同一个数据集上并发执行时，可能会出现许多意外的问题：

① 脏读：脏读发生在一个事务读取了另一个事务改写但尚未提交的数据时。如果改写在稍后被回滚了，那么第一个事务获取的数据就是无效的。

② 不可重复读：不可重复读发生在一个事务执行相同的查询两次或两次以上，但是每次都得到不同的数据时。这通常是因为另一个并发事务在两次查询期间更新了数据

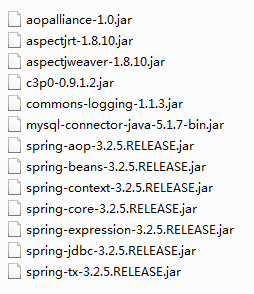
③ 幻读：幻读与不可重复读类似。它发生在一个事务(T1)读取了几行数据，接着另一个并发事务(T2)插入了一些数据时。在随后的查询中，第一个事务(T1)就会发现多了一些原本不存在的记录

## 7.2声明式事务管理 – JDBC

### 7.2.1 XML 配置方式实现

.步骤：

1. 引入jar文件



Spring 核心

Spring Aop 切面编程

Spring-jdbc / Spring-tx / 驱动包、连接池

2. 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"  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">    <!-- 1. 数据源配置 -->  <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">  <property name="driverClass" value="com.mysql.jdbc.Driver"></property>  <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/day37"></property>  <property name="user" value="root"></property>  <property name="password" value="123"></property>  <property name="initialPoolSize" value="5"></property>  <property name="maxPoolSize" value="8"></property>  </bean>    <!-- 2. JdbcTemplate配置 , 注入数据源-->  <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">  <property name="dataSource" ref="dataSource"></property>  </bean>    <!-- 3. dao实例，注入jdbcTemplate -->  <bean id="deptDao" class="cn.itcast.a\_tx\_jdbc.DeptDao">  <property name="jdbcTemplate" ref="jdbcTemplate"></property>  </bean>    <!-- 4. Service实例，注入dao实例 -->  <bean id="deptService" class="cn.itcast.a\_tx\_jdbc.DeptService">  <property name="deptDao" ref="deptDao"></property>  </bean>    <!-- 5. Spring声明式事务管理配置 -->  <!-- 5.1 配置事务管理器类 -->  <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  <property name="dataSource" ref="dataSource"></property>  </bean>    <!-- 5.2 事务通知配置, 拦截到指定的方法后如何管理事务 -->  <!-- find\* find开头的方法，是只读的事务 -->  <!-- \* 上面所有的方法都不满足时候，采用的事务控制规则 -->  <!--事务增强 -->  <tx:advice id="txAdvice" transaction-manager="txManager">  <!-- 事务属性定义 -->  <tx:attributes>  <tx:method name="find\*" read-only="true"/>  <tx:method name="get\*" read-only="true"/>  <tx:method name="\*" read-only="false"/>  </tx:attributes>  </tx:advice>    <!-- 5.3 事务Aop配置 = 切入点表达式 + 应用上面的事务通知 -->  <!-- 使用tx/aop来配置 -->  <aop:config>  <!-- 通过aop定义事务增强切面 -->  <aop:pointcut expression="execution(\* cn.itcast.a\_tx\_jdbc.\*Service.\*(..))" id="pt"/>  <!-- 引用事务增强 -->  <aop:advisor advice-ref="txAdvice" pointcut-ref="pt"/>  </aop:config>  </beans> |

注意:

(1) advice(建议)的命名：由于每个模块都会有自己的Advice，所以在命名上需作出规范，如模块名+Advice（只是一种命名规范）。

(2) tx:attribute：配置的是作为事务的方法的命名类型

如<tx:method name="save\*" propagation="REQUIRED"/>

\*为通配符，代表以save为开头的所有方法，即表示符合此命名规则的方法作为一个事务。

propagation="REQUIRED"代表支持当前事务，如果当前没有事务，就新建一个事务。最常见的选择。

(3)aop:pointcut:配置参与事务的类，由于是在Service中进行数据库业务操作，配的应该是包含那些作为事务的方法的Service类。

特别注意的是id的命名，同样由于每个模块都有自己事务切面，所以我觉得初步的命名规则应为 all+模块名+ServiceMethod。且每个模块之间不同之处还在于以下一句：expression="execution(\* com.test.testAda.test.model.service.\*.\*(..))"

