一、Spring框架的学习目标：

1.Spring框架的两大主题：

  (1)控制反转与依赖注入(IoC&DI)

     IoC: Inversion of Control

     DI:  Dependency Injection

         依赖注入是控制反转的具体实现！

  (2)面向切面编程(AOP)

     AOP: Aspect Orinented Programming

2.Spring框架的整合：

  (1)与JDBC及Hibernate的整合；

  (2)与Struts2的整合；

二、依赖关系和依赖解耦

1.什么是类的依赖关系？

    如果在一个类的业务方法中使用另外一个类的业务方法来完成自己的功能，

    那么这两个类就构成了依赖关系。

  【举例】

    有两个Bean类：UseBean和EnHelloBean。

    其中EnHelloBean中封装了一个输出英文问候语的业务方法sayHello()方法；

    而UseBean封装了一个业务方法show()方法并且在该方法中 通过调用

  EnHelloBean的sayHello()方法，输出问候语。

  【分析】:

  UseBean是调用类

  EnHelloBean是被调用类(所依赖的类)。

2.依赖关系的两种表现形式

  (1)仅在调用类方法中依赖

          ①在方法中创建所依赖对象；

          ②被依赖对象的生命周期仅局限在调用类的方法范围内；

         【案例】:

        spring01\_dependent1->dependent1.method

  (2)属性依赖

           ①将所依赖的类作为调用类属性的类型；

           ②被依赖对象的生命周期与调用类对象的生命周期相同；

          【注意】:Spring框架中研究的依赖形式是属性依赖！

          【案例】:

        spring01\_dependent1->dependent1.property

UserBean类：

|  |
| --- |
| /\*  【知识点】   \*     依赖关系的表现形式二（属性依赖）   \*/  **public** **class** UserBean {  **private** EnHelloBean ehello;    **public** EnHelloBean getEhello() {  **return** ehello;      }    **public** **void** setEhello(EnHelloBean ehello) {  **this**.ehello = ehello;      }        //业务方法  **public** **void** show(){          System.*out*.println("调用"+ehello.getClass().getName()+"对象的sayHello方法：");          ehello.sayHello();      }  } |

UserBeanTest类：

|  |
| --- |
| **public** **class** UserBeanTest {      /\*       \* UserBean类的执行方法：       \*  1）创建被调用类的对象       \*  2）创建调用类对象       \*  3）设置调用类与被调用类的依赖关系       \*  4）执行调用类的业务方法       \*/  **public** **static** **void** main(String[] args) {          //①创建被调用类的对象          EnHelloBean enObj = **new** EnHelloBean();          //②创建调用类对象          UserBean userBean = **new** UserBean();          //③设置调用类与被调用类的依赖关系          userBean.setEhello(enObj);            //④执行调用类的业务方法          userBean.show();      }  } |

3.依赖耦合

  (1)什么是依赖耦合？

          调用类的依赖属性的类型是具体的实现类；

          即调用类依赖具体的实现类；

  (2)依赖耦合的问题是:

          针对调用类实施功能扩展时，需要大量修改代码，

         【例如】:

               功能扩展——要求UseBean显示一个中文的问候语。

         【方案】:

       (1)增加一个能够显示中文问候语的bean类，即ZhHelloBean。

       (2)需要修改UseBean及测试类中的代码；

          【案例】:

        spring01\_dependent1->dependent1.property.couple

4.依赖解耦

  (1)什么是依赖解耦？

          封装调用类时，它的依赖属性的类型是接口而不是具体的实现类；

|  |
| --- |
| /\*   \* UserBean依赖所定义的接口   \*  要求UserBean的依赖属性的类型是接口类型   \*/  **public** **class** UserBean {  **private** IHelloBean hello;    **public** IHelloBean getHello() {  **return** hello;      }    **public** **void** setHello(IHelloBean hello) {  **this**.hello = hello;      }        //业务方法  **public** **void** show(){          System.*out*.println("调用"+hello.getClass().getName()+"对象的sayHello方法：");          hello.sayHello();      }  } |

  (2)依赖解耦的目的：

          当调用类需要增加新的功能时，调用类的代码不需要变更或者变更量最小。

  (3)如何实施依赖解耦？

       ①调用类依赖接口，而不是具体的类；

      ②被调用的类（所依赖的类）需要实现接口；

|  |
| --- |
| /\*       \* 【测试】       \*  UserBean类的业务方法------show（）       \*/  **public** **static** **void** main(String[] args) {          //只想检测接口里面的方法，所以定义为接口变量          IHelloBean hello; //接口类型的变量            //①创建所依赖的对象          EnHelloBean enHello = **new** EnHelloBean();          ZhHelloBean zhHello = **new** ZhHelloBean();          //②创建调用对象          UserBean userBean = **new** UserBean();          //③设置调用对象与依赖对象的依赖关系  //      hello = enHello; //接口变量引用所依赖的对象          hello = zhHello;          userBean.setHello(hello);            //执行调用对象的业务方法          userBean.show();      }  } |

  【案例】:

    spring01\_dependent1->dependent1.property.decouple

-------------------------------------------------------------------------------------------------------------------------一、控制反转(IoC)的概念

    控制反转指的是控制权反转，英文Inversion of Control

1.什么是控制权？(指谁来控制对象的创建；传统的应用程序对象的创建是由程序本身控制的。使用spring后，是由spring来创建对象的)

  (1)创建对象的权利；

  (2)设置对象依赖关系的权利；

2.什么是反转？(正传指程序来创建对象，反转指程序本身不去创建对象，而变为被动接受对象)

  (1)传统方式:

          调用类掌握控制权，即在调用类的执行方法中进行:

          ①创建所依赖的对象；

          ②创建调用对象；

          ③设置调用对象和所依赖对象的依赖关系；

  (2)反转方式:

          ①调用类将控制权转移给第三方工厂类；

          ②调用类在执行方法中向工厂类索要对象并执行相应的业务方法；

          【案例】:

         spring02\_dependent2->dependent2.demo1

     MyBeanFactory类：

|  |
| --- |
| /\*   \* 【说明】   \*   将控制权转移给第三方工厂类   \*/  **public** **class** MyBeanFactory {    **public** **static** Object getUserBean(){          //①创建调用类所依赖的对象          EnHelloBean enHello = **new** EnHelloBean();          ZhHelloBean zhHello = **new** ZhHelloBean();          //②创建调用类的对象          UserBean userBean = **new** UserBean();            //使用接口变量引用所依赖对象  //      IHelloBean hello = enHello;            //使用接口变量引用所依赖对象          IHelloBean hello = zhHello;            //③设置对象之间的依赖关系          userBean.setHello(hello);    **return** userBean;      }  } |
| /\*   \* 【说明】 通过工厂类索要目标对象   \*/  **public** **class** UserBeanTest {  **public** **static** **void** main(String[] args) {          UserBean userBean = (UserBean) MyBeanFactory.*getUserBean*();          userBean.show();      }  } |

二.Spring框架的IoC容器

  (1)Spring框架的IoC容器实际上是一个实现了BeanFactory接口的工厂类，

          例如：XmlBeanFactory

  (2)BeanFactory的功能：

          ①通过Resource对象，加载对象的配置信息；

       (a)对象的配置信息通常采用xml文件；

       (b)xml文件的标准名称是：applicationContext.xml，也可以采用自定义名称；

       (c)配置文件放在类路径下，即src下的相应包中；

|  |
| --- |
| public class UserBeanTest {    /\*     \*  【知识点】  \*  1）创建SpringIoC容器----BeanFactory     \*  2）通过IoC容器获得对象     \*/  public static void main(String[] args) {  /\*\* 将applicationContext.xml配置文件，封装成一个Resource对象         \*/  Resource config =new ClassPathResource("dependent2/demo2/applicationContext.xml");  /\*  根据Resource对象，创建BeanFactory对象         \*/  BeanFactory bf = new XmlBeanFactory(config);  //IoC容器  通过BeanFactory，获得需要的bean对象  UserBean ub = (UserBean)bf.getBean("userBean");  //执行bean对象的业务方法     ub.show();    }} |

          ②根据xml文件的配置，创建对象和设置对象的依赖关系

            【注意】:SpringIoC容器是通过反射方式创建对象的。

|  |
| --- |
| <bean id=*"enHello"* class=*"dependent2.demo2.EnHelloBean"*/>      <!-- EnHelloBean enHello = new EnHelloBean(); -->        <bean id=*"zhHello"* class=*"dependent2.demo2.ZhHelloBean"*/>        <bean id=*"userBean"* class=*"dependent2.demo2.UserBean"*>          <property name=*"hello"* ref=*"enHello"*/>          <!-- userBean.setHello(hello) -->      </bean> |

          ③管理对象的生命周期并负责提供对象；

                         =======

   (3)案例参考：

      spring02\_dependent2->dependent2.demo2

-------------------------------------------------------------------------------------------------------------------------

一、Spring框架中IoC容器的常见类型

1.BeanFactory接口，实现类是XmlBeanFactory，

  (1)特点:

          ①提供了IoC容器的核心功能，通常被称为"Bean工厂"；

          ②创建容器时，需要Resource（资源）对象；

  (2)功能:

      ①实例化对象及设置对象的依赖关系；

      ②管理对象的生命周期

      ③负责提供对象；

2.ApplicationContext接口，

    实现类是ClassPathXmlApplicationContext （\*）

  (1)特点:

          ①继承于BeanFactory接口，因而功能更加完善，

             是实际应用中的标准容器，通常被称为"应用上下文"；

          ②不需要资源对象，直接针对XML配置文件的路径创建容器对象；

  (2)功能:

          ①拥有BeanFactory的所有功能；

          ②支持国际化和事件发布等先进功能

3.WebApplicationContext接口，实现类是XmlWebApplicationContext

  (1)特点:

          ①ApplicationContext接口的子接口，专用于Web应用中的IoC容器，

          ②通过Web容器监听器(ContextLoaderListener)加载配置文件

             并创建Spring容器；

  (2)功能:

          整合时提供组件，有两种情况:

          ①为控制器提供业务组件；

          ②负责提供所有组件(包括控制器);

二、利用ApplicationContext容器管理对象

1.bean对象的创建时机:

  (1)默认情况下，容器创建的同时bean也被创建；

  (2)设置lazy-init属性为true

          当向容器索要对象时，容器才创建对象；

|  |
| --- |
| **public** **class** Test3 {    **private** **static** **final** String *CONFIG* = "applicationContext.xml";    **public** **static** **void** main(String[] args) {          /\*           \* ①创建一个IoC容器对象           \* ②创建对象的时机，即当容器对象创建时，bean对象也被创建           \*/          ClassPathXmlApplicationContext ioc = **new** ClassPathXmlApplicationContext(*CONFIG*);          /\*           \* 当我们需要获得IoC容器中所管理的bean对象时，IoC容器才会创建bean对象           \*/          MyBean3 myBean = (MyBean3) ioc.getBean(*CONFIG*);          //关闭 容器          ioc.close();      }  } |

2.bean对象的创建模式，即设置scope属性的值：

  (1)singleton(单例对象)

          容器只创建一个对象

          当向容器索要对象时，IoC容器每次提供相同的对象

  (2)prototype(原型对象):

          容器创建多个对象(多例对象)

          当向容器索要对象时，IoC容器每次提供不同的对象

  (3)针对于Web应用，IoC容器可以将所创建的对象存入

     HttpServletRequest对象和HttpSession对象中:

       ①request

       ②session

3.bean对象的再次初始化:

  (1)执行时机:在构造器之后执行的方法；

  (2)配置属性:init-method

4.bean对象销毁之前执行的方法(在单例模式中有效):

  (1)执行时机:关闭容器前，容器首先要销毁容器中所创建的对象

  (2)配置属性：destroy-method

|  |
| --- |
|  |

 ------------------------------------------------------------------------------------------------------------------------

1.传统的JavaBean概念

  (1)Bean代表类的一种封装形式。

  (2)特点：

          ①无参构造器；

          ②bean对象的属性名称由set和get方法的名称决定，

             比如：setName(String name),属性是name；

       public Class Student{

           private String sname;

           public void setName(String sname){

                       this.sname=sname;

           }

       }

          ③实现序列化接口。

2.Spring框架下，Bean类(对象）的概念

  (1)对传统意义上的javabean的概念进行了扩展。

  (2)保留了set/get方法的规范标准。

  (3)其它标准放松要求

 ------------------------------------------------------------------------------------------------------------------------

一、依赖注入的概念：

1.什么是依赖注入？

  (1)依赖注入的英文是:Dependency Injection，简称:DI；

  (2)依赖注入是指通过IoC容器设置bean对象的属性，而不是在程序中通过调用setter方法或者有参构造器进行设置；

  (3)依赖注入需要在xml配置文件中，提供具体的属性值；

2.属性值的注入分为两种情况(配置形式)

    根据Bean类中属性的类型不同，属性值有两种配置形式:

  (1)字面值注入

          如果bean类中属性的类型是如下2中情况，bean标签使用value属性

          或者在bean标签内使用value子标签；

        ①基本数据类型及其基本数据类型包装类:

      byte、short、char、int、long、boolean、double、float

        ②String类型

  (2)对象注入

          如果bean类中属性的类型是其它引用类型(String和基本数据类型包装类以外的)，

          配置时bean标签使用ref属性或者在bean标签内使用ref子标签。

|  |
| --- |
| <!-- 通过配置文件，为JavaBean对象的属性实施注入 -->      <bean id=*"regtime"* class=*"java.util.Date"*/>        <bean id=*"someone"* class=*"di.demo.SomeOneBean"*>          <!-- 字面值注入 -->          <property name=*"name"* value=*"王大"*/>          <!-- 相当于someone.setName("王大") -->          <property name=*"age"* value=*"18"*/>          <property name=*"height"* value=*"1.78"*/>          <property name=*"ismale"* value=*"true"*/>            <!-- 对象注入 -->          <property name=*"regtime"* ref=*"regtime"*/>      </bean> |

二、IoC容器实施依赖注入的三种形式：

1.Setter注入(最为常用)