1. aop:advisor：就是把上面所配置的事务管理两部分属性整合起来作为整个事务管理。



3. (1)IDeptDao.java

|  |
| --- |
| package cn.itcast.a\_tx\_jdbc;  public interface IDeptDao {  void save();  } |

(2) DeptDao.java

|  |
| --- |
| package cn.itcast.a\_tx\_jdbc;  import org.springframework.jdbc.core.JdbcTemplate;  public class DeptDao implements IDeptDao {  // 注入JdbcTemplate对象  private JdbcTemplate jdbcTemplate;  public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {  this.jdbcTemplate = jdbcTemplate;  }  @Override  public void save() {  jdbcTemplate.update("insert into t\_dept(deptName)values('test..')");  }  } |

4. (1)IDeptService.java

|  |
| --- |
| package cn.itcast.a\_tx\_jdbc;  public interface IDeptService {  void save();  } |

(2) DeptService.java

|  |
| --- |
| package cn.itcast.a\_tx\_jdbc;  public class DeptService implements IDeptService {  // 注入dao  private IDeptDao deptDao;    public void setDeptDao(IDeptDao deptDao) {  this.deptDao = deptDao;  }  @Override  public void save() {  deptDao.save();  int i = 1/0;//此处出错，之前保存的数据也全部回滚;  //若此处加上try catch,则此处不会触发回滚  deptDao.save();  }  } |

5. App.java

|  |
| --- |
| package cn.itcast.a\_tx\_jdbc;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class App {  @Test  public void test() {  //spring-context-3.2.5.RELEASE.jar  //commons-logging-1.1.3.jar  //spring-expression-3.2.5.RELEASE.jar  ApplicationContext ac = new ClassPathXmlApplicationContext("bean.xml",getClass());  IDeptService deptService = (IDeptService) ac.getBean("deptService");  deptService.save();  }  } |

2. dao/service

3. 配置

\* 数据源

\* JdbcTemplate

\* Dao/Service

\* spring声明式事务管理配置

(拦截service方法的执行，动态植入事务控制代码！)

4. 测试

Save();

Int i = 1/0;

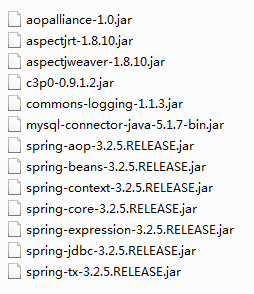
Save();

|  |
| --- |
| <?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"*  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"*>  <!-- 1. 数据源配置 -->  <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\_demo"*></property>  <property name=*"user"* value=*"root"*></property>  <property name=*"password"* value=*"root"*></property>  <property name=*"initialPoolSize"* value=*"3"*></property>  <property name=*"maxPoolSize"* value=*"6"*></property>  </bean>  <!-- 2. JdbcTemplate配置 , 注入数据源-->  <bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>  <property name=*"dataSource"* ref=*"dataSource"*></property>  </bean>  <!-- 3. dao实例，注入jdbcTemplate -->  <bean id=*"deptDao"* class=*"cn.itcast.a\_tx\_jdbc.DeptDao"*>  <property name=*"jdbcTemplate"* ref=*"jdbcTemplate"*></property>  </bean>    <!-- 4. Service实例，注入dao实例 -->  <bean id=*"deptService"* class=*"cn.itcast.a\_tx\_jdbc.DeptService"*>  <property name=*"deptDao"* ref=*"deptDao"*></property>  </bean>  <!-- 5. Spring声明式事务管理配置 -->    <!-- 5.1 配置事务管理器类 -->  <bean id=*"txManager"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>  <property name=*"dataSource"* ref=*"dataSource"*></property>  </bean>    <!-- 5.2 事务通知配置, 拦截到指定的方法后如何管理事务 -->  <!-- find\* find开头的方法，是只读的事务 -->  <!-- \* 上面所有的方法都不满足时候，采用的事务控制规则 -->  <tx:advice id=*"txAdvice"* transaction-manager=*"txManager"*>  <tx:attributes>  <tx:method name=*"find\*"* read-only=*"true"*/>  <tx:method name=*"get\*"* read-only=*"true"*/>  <tx:method name=*"\*"* read-only=*"false"*/>  </tx:attributes>  </tx:advice>    <!-- 5.3 事务Aop配置 = 切入点表达式 + 应用上面的事务通知 -->  <aop:config>  <aop:pointcut expression=*"execution(\* cn.itcast.a\_tx\_jdbc.\*Service.\*(..))"* id=*"pt"*/>  <aop:advisor advice-ref=*"txAdvice"* pointcut-ref=*"pt"*/>  </aop:config>  </beans> |

### 7.2.2 注解方式实现

步骤：

1. 引入相关jar包



2. 开启：<tx:annotation-driven transaction-manager="txManager"/>

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"  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">  <!-- 1. 数据源配置 -->  <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">  <property name="driverClass" value="com.mysql.jdbc.Driver"></property>  <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/day37"></property>  <property name="user" value="root"></property>  <property name="password" value="123"></property>  <property name="initialPoolSize" value="5"></property>  <property name="maxPoolSize" value="8"></property>  </bean>    <!-- 2. JdbcTemplate配置 , 注入数据源-->  <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">  <property name="dataSource" ref="dataSource"></property>  </bean>    <!-- 事务管理器类 -->  <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  <property name="dataSource" ref="dataSource"></property>  </bean>    <!-- 开启注解扫描 -->  <context:component-scan base-package="cn.itcast.b\_tx\_jdbc\_anno"></context:component-scan>    <!-- spring声明式事务管理，注解开启 -->  <tx:annotation-driven transaction-manager="txManager"/>  </beans> |

3. 使用@Transactional 注解:在需要添加事务控制的方法上写这个注解

@Transactional

写到方法上， 表示当前方法应用事务控制

写到类上， 表示当前类的所有方法都会应用事务

写到父类上， 当执行父类的这个方法时候才应用事务！

(1) LogService.java

|  |
| --- |
| package cn.itcast.b\_tx\_jdbc\_anno;  import javax.annotation.Resource;  import org.springframework.jdbc.core.JdbcTemplate;  import org.springframework.stereotype.Service;  import org.springframework.transaction.annotation.Propagation;  import org.springframework.transaction.annotation.Transactional;  @Service  public class LogService {  @Resource  private JdbcTemplate jdbcTemplate;  //事务控制 Propagation.REQUIRES\_NEW- 外层出错,这里不回滚;Propagation.REQUIRED:回滚  //不管当前执行方法有没有事务环境，都开启新事务！  @Transactional(propagation=Propagation.REQUIRES\_NEW)  public void insertLog() {  jdbcTemplate.update("insert into t\_dept(deptName)values('test\_insertLog..')");  }  } |

(2) IDeptService.java

|  |
| --- |
| package cn.itcast.b\_tx\_jdbc\_anno;  public interface IDeptService {  void save();  } |

(3) DeptService.java

|  |
| --- |
| package cn.itcast.b\_tx\_jdbc\_anno;  import javax.annotation.Resource;  import org.springframework.stereotype.Service;  import org.springframework.transaction.annotation.Isolation;  import org.springframework.transaction.annotation.Propagation;  import org.springframework.transaction.annotation.Transactional;  @Service  public class DeptService implements IDeptService{  @Resource  private IDeptDao deptDao;    @Resource  private LogService logService;  @Transactional(  readOnly=false,// 读写的事务，当修改数据时候用；如果查询就设置为true  isolation=Isolation.DEFAULT,// 事务隔离级别  timeout=-1,// 事务执行的超时时间, -1 表示不超时  //noRollbackFor=ArithmeticException.class, // 遇到指定的异常不回滚  propagation=Propagation.REQUIRED// 事务传播行为  )  public void save() {  // 插入日志  //Spring遇到嵌套事务时，当被嵌套的事务被定义为“PROPAGATION\_REQUIRES\_NEW”时，内层Service方法被调用时，  //外层方法的事务被挂起，内层事务相对于外层事务是完全独立的，有独立的隔离  性等  //当一个Service类中的某方法调用另一个Service类  (“PROPAGATION\_REQUIRES\_NEW”)中某方法时，内层事务提交也好，  //回滚也好，都不受外层事务提交或回滚的影响。就是如果内层事务提交了，即使  外层事务回滚了，内层事务提交了的数据也不会回归回来了  logService.insertLog(); // 始终插入    int i = 1/0;    // 插入部门  deptDao.save();  }  } |

4.