  (1)要求：bean类定义了属性的set方法；

  (2)特点：先创建对象，再设置属性；

  【案例】：spring03\_DI-di.demo2

2.构造器注入

  (1)要求：bean类定义了有参构造器；

  (2)特点：创建对象的同时设置了属性；

  【案例】：spring03\_DI-di.demo3

Bean类：

|  |
| --- |
| **public** **class** A {  **private** IB **b**;  **private** String name;    **public** A() {      }      //提供一个有参构造器  **public** A(IB b, String name) {  **this**.b = b;  **this**.name = name;      }        //业务方法  **public** **void** print(){          System.*out*.println("--输出--");          b.show();          System.*out*.println("name:"+name);      }  } |

配置文件：

|  |
| --- |
| <bean id=*"***b***"* class=*"di.demo3.B"*/>      <bean id=*"a"* class=*"di.demo3.A"*>          <**constructor-arg index="0" ref="b"**/>   //对象注入，对应bean类中第一个属性           <**constructor-arg index="1" value="李芬"**/>  //字面值注入，对应bean类name属性          <!--          没有set方法，不能使用property标签          <property name="b" ref="b"/>          <property name="name" value="李芬"/>          -->      </bean> |

3.接口注入：

  (1)要求：bean类实现IoC容器中所定义的接口，该接口中提供了属性的set方法；

  (2)特点：先创建对象，再设置属性；

   【案例】：

      在Struts2框架中，向Action注入HttpSession对象的Map结构的一种方式是:

      可以在封装Action时实现SessionAware接口，从而使Action具有如下方法：

                void setSession(Map<String,Object> session)

          ,或者实现ServletRequestAware接口,拥有如下方法:

                void setServletRequest(HttpServletRequest request)

三、bean对象的属性继承

1.子bean对象的配置，需要使用parent属性来说明与父bean对象的关系，

     表明子bean对象的所有属性来源于父bean对象。

2.子bean对象可以重新配置继承下来的属性或者添加自己特有的属性配置；

3.父bean对象可以设置abstract属性为true，表明IoC容器不必创建父bean对象，

此时父bean标签中的class属性可以省略。

4.注册的抽象bean，是不能通过IoC容器获得的；强行获得会产生异常。

【案例】：spring03\_DI中的di.demo4包和di.demo5包

|  |
| --- |
| <!--          abstract表明是抽象的bean对象，Spring框架不会创建抽象的bean对象          注册的抽象bean，是不能通过IoC容器获得的，强行获得会产生异常      -->      <bean id=*"inheritedBean"* class=*"di.demo4.SomePersonBean"* **abstract=*"true****"*>          <property name=*"name"* value=*"王大"*/>          <property name=*"age"* value=*"30"*/>      </bean>      <!-- parent属性表明bean对象继承于parent属性指定的bean对象 -->      <bean id=*"person1"* class=*"di.demo4.SomePersonBean"*  **parent**=*"inheritedBean"*>          <property name=*"name"* value=*"王二"*/>          <property name=*"male"* value=*"男"*/>      </bean>      <bean id=*"person2"* class=*"di.demo4.SomePersonBean"*  **parent**=*"person1"*>          <property name=*"name"* value=*"王三"*/>      </bean> |

四、bean类中集合属性的依赖注入（重点）

1.集合属性的类型：

  (1)数组(2)List(3)Set(4)Map(5)Properties

2.参考案例:

  spring03\_DI中的di.demo6包

Bean类：

|  |
| --- |
| **public** **class** CollectionBean {  **private** String[] strArry;  **private** List<String> city;  **private** Set<String> name;  **private** Map<String,Object> books;  **private** Properties params;//键值对   ----注意输出的方法      -------------Set/get方法--------------      //显示集合属性  **public** **void** show(){          System.*out*.println("--字符串数组--");  **for** (String str : strArry) {              System.*out*.println(str+" ");          }            System.*out*.println("\n--List城市名称---");  **for** (String c : city) {              System.*out*.println(c+" ");          }            System.*out*.println("\n--Set朋友姓名---");  **for** (String na : name) {              System.*out*.println(na+" ");          }            System.*out*.println("\n--Map书籍名称---");          Set<String> keys = books.keySet();  **for** (String key : keys) {              System.*out*.println(key+":"+books.get(key));          }            System.*out*.println("--MySQL连接参数--");          //获得属性名称的Set集合          Set<String> ids = params.stringPropertyNames();  **for** (String id : ids) {              System.*out*.println(id+":"+params.getProperty(id));          }      }  } |

配置文件：

|  |
| --- |
| <bean id=*"collectionBean"* class=*"di.demo6.CollectionBean"*>          <!-- 注入数组对象，【注意】使用list标签 -->  **<property** name=*"strArry"*>  **<list>**  **<value>**welcome</value>                  <value>beijing</value>  **</list>**  **</property>**            <!-- 注入List对象 -->          <property name=*"city"*>  **<list>**  **<value>**北京</value>                  <value>上海</value>  **</list>**          </property>            <!-- 注入Set对象 -->          <property name=*"name"*>  **<set>**  **<value>**王大</value>                  <value>钱贵</value>  **</set>**          </property>            <!-- 注入Map对象 -->          <property name=*"books"*>  **<map>**  **<entry key**=*"ISBNoo1"* **value**=*"天龙八部"*/>                  <entry key=*"ISBN002"* value=*"白发摩旅"*/>  **</map>**          </property>            <!-- 注入Properties对象 -->  **<property** name=*"params"*>  **<props>**  **<prop key**=*"url"***>**jdbc:mysql://localhost:3306/test**</prop>**                  <prop key=*"driver"*>com.mysql.jdbc.Driver</prop>                  <prop key=*"username"*>root</prop>                  <prop key=*"password"*>123456</prop>  **</props>**  **</property>**      </bean> |

 ------------------------------------------------------------------------------------------------------------------------

1.什么是自动装配？

  (1)在XML配置文件中注册一个bean时，通过使用autowire属性告知IoC容器

          自动为其属性实施依赖注入。例如：

          <bean id="" class="" autowire="">

  (2)如果bean类中属性的类型是基本数据类型、String类型、Class类型以及

          这些类型的数组，则不能实施自动装配。

2.自动装配的三种形式：

  (1)按名称自动装配(autowire="byName")

          根据bean类的属性名称，在xml配置文件中查找相同注册名称的bean对象，

          如果找到，实施自动装配。

         【注意】:

            ①bean类的属性名称由setter/getter方法指定；

            ②xml文件中bean对象的注册名称由bean标签的id属性指定；

         【案例】:

      spring03\_DI\_autowire->autowire1.byname

Bean类：

|  |
| --- |
| **public** **class** Person {  **private** String pname;  **private** Company xxx; //company    **public** String getPname() {  **return** pname;      }  **public** **void** setPname(String pname) {  **this**.pname = pname;      }  **public** Company getXxx() {  **return** xxx;      }  **public** **void** setXxx(Company cmpy) {  **this**.xxx = cmpy;      }  } |

配置文件:

|  |
| --- |
| <bean id=*"xxx"* class=*"autowire1.byname.Company"*>          <property name=*"cname"* value=*"IBM"*/>          <property name=*"address"* value=*"Beijing"*></property>      </bean>        <bean id=*"person"*            class=*"autowire1.byname.Person"*  **autowire=*"byName"*>**            <property name=*"pname"* value=*"wangda"*/>      </bean> |

  (2)按类型自动装配(autowire="byType")

          根据bean类的属性类型，在xml文件中查找相同类型的bean对象，

          如果找到，实施自动装配。

    【注意】:

     ①bean类的属性类型是set方法中的参数类型(可能是接口类型)；

     ②xml文件中，注册bean的类型由bean标签的class属性指定，

      (指定的可能是一个接口的具体实现类)。

     ③如果xml文件中注册了多个相同类型的bean对象，不能采用byType

          自动装配，此时应采用手动装配，明确指明依赖关系！

     【案例】:

       spring03\_DI\_autowire->autowire2.bytype

Bean类：

|  |
| --- |
| **public** **class** Person {  **private** String pname;  **private** ICompany cmpy;    **public** String getPname() {  **return** pname;      }  **public** **void** setPname(String pname) {  **this**.pname = pname;      }  **public** ICompany getCmpy() {  **return** cmpy;      }  **public** **void** setCmpy(ICompany cmpy) {  **this**.cmpy = cmpy;      }  } |

配置文件：

|  |
| --- |
| <bean id=*"bbb"* class=*"autowire1.bytype.Company"*>          <property name=*"cname"* value=*"IBM"*/>          <property name=*"address"* value=*"Beijing"*/>      </bean>        <bean id=*"company2"* class=*"autowire1.bytype.Company"*>          <property name=*"cname"* value=*"HP"*/>          <property name=*"address"* value=*"Shanghai"*></property>      </bean>        <bean id=*"person1"* class=*"autowire1.bytype.Person"*  **autowire=*"byType"***> <!-- 自动装配 -->          <property name=*"pname"* value=*"wangda"*/>          <property name=*"cmpy"* **ref**=*"bbb"*/> <!-- 手动装配 -->      </bean> |

   (3)构造器自动装配(autowire="constructor")

            根据bean类中构造器的参数类型，在xml文件中查找相同类型的bean对象，

            如果找到，实施自动装配。

           【注意】:

              ①要求bean类提供有参构造器；

              ②bean类中构造器的参数类型实际上是bean类的属性类型，

                 因此构造器自动装配本质上是按类型自动装配；

         【案例】:

         spring03\_DI\_autowire->autowire3.constructor

Bean类：

|  |
| --- |
| **public** **class** Person {  **private** String pname;  **private** Company cmpy;    **public** Person() {      }        //参数的顺序无所谓  **public** Person(String pname, Company cmpy) {  **super**();  **this**.pname = pname;  **this**.cmpy = cmpy;      }  **public** String getPname() {  **return** pname;      }  **public** Company getCmpy() {  **return** cmpy;      }  } |

配置文件：

|  |
| --- |
| <bean id=*"cmpy1"* class=*"autowire1.contructor.Company"*>          <property name=*"cname"* value=*"IBM"*/>          <property name=*"address"* value=*"Beijing"*/>      </bean>      <!-- 构造器自动装配本质上是按类型自动装配 -->      <bean id=*"person2"* class=*"autowire1.contructor.Person"*  **autowire=*"constructor****"*>          <**constructor-arg value**=*"wangda"*/>      </bean> |

   (4)理论上还有一种情况——自动探测(autowire="autodetect")

            如果bean类提供了有参的构造器则采用constructor自动装配，

            如果没有提供有参构造器则采用byType自动装配。

           【注意】:

               在MyEclipse中并不支持此种形式的自动装配！

3.自动装配属性(autowire="")设置小结:

  (1)手动装配——显示指定

          ①缺省autowire属性

          ②autowire="default"

          ③autowire="no"

  (2)自动装配——按照属性名称

          ④autowire="byName"

  (3)自动装配——按照属性类型

          ⑤autowire="byType"

  (4)自动装配——按照构造器参数

          ⑥autowire="constructor"

 ------------------------------------------------------------------------------------------------------------------------

#常用模式小结：

1.工厂模式(\*)

 (1)静态工厂方法模式；例如：dao组件的解耦

 (2)(实例)工厂方法（Hibernate框架产生Session对象的方式）；

 (3)抽象工厂（了解、Spring框架底层用来管理代理对象创建的方式）

2.责任链模式(\*)（过滤器、拦截器） 难点

3.模板方法模式（Spring整合jdbc以及Spring整合Hibernate）

4.装饰模式（new BufferedReader(new FileReader())

5.代理模式(\*)（静态代理、动态代理）难点

6.单例模式(\*)

  (1)特点：

          ①单例类自己创建自己的唯一实例；

          ②构造器私有；

          ③使用静态变量存储唯一实例；

  (2)单例模式与静态工厂方法模式的关系：

          单例模式使用了静态工厂方法来提供自己的实例。

7.多例模式：

  Locale

    枚举类 enum

  public enum class  Season

8.观察者模式 Swing

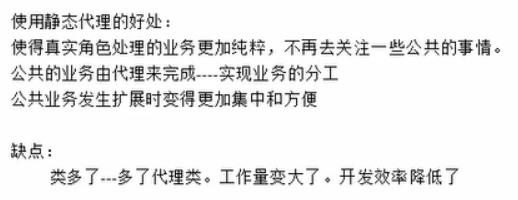
  JavaSE-GUI(Swing)

    监听器(ServletContextListener)

-------------------------------------------------------------------------------------------------------------------------

#静态代理模式的改进：

|  |
| --- |
| 279B89E973BC455E90AC81B70CE4734B.jpg |



01.目标：

    去除冗余代

码，即代码合并；

  (1)增强代码的合并；

  (2)目标对象业务方法调用的合并；

2.特点：

引入InvocationHandler接口（InvocationHandler接口是JavaSE在java.lang.reflect包下定义的一个接口，它的作用是实施调用委派。）

调用委派：是指将任意对象的方法调用委派给接口对象的invoke方法调用，目的是为了代码的合并。

|  |
| --- |
| **public** **class** LogHandler **implements InvocationHandler** {    **private** Object target;    **public** LogHandler() {      }    **public** **void** setTarget(Object target) {  **this**.target = target;      }        @Override      //h.invoke(this,m1,null);  **public** Object invoke(Object proxy, Method method, Object[] args)  **throws** Throwable {          //增强代码  **long** start = System.*currentTimeMillis*();          System.*out*.println(**new** Date(start)+"开始：");            //核心业务功能          //target对象调方法对象的方法（即调method的方法对象），参数为args          Object result = method.invoke(target, args);          //target.addString();          //target.buildString();            //增强代码  **long** end = System.*currentTimeMillis*();          System.*out*.println(**new** Date(end)+"结束,耗时："+(end-start));  **return** result;      }  } |