(1) IDeptDao.java

|  |
| --- |
| package cn.itcast.b\_tx\_jdbc\_anno;  public interface IDeptDao {  void save();  } |

(2) DeptDao.java

|  |
| --- |
| package cn.itcast.b\_tx\_jdbc\_anno;  import javax.annotation.Resource;  import org.springframework.jdbc.core.JdbcTemplate;  import org.springframework.stereotype.Repository;  @Repository // 当前类加入ioc容器  public class DeptDao implements IDeptDao{  //注入JdbcTemplate对象  @Resource  private JdbcTemplate jdbcTemplate;  @Override  public void save() {  jdbcTemplate.update("insert into t\_dept(deptName)values('test\_anno..')");  }  } |

5. App.java

|  |
| --- |
| package cn.itcast.b\_tx\_jdbc\_anno;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class App {  @Test  public void testApp() throws Exception {  ApplicationContext ac = new ClassPathXmlApplicationContext("bean.xml",getClass());  IDeptService deptService = (IDeptService) ac.getBean("deptService");  deptService.save();  }  } |

### 7.2.3事务属性

1.通常由事务的传播行为、事务的隔离级别、事务的超时值、事务只读标志组成。在进行事务划分时，需要进行事务定义，也就是配置事务的属性。Spring在TransactionDefinition接口中定义这些属性，以供PlatfromTransactionManager使用，PlatfromTransactionManager是spring事务管理的核心接口。

2.

// 当前方法应用事务

@Transactional(

readOnly=**false**, // 读写的事务，当修改数据时候用；如果查询就设置为true

isolation=Isolation.*DEFAULT*, // 事务隔离级别

timeout=-1, // 事务执行的超时时间, -1 表示不超时

noRollbackFor=ArithmeticException.**class**, // 遇到指定的异常不回滚

propagation=Propagation.*REQUIRES\_NEW* // 事务传播行为

)

1. 事务传播行为:

Propagation.

|  |  |
| --- | --- |
| REQUIRES\_NEW | 当前执行方法必须在事务环境下运行！ 且当前执行方法始终开启一个新的事务！ |
| REQUIRED | 当前执行方法必须在事务环境下运行！  如果调用当前方式时候已经有一个事务环境，当前执行方法会加入当前事务环境，就不开启新的事务；  如果调用当前方法时候没有事务环境，就开启一个新的事务! |
| SUPPORTS | 支持事务环境！ 如果当前方法没有事务，也可以运行！ |
| Never | 当前方法不能在事务环境下运行！ |

# 8. Spring 与 Hibernate 整合

Spring与Hibernate整合，

**\* 单例的SessionFactory对象，交给spring的IOC容器创建！**

**\* 事务管理，交给spring声明式事务管理器**

演示步骤：

1. 没有整合案例

2. 整合

整合步骤：

1. 引入Hibernate/spring框架相关包

\* hibernate jar

\*spring – core

\* spring – aop

\* spring – orm 对orm支持

spring-jdbc-3.2.5.RELEASE.jar

**spring-orm-3.2.5.RELEASE.jar**

spring-tx-3.2.5.RELEASE.jar

2. hibernate.cfg.xml

3. dao/service

## 8.1 \* Spring创建SessionFactory几种方式

|  |
| --- |
| <?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"*  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"*>    <!-- 连接池, 通过spring管理 -->  <bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>  <property name=*"jdbcUrl"* value=*"jdbc:mysql:///hib\_demo"*></property>  <property name=*"driverClass"* value=*"com.mysql.jdbc.Driver"*></property>  <property name=*"user"* value=*"root"*></property>  <property name=*"password"* value=*"root"*></property>  <property name=*"initialPoolSize"* value=*"3"*></property>  <property name=*"maxPoolSize"* value=*"6"*></property>  </bean>    <!-- Spring 与 Hibenate整合 (Spring创建SessionFactory) -->    <!-- 方式1： 直接加载hibernate.cfg.xml的方式，创建sessionFactory对象  <bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">  <property name="configLocation" value="classpath:hibernate.cfg.xml"></property>  </bean>  -->    <!-- 方式2： 连接池交给spring管理，其他配置还是写到hibernate.cfg.xml中  <bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">  <property name="dataSource" ref="dataSource"></property>  <property name="configLocation" value="classpath:hibernate.cfg.xml"></property>  </bean>  -->    <!-- 方式3：(推荐) 所有的配置都在spring中完成-->  <bean id=*"sessionFactory"* class=*"org.springframework.orm.hibernate3.LocalSessionFactoryBean"*>  <!-- a. 注入连接池 -->  <property name=*"dataSource"* ref=*"dataSource"*></property>    <!-- b. hibernate常用配置： 方言、自动建表、显示sql -->  <property name=*"hibernateProperties"*>  <props>  <prop key=*"hibernate.dialect"*>org.hibernate.dialect.MySQLDialect</prop>  <prop key=*"hibernate.show\_sql"*>true</prop>  <prop key=*"hibernate.hbm2ddl.auto"*>update</prop>  </props>  </property>    <!-- c. 加载所有的映射(根据路径加载)  <property name="mappingLocations">  <list>  <value>classpath:cn/itcast/entity/\*.hbm.xml</value>  </list>  </property>  -->  <!-- c. 根据目录加载所有的映射 -->  <property name=*"mappingDirectoryLocations"*>  <list>  <value>classpath:cn/itcast/entity</value>  </list>  </property>  </bean>    <!-- 创建dao实例 -->  <bean id=*"deptDao"* class=*"cn.itcast.dao.DeptDao"*>  <property name=*"sessionFactory"* ref=*"sessionFactory"*></property>  </bean>    <!-- 创建service实例 -->  <bean id=*"deptService"* class=*"cn.itcast.service.DeptService"*>  <property name=*"deptDao"* ref=*"deptDao"*></property>  </bean>      <!--  Spring声明式事务管理配置  -->  <!-- a. 事务管理器 -->  <bean id=*"txManager"* class=*"org.springframework.orm.hibernate3.HibernateTransactionManager"*>  <property name=*"sessionFactory"* ref=*"sessionFactory"*></property>  </bean>    <!-- b. 事务通知 -->  <tx:advice id=*"txAdvice"* transaction-manager=*"txManager"*>  <tx:attributes>  <tx:method name=*"\*"* read-only=*"false"*/>  </tx:attributes>  </tx:advice>    <!-- c. Aop配置 = 切入点表达式 + 应用通知规则 -->  <aop:config>  <aop:advisor advice-ref=*"txAdvice"* pointcut=*"execution(\* cn..\*Service.\*(..))"*/>  </aop:config>  </beans> |

## 8.2 \* Spring对dao操作的支持

如下：

1. JDBC

Spring 提供了JdbcTemplate模板工具类，对原始的jdbc操作进行简化！

2. Hibernate

Spring 提供了对hibernate的sessionFactory创建的支持 (整合)

🡪 直接在dao中使用sessionFactory对象操作数据库

🡪 使用Spring提供的 HibernateTemplate 工具类操作数据库

优点： 对session的常用操作进行封装！ 比较方便!