3.编程方式-方法的调用委派

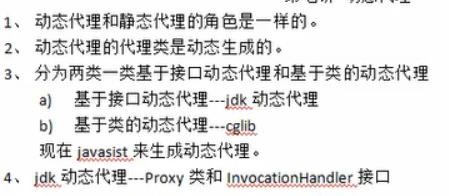
  (1)通过反射获得Method对象；

  (2)将代理方法的调用，委派给InvocationHandler对象的invoke方法;

|  |
| --- |
| //public class $Proxy extends Proxy implements I  **public** **class** StringProcessingProxy **implements** IStringProcessing{  **private** InvocationHandler h; //关键    **private** **static** Method *m1*;  **private** **static** Method *m2*;    **static**{          //①方法的形参列表可以用数组表示          //②new Class[0] 表明是数组的长度为零实际上是没有形参列表，即方法中没有参数  **try** {              //Class.forName("className")根据给定的类名来获得Class对象，用于做类加载              //.getMethod(methodName, new Class[]{paramClazz1, paramClazz2})获得该类的方法所对应的方法对象  *m1* = Class.*forName*("demo.staticproxy.IStringProcessing").getMethod("addString", **new** Class[0]);  *m2* = Class.*forName*("demo.staticproxy.IStringProcessing").getMethod("buildString", **new** Class[0]);          } **catch** (NoSuchMethodException e) {              e.printStackTrace();          } **catch** (ClassNotFoundException e) {              e.printStackTrace();          }      }        //有参构造器，根据传递过来的目标对象创建一个代理对象  **public** StringProcessingProxy(InvocationHandler h) {  **this**.h = h;      }        //proxyObj.addString();      //代理方法的调用委派      @Override  **public** **void** addString() {  **try** {              h.invoke(**this**, *m1*, **null**);          } **catch** (Throwable e) {              e.printStackTrace();          }      }        //proxyObj.buildString      //代理方法的调用委派      @Override  **public** **void** buildString() {  **try** {              h.invoke(**this**, *m2*, **null**);          } **catch** (Throwable e) {              e.printStackTrace();          }      }  } |

-------------------------------------------------------------------------------------------------------------------------

动态代理模式：



       动态代理模式是在静态代理模式改进的基础上，动态的创建代理对象；

如何让JVM动态创建代理对象呢？

      需要使用JavaSE JDK中的Proxy类。Proxy类提供了如下静态方法，用于通知JVM执行时，动态生成代理类（Proxy的子类），并创建代理对象：

Static Object new

ProxyInstance( ClassLoader loader, Class<?>[ ] interfaces, InvocationHandler h)

  ①参数1：动态代理类的类加载器（与接口的类加载器一致）

  ②参数2：动态代理类所实现的接口（与目标类所实现的接口一致）

  ③参数3：表明动态代理类依赖InvocationHandler对象

  ④参数值：动态代理对象

|  |
| --- |
| **public** **class** ProxyFactory{      /\*       \* 静态工厂方法       \*   1)使用接口表示目标对象的类型       \*  2)Proxy类返回一个类型为Object的动态代理对象，动态代理对象的类型也可以用ISomething接口表示       \*/  **public** **static** Object getDynamicProxy(IStringProcessing target){          //①创建InvocationHandler对象          LogHandler logHandler = **new** LogHandler();          //②设置和目标对象的依赖关系          logHandler.setTarget(target);            /\*  newProxyInstance(ClassLoaderloader,Class<?>[]interfaces,InvocationHandler h)           \*   loader 类加载器：（root类加载器，扩展类加载器，应用类加载器）           \*   Class<?>[]interfaces : 目标类可以实现的接口           \*   InvocationHandler h : 是动态代理模式中最重要的，作用(避免代码重复，偷换对象)           \*/          //动态创建代理对象          Object proxyObj = Proxy.*newProxyInstance*(target.getClass().getClassLoader()                                 target.getClass().getInterfaces(),                                   logHandler);  **return** proxyObj;      }  } |

-------------------------------------------------------------------------------------------------------------------------

一、面向切面编程介绍(Aspect Oriented Programming——AOP)

1.什么是AOP？（在不改变原有代码的情况下，增加新的功能）

Spring Aop 就是公共的业务(如日志，安全等)和领域业务结合。当执行领域业务时，将会把公共业务加进来。实现公共业务的重复利用。领域业务更纯粹，程序员专注于领域业务，其本质还是动态代理。

    在传统的软件系统中存在一些象日志记录、事务处理等通用性的代码块，它们通常

    散落在各个程序中，被称之为"横切逻辑"(cross-cutting concern)。

    将这些横切逻辑以面向对象的方式单独进行封装，从而形成切面(aspect)，并在

    程序执行时切入进来的编程方法，被称之为面向切面编程。

2.AOP与OOP的关系

    面向切面编程(AOP)与面向对象编程(OOP)并不矛盾，是在面向对象编程基础之上

    的一种软件开发模式；

3.AOP有几种实现方式？

  (1)AspectJ

     AspectJ是一个基于Java的AOP编程语言，它通常采用ajc编译器，将横切逻辑

          切入到目标类的字节码中，所以被称为静态AOP。

  (2)SpringAOP(重点)

     Spring框架提供了一个AOP的实现机制，这个实现机制是基于动态代理模式的，因此

          被称为动态AOP。Spring框架最初(1.x版)定义了一套自己的接口和类用于AOP编程，

          但在2.x版以后，支持和引入了AspectJ类库，并建议在实际应用中使用。

  (3)JBossAOP

          基于JBoss应用服务器的一种动态AOP的实现。

二、SpringAOP中的常用概念

1.连接点(JoinPoint)

  (1)什么是连接点？

          目标类在执行过程中可以切入增强代码的位置；从理论上讲有静态代码块，

          循环语句块，方法等程序执行位置；

  (2)连接点在Spring中的定义

     Spring框架只支持方法类型的连接点，不支持其它类型；因此在Spring中，

     JoinPoint专指目标对象的执行方法。

2.切入点(Pointcut)

  (1)什么是切入点？

          切入点是连接点的表示方式，可以指定一个连接点或者多个连接点，通常

          使用AspectJ所定义的切入点表达式；

  (2)SpringIoC容器会根据配置，将切入点表达式封装成一个对象；

3.通知(Advice)

  (1)什么是通知？

          通知代表横切逻辑，是切入连接点的增强代码，通常被定义在一个类的方法中

          或者被单独封装到一个类中，比如事务拦截器；

  (2)如何理解通知？

          通知最初是指在控制台中显示字符串的输出语句，用来说明连接点的执行情况，

          是最简单的一种横切逻辑，属于日志记录。复杂的横切逻辑还有：事务处理，

          安全检查等；使用Advice表示横切逻辑，是"一斑窥豹"的命名手法。

4.切面(Aspect)

  (1)什么是切面？

        切面是通知和切入点的组织与封装；SpringIoC容器根据切面，将Advice作用到

          相应的JoinPoint上；因此：Aspect=Advice+Pointcut;

  (2)SpringIoC容器会根据配置创建切面对象；但是切面对象通常依赖于一个普通类，

          即基于POJO的Aspect。

5.织入(weave)

  (1)什么是织入？

     在不侵入目标对象的业务方法的情况下，切入横切逻辑，即将附加的增强代码和

          目标对象的业务方法整合起来。

  (2)如何织入呢？

          通过代理对象；代理对象在其所依赖的InvocationHandler对象的

     invoke方法中将增强代码与业务代码实施整合。

  (3)如何创建代理对象呢？

     Spring的IoC容器通过ProxyFactory，动态产生代理对象。

          由于ProxyFactory封装了复杂的织入过程，因此也被称为织入器(weaver)。

  (4)ProxyFactory产生代理对象的两种方式：

     ①基于JDK(Proxy)的动态代理模式(默认方式)

             特点：目标类实现接口

     ②基于CGLIB(Code Generation Library)的动态字节码生成库

             特点：目标类没有实现接口；

                      通过子类继承，重写目标类的业务方法完成横切逻辑的织入；

-------------------------------------------------------------------------------------------------------------------------

一、AOP编程案例：

1.基于POJO的通知封装

  (1)什么是POJO?