🡪 (推荐)HibernateDaoSupport工具类

Dao类直接继承HibernateDaoSupport工具类即可

HibernateDaoSupport对hibernateTemlate类进行了封装

# 9. SSH整合

SSH 整合：

Spring 与 Struts 整合

🡪 Action创建交给Spring完成

Spring 与 Hibernate整合

🡪 SessionFactory创建，交给spring完成 （管理事务）

步骤：

1. 引入jar文件

Struts核心jar

Hibernate核心jar

Spring

SpringCore 核心jar文件 （5个）

SpringWeb 对struts支持（2个）

SpringAop 声明式事务管理（4个）

SpringORM 对hibernate支持 （3个）

Orm + jdbc + tx jar文件

其他

驱动 + 连接池

2. 配置

Web.xml 配置struts核心过滤器 + Spring容器初始化

Struts.xml 配置访问路径与action类的映射关系

applicationContext-public.xml Spring容器配置 【公用配置】

applicationContext-dao.xml Spring容器配置 【dao配置】

applicationContext-service.xml Spring容器配置 【service配置】

applicationContext-action.xml Spring容器配置 【action配置】

3. 代码

cn.itcast.entity 实体类： 封装数据/业务

cn.itcast.dao 数据访问层接口： 定义功能

cn.itcast.dao.impl 接口实现： 功能实现

cn.itcast.service 业务逻辑层 （控制事务）

cn.itcast.service.impl 实现

cn.itcast.action 控制层： 接收请求数据、处理请求、返回结果视图标记跳转

任务：

Action中写add/update/delete/findById/showAll 方法

1. 数据要在页面输入

2. 查询到的结果要在jsp页面显示

# 10. 补充

## 10.1 IntrospectorCleanupListener作用

|  |
| --- |
| <!--web.xml-->  <listener> <listener-class>org.springframework.web.util.IntrospectorCleanupListener</listener-class>  </listener> |

1、用于解决java.beans.Introspector导致的内存泄漏的问题

2、应配置在web.xml中与Spring相关监听器中第一个位置(要在ContextLoaderListener前面)

3、JDK中的java.beans.Introspector类的用途是发现Java类是否符合JavaBean规范如果有的框架或程序用到了Introspector类,那么就会启用一个系统级别的缓存,此缓存会存放一些曾加载并分析过的JavaBean的引用。当Web服务器关闭时,由于此缓存中存放着这些JavaBean的引用,所以垃圾回收器无法回收Web容器中的JavaBean对象,最后导致内存变大。而org.springframework.web.util.IntrospectorCleanupListener就是专门用来处理Introspector内存泄漏问题的辅助类。IntrospectorCleanupListener会在Web服务器停止时清理Introspector缓存,使那些Javabean能被垃圾回收器正确回收。Spring自身不会出现这种问题，因为Spring在加载并分析完一个类之后会马上刷新JavaBeans Introspector缓存,这就保证Spring中不会出现这种内存泄漏的问题。但有些程序和框架在使用了JavaBeans Introspector之后,没有进行清理工作(如Quartz,Struts),最后导致内存泄漏

1. 应用程序类，几乎不需要直接使用JavaBeans Introspector，所以，通常都不是Introspector resource造成内存泄露。相反，许多库和框架，不清理Introspector，例如： Struts和Quartz。

一个简单Introspector泄漏将会导致整个Web应用程序的类加载器不会被回收！结果将会是在web应用程序关闭时，该应用程序所有的静态类资源（比如：单实例对象）都没有得到释放。而导致内存泄露的根本原因其实并不是这些未被回收的类！

IntrospectorCleanupListener应该注册为web.xml中的第一个Listener，在任何其他Listener之前注册，比如在Spring's ContextLoaderListener注册之前，才能确保IntrospectorCleanupListener在Web应用的生命周期适当时机生效。

1. 源码

|  |
| --- |
| public class IntrospectorCleanupListener implements ServletContextListener {  public void contextInitialized(ServletContextEvent event) {  }    public void contextDestroyed(ServletContextEvent event) {  Introspector.flushCaches(); //把缓存中的内容清除。  }  } |

6.使用IntrospectorCleanupListener 解决quartz引起的内存泄漏问题

"在服务器运行过程中，Spring不停的运行的计划任务和OpenSessionInViewFilter，使得Tomcat反复加载对象而产生框架并用时可能产生的内存泄漏，则使用IntrospectorCleanupListener作为相应的解决办法。"

只知道servlet标准不允许在web容器内自行做线程管理，quartz的问题确实存在。

对于Web容器来说，最忌讳应用程序私自启动线程，自行进行线程调度，像Quartz这种在web容器内部默认就自己启动了10线程进行异步job调度的框架本身就是很危险的事情，很容易造成servlet线程资源回收不掉，所以我一向排斥使用quartz。

quartz还有一个问题就是不支持cluster。导致使用quartz的应用都没有办法做群集。

如果是我的话，我采取的办法就是自己单独启动一个Job Server，来跑job，不会部署在web容器中。