     POJO(Plain Old Java Object)即普通的经典的(传统的)Java对象

  (2)将通知(Advice)按照不同的类型封装为POJO中的方法

2.基于POJO的切面配置

  (1)使用AOP名称空间

|  |
| --- |
| xmlns=*"http://www.springframework.org/schema/beans"*      xmlns:aop=*"http://www.springframework.org/schema/aop"*      xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*      xmlns:p=*"http://www.springframework.org/schema/p"*      xsi:schemaLocation=  *"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans-3.0.xsd*      http://www.springframework.org/schema/aop      http://www.springframework.org/schema/aop/spring-aop-3.0.xsd  *"*> |

 (2)注册POJO

|  |
| --- |
| <!-- 注册切面对象所依赖的对象pojo -->  **<bean id=***"aopbean"* **class=***"aop.aspect.AopBean"***/>** |

 (3)配置切面

          ①注册Pointcut对象；

          ②配置advice对象；

|  |
| --- |
| <!-- 向IoC容器提供面向切面编程的配置 -->  **<aop:config>**          <!-- 定义切入点 -->          <**aop:pointcut** **expression=***"within(aop.service.\*)"* **id=***"servicepointcut"*/>            <!-- 切面(横切逻辑(通知)+切入点) -->  **<aop:aspect** **id**=*"aspectbean"* **ref**=*"aopbean"*>                          <!-- 横切逻辑，前置通知  -->              <**aop:before** **method=***"mybefore"* **pointcut-ref=***"servicepointcut"*/>                 <!-- 正常的后置通知 -->              <**aop:after-returning method**=*"myafterReturning"* **returning**=*"returnVal"*  **pointcut-ref**=*"servicepointcut"*/>                <!-- 异常通知 -->              <**aop:after-throwing method**=*"myafterException"* **throwing**=*"ex"*  **pointcut-ref**=*"servicepointcut"*/>                <!-- 最终通知 -->              <**aop:after method**=*"myafter"* **pointcut-ref**=*"servicepointcut"*/>                <!-- 环绕通知 -->              <**aop:around method**=*"myaround"* **pointcut-ref**=*"servicepointcut"*/>  **</aop:aspect>**  **</aop:config>** |

二、Adivce类型：

通知根据切入连接点的时机和方式，划分为5种类型：

（1）前置通知

           配置标签：<aop:before>

           切入时机 ：在目标对象方法执行之前切入

（2）后置通知

         配置标签： <aop:after-returning>

         切入时机 ：在目标对象方法正常执行完最后一条语句(return语句)时切入

（3）异常通知

           配置标签：<aop:after-throwing>

           切入时机：在目标对象方法执行出现异常时切入

（4）最终通知

           配置标签：<aop:after>

         切入时机：在目标对象方法执行完finally语句块之后切入

------------------------------------------------------------

（5）环绕通知：

           配置标签：<aop:around>

           切入时机：在目标对象方法的执行之前和执行之后切入

           特殊要求: 封装环绕通知的方法必须含有ProceedingJoinPoint类型的参数

       ①ProceedingJoinPoint位于aspectjrt.jar包中；

                ②ProceedingJoinPoint继承于JoinPoint，扩展了连接点的

                    概念，包含了多个环绕通知和连接点，并将它们组成了一个链式结构，

                    其中连接点是这个链式结构的最后一个元素。

                ③ProceedingJoinPoint#proceed()方法，表示下一个链式

                    元素的执行。下一个链式元素可能是另外一个环绕通知或者是JoinPoint。

三、切入点表达式（Pointcut Expression）

（Ⅰ）execution表达式

      ① execution(public \* \*(..))

         说明： (a)(..)表示方法中的零个或多个参数；

             (b)后面紧跟括号的\*，代表方法的名称；

             (c)第一个\*，表示方法的返回值类型；

      ② execution(\* hello\*(..))  // void helloSave()

          符合任何以hello开头的方法

          说明：(a)后面紧跟括号的\*，代表方法的部分名称；

             (b)第一个\*，表示方法的返回值类型；

      ③ execution(\* com.company.IHello.\*(..))

          符合IHello接口/类中声明的任何方法

          说明：(a)后面紧跟括号的\*，代表方法的名称；

             (b)第一个\*，表示方法的返回值类型；

      ④ execution(\* com.company.service.\*.\*(..))

         符合com.company.service包下的所有方法

         说明：(a)后面紧跟括号的\*，代表方法的名称；

            (b)倒数第二颗\*，代表包下的任意类或者是接口；

            (c)第一个\*，表示方法的返回值类型；

      ⑤ execution(\* com.company.service..\*.\*(..))

         符合com.company.service包或者其子包下的任何方法

         说明：(a)后面紧跟括号的\*，代表方法的名称；

           (b)倒数第二颗\*，代表包或者其子包下的任意类或者是接口；

           (c)第一个\*，表示方法的返回值类型；

（Ⅱ）within表达式

   within表达式只接受类型声明，表明了符合要求的类或接口中的所有方法

     ① within(com.company.service.\*)

     说明：符合com.company.service包下的类和接口所定义的方法

     ② within(com.company.service..\*)

     说明：符合com.company.service包下或其子包下类和接口所定义的方法

-------------------------------------------------------------------------------------------------------------------------

 1.目标对象的业务方法的执行

public void oneMethod(){

      statement1-sq1;

      statement2-sq2;

      ...

      statementn;

      return;

}

2.基于四大通知的代理对象的代理方法的执行

<aop:before>

try{

  target.oneMethod();//核心代码

  <aop:after-returning>

}catch(Exception ex){

  <aop:after-throwing>

}finally{

  <aop:after>

}

3.基于事务的代理对象的执行

try{

  Session sess=factory.openSession();

  Transaction tx=sess.beginTransaction();

  target.method();

  tx.commit();

}catch(Exception ex){

  tx.rollback();

}finally{

  sess.close();

}

(三)五大通知的执行顺序

(1)针对目标对象的regist方法

--前置通知--

--arond通知之前，显示执行的对象类型是：service.target.UserServiceImpl，执行的方法是:regist--

  target-regist:注册用户的信息...

--后置通知--:null

--最终通知--

--arond通知之后，显示执行的对象类型是：service.target.UserServiceImpl，执行的方法是:regist--

(2)针对目标对象的modify方法

--前置通知--

--arond通知之前，显示执行的对象类型是：service.target.UserServiceImpl，执行的方法是:modify--

  target-modify:修改用户的信息...

--后置通知--:200

--最终通知--

--arond通知之后，显示执行的对象类型是：service.target.UserServiceImpl，执行的方法是:modify--

(3)针对目标对象的delete方法(注意:环绕通知的后半部分没有执行)

--前置通知--

--arond通知之前，显示执行的对象类型是：service.target.UserServiceImpl，执行的方法是:delete--

  target-delete:删除用户的信息...

--产生异常通知/ by zero--

--最终通知--

Exception in thread "main" java.lang.ArithmeticException: / by zero

      at service.target.UserServiceImpl.delete(UserServiceImpl.java:28)

-------------------------------------------------------------------------------------------------------------------------

一、事务概念（复习）Transaction(交易)

(1)什么是事务？

      对应底层的一组sql语句，形成一个不可分割的原子操作；如果这组

   sql语句有一条执行失败，则一组的操作都被视为是失败的。

(2)如何构建事务？

      构建事务，需要如下核心结构：(TransactionInterceptor)

   try{

     tx.begin;//通过HibernateTransactionManager开启事务

     sql1;//save(user)

     sql2;//save(user2)

     ...

     sqln;

     tx.commit; //HibernateTransactionManager#commit();

   }catch(Exception ex){

      tx.rollback;//HibernateTransactionManager#rollback();

   }finally{

     //关闭资源

   }

二、Spring框架对事务的管理

（1）在配置文件中，添加"tx"名称空间，并指明该空间下的

        约束定义文件（xsd）

（2）配置事务的横切逻辑

    ①使用<tx:advice>对应于Spring中TransactionInterceptor

    ②事务的横切逻辑又依赖于TransactionManager的具体对象，

             该对象提供支持事务的连接对象，及事务的边界语句

       (i)事务对象；

       (ii)commit()、rollback()语句

    ③配置事务属性，主要指明事务的传播策略：

    <tx:attributes>

（3）配置事务的切面（advice+pointcut）：

      ①使用<aop:config>标签

      ②切面的标签：<aop:aspect>

                 <aop:advisor>专用于事务

      ③配置顺序是固定的：

      <aop:config>

            ①<aop:pointcut /> 切入点

            ②<aop:advisor />

            ③<aop:aspect />

      </aop:config>

（4）事务属性：隔离性、传播性(propagation)、只读性等

       事务的传播策略

      ①REQUIRED (\*)

        支持当前事务，

        如果当前没有事务，则创建新事务

      ②SUPPORTS (\*查询)

        支持当前事务，

        如果当前没有事务，则不创建新事务

      ③MANDATORY

       支持当前事务；

       如果当前没有事务，则抛异常

      ④REQUIRES-NEW

       如果当前没有事务，则创建一个新事物；

       如果当前有事务，则将当前事务挂起，再创建一个新的事务；

      ⑤NOT-SUPPORTED

       以非事务方式执行；

       如果当前有事务，则将当前事务挂起。

      ⑥NEVER

       永远不需要事务，即以非事务方式执行；

      ⑦NESTED

       如果当前有事务，则在事务内再创建事务；

       如果当前没有事务，则创建事务。

三 Spring框架的事务管理方式:

  (1)编程式事务;

  (2)声明式事务(①基于xml配置②基于注解配置)

-------------------------------------------------------------------------------------------------------------------------

（一）Spring框架整合Hibernate:

（1）配置方面

1.Spring框架用LocalSessionFactoryBean提供SessionFactory

    对象，而不是Hibernate框架生成的HibernateSessionFactory。

2.LocalSessionFactoryBean以连接池作为自己的数据源

（2）编程方面

Spring框架提供了HibernateDaoSupport，通常将它作为DAO组件

开发的父类；HibernateDaoSupport有两个重要的属性：

①SessionFactory sessionFactory;

②HibernateTemplate template;

（二）目前常用的数据库连接池产品

连接池是一个集合结构（比如：ArrayList/Vector），在集合

结构中存储若干个连接对象（已经连接上数据库），并随着访问

数据库的并发强度，来动态的调整连接池中连接对象的数量。

（1）c3p0（Hibernate默认使用的连接池技术）

（2）dbcp  （Apache开源组织提供）

（3）proxool （开源产品）

（三）c3p0作为数据源

参数说明：

initialPoolSize:连接池初始化时所创建的连接对象的数量；

maxPoolSize:最大连接对象的数量

minPoolSize:最小连接对象的数量

maxIdleTime:如果在设置时间内(秒)，连接对象没有被使用就被释放掉；

checkoutTimeout:获得连接所需的最长时间(毫秒)，如果超过这个时间会抛异常

（四）HibernateDaoSupport

（1）HibernateTemplate属性

（2）SessionFactory属性

（五）ApplicationContext.xml配置

（1）连接工厂的配置

   LocalSessionFactoryBean

（2）组件依赖关系的设定

UserServiceImpl-->HibernateUserDAO-->LocalSessionFactoryBean-->c3p0

（3）切面配置

-------------------------------------------------------------------------------------------------------------------------

1.Spring框架与Struts2整合

（1）在web工程中利用MyEclipse添加类库:

    build path->MyEclipse->Struts 2 Spring libraries

        在该库中有如下两个重要的jar包:

        ①从Spring的角度，需要spring-web.jar;

           核心类:

           org.springframework.web.context.ContextLoaderListener

           继承:

           public class ContextLoaderListener extends ContextLoader

                        implements ServletContextListener

           目的:

                   使web容器能够加载applicationContext.xml;加载后，Spring框架

                   创建了SpringIoC容器，管理业务逻辑组件的创建、依赖关系及事务处理等；

        ②从struts2的角度，需要struts-spring-plugin.jar插件

           核心类：

           org.apache.struts2.spring.StrutsSpringObjectFactory

                                     (插件容器)

           目的：

                    使用插件容器(StrutsSpringObjectFactory)代替Struts2默认容器

          (StrutsObjectFactory)，插件容器不仅可以创建Action对象，还可以

                    访问SpringIoC容器，为Action对象自动寻找和绑定业务组件，注入依赖关系。

           两种用法：

                    ①访问Spring容器，为Action对象的依赖属性实施依赖注入；

         --------------------------------------------------------

                    ②由SpringIoC容器负责所有对象的创建，包括Action对象的创建；

                       插件容器只负责根据用户的请求，去springIoC容器中查找相应的控制器；

（2）在struts.xml文件中，配置Action自动装配的方式：

     ①默认是byName

            因为在struts框架->struts2-core-2.1.8.1.jar->org.apache.struts2->

        default.properties文件中的设置是：

        struts.objectFactory.spring.autoWire = name

       ②通过<constant>标签设置成byTye方式:

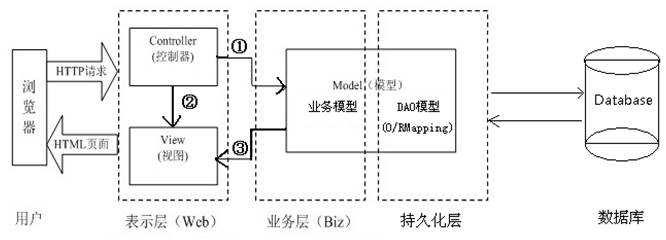
         <constant name="struts.objectFactory.spring.autoWire" value="type" />

【总结】

S2SH的整合，即Struts2与Spring及Hibernate的整合

（1）Spring先和Hibernate整合，并测试成功；

（2）struts2再与整合好的SH进行整合；



-------------------------------------------------------------------------------------------------------------------------

#JDK中的Annotation

一、注解的概念

1.注解是JavaJDK1.5引入的一种新的数据类型，它的本质其实是一个特殊的接口。这种特殊性在于：

  (1)直接继承java.lang.anotation.Annotation接口，这意味着注解接口没有子接口；

  (2)表现形式：public @interface Xxx

     @的含义是： public interfac Xxx extends Annotation

  (3)注解接口不需要程序员实现，因为JVM在程序运行时为注解接口动态产生代理类及其对象；

2.注解的定义形式:

  public @interface AnnotationName{

      元素声明1;

      元素声明2;

      ...

  }

  其中，元素声明通常有如下两种形式：

  (1)  元素类型 元素名称()

  (2)  元素类型 元素名称() default value;

3.注解接口中的元素实质：

  (1)注解接口中的元素声明实际上是方法的声明；

  (2)注解接口中的方法没有任何参数；

  (3)注解接口中的方法有返回值，可以通过default value，为方法指定默认的返回值；

4.注解元素的类型为下列之一：

  √(1)一个基本类型(int short long byte char double float boolean);

  √(2)String类型

  --------------

  √(3)enum类型

  --------------

   (4)Class类型

   (5)注解类型

  √(6)由上述类型组成的数组；

5.举例：注解接口定义：

  public @interface BugReport{

         String assignedTo() default "none";

         int severity() default 0;

  }

二、注解的使用

1.注解作为修饰符用来标注程序元素；

2.程序元素在java.lang.annotation包下的ElementType枚举类有明确的定义：

  (1)PACKAGE                   包

 √(2)TYPE                      类和普通接口

  (3)ANNOTATION\_TYPE           注解接口

 √(4)FIELD                     成员变量

 √(5)CONSTRUCTOR               构造器

 √(6)METHOD                    方法

  (7)PARAMETER                 方法参数

  (8)LOCAL\_VARIABLE            方法局部变量

3.注解的标注格式

  @AnnotationName(元素名称1=值1,元素名称2=值2,...)

  举例：

  @BugReport(assignedTo="Wangda", severity=10);

  如果某个元素在使用时未指明，则使用声明时的默认值，例如：

  @BugReport(assignedTo="Lifen");

  元素serverity的值是0。

  @BugReport

4.注解使用的进一步说明：

  (1)标记型注解，

     例如: @BugReport

     如下两种情况：

         ①注解接口中可能没有定义任何元素；

         ②注解使用元素的默认值

  (2)单值注解的简洁格式

     如果注解中的元素名称是value，且没有其它元素，则使用时可以忽略元素的名称及等号。

     例如：一个注解接口如下：

     public @interface ActionListenerFor{

               String value();

     }

     使用时，可以有两种形式：

     @ActionListernerFor(value="yelloButton")

     或者：

     @ActionListenerFor("yellowButton")

  (3)注解元素值是一个数组

     例如：

        public @interface BugReport{

                        String[] reportedBy();

             }

             使用时，需要使用{}表示数组值

             @BugReport(reportedBy={"wangda","qiangui"})

             如果数组中只有一个单值，也可以省略{}

             @BugReport(reportedBy="Joe")等价于@BugReport(reportedBy={"Joe"})

三、JDK中所定义的注解

[①元注解②规则注解③和依赖注入相关的注解]

1.元注解(位于java.lang.annotation包下)

 (Ⅰ)什么是元注解？

         元注解是用来标注注解的注解！

 (Ⅱ)为什么要使用元注解？

         说明注解的使用方式，明确如下问题：

         ①注解标注哪些程序元素？

         ②注解的保留到哪一个阶段？

         ③注解是否进入文档？

         ④注解是否被继承？

 (Ⅲ)JDK定义了哪些元注解？

  (1)@Documented

          ①用途

             注解进入文档，JavaDoc文档工具可以将注解提取到文档中。

          ②定义

       @Documented

       @Retention(RetentionPolicy.RUNTIME)

       @Target(ElementType.ANNOTATION\_TYPE)

       public @interface Documented {

       }

  (2)@Inherited

          ①用途

             继承的子类自动拥有父类的注解；

          ②定义

       @Documented

       @Retention(RetentionPolicy.RUNTIME)

       @Target(ElementType.ANNOTATION\_TYPE)

       public @interface Inherited {

       }

  (3)@Retention(enumValue)

          ①用途

             指明程序中的注解保留到哪一个阶段；有三个阶段:source、class和runtime,

             被称为保留策略。

          ②注解的保留策略

       RetentionPolicy是枚举类，在该类中定义了注解的3种保留策略:

       (a)RUNTIME √

                    注解保留到程序运行时，JVM通过反射产生注解的代理对象；

       (b)CLASS

                    注解保留到编译期，编译器根据注解标注，判断是否显示警告；

                    但JVM不载入，因此运行时无效；

       (c)SOURCE

                    注解只保留在源码中，在class文件中不存在；

           ③定义

       @Documented

       @Retention(RetentionPolicy.RUNTIME)

       @Target(ElementType.ANNOTATION\_TYPE)

       public @interface Retention {

              RetentionPolicy value();

       }

   (4)@Target({enumValue1, enumValue2, ...})

            ①用途

               指明注解所标注的程序元素，程序元素在ElementType枚举类中进行了定义，

               参考前面

            ②定义

        @Documented

            @Retention(RetentionPolicy.RUNTIME)

            @Target(ElementType.ANNOTATION\_TYPE)

            public @interface Target {

              ElementType[] value();

            }

2.规则注解(位于java.lang包下)：

  (1)@Override

          ①定义：

       @Target(value=METHOD)

       @Retention(value=SOURCE)

       public @interface Override

          ②用途：

             如果方法利用此注解进行标注但没有重写超类方法，则编译器会生成一条错误消息。

  (2)@Deprecated

          ①定义：

       @Documented

       @Retention(value=RUNTIME)

       public @interface Deprecated

          ②用途：

             用 @Deprecated 所标注的程序元素，不鼓励程序员使用;如果强行使用,编译器会发出警告。

             这个注解与Java文档标签@deprecated具有同等的功效。

|  |
| --- |
| **@Deprecated**         public int getAge() {                return age;         } |

  (3)@SuppressWarnings({"value1","value2",...})

          ①定义

     @Target(value={TYPE,FIELD,METHOD,PARAMETER,CONSTRUCTOR,LOCAL\_VARIABLE})

     @Retention(value=SOURCE)

     public @interface SuppressWarnings{

          String[] value()

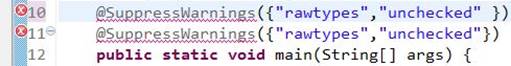
     }

          ②用途：

             抑制编译器所产生的警告信息

|  |
| --- |
| **@SuppressWarnings**(value="**rawtypes**")      //抑制使用原类型警告         public static List cache = new ArrayList();    **@SuppressWarnings**(value = "**unchecked**")   //抑制类型检查警告         public void add(String data) {                cache.add(data);         }    **@SuppressWarnings**({"**rawtypes**","**unchecked**"})  **public** **static** **void** main(String[] args) {          Demo d=**new** Demo();          d.setVar("wangda" );          System.*out*.println("信息:"+d.getVar());      }            @SuppressWarnings({"serial"})  **public** **class** SuppressWarningsTest3 **implements** Serializable {  //  public static final long serialVersionUID=123456789012l; |

【注】 不能同时使用两次，否则会报错



3.和依赖注入相关的注解(位于javax.annotation包下)

  (1)@Resource(name="value1")-------->（利用JDK注解设置依赖关系）

     ①定义：

       @Target(value={TYPE,FIELD,METHOD})

         @Retention(value=RUNTIME)

         public @interface Resource{

         √ String name() default ""

           Class type() default Object.class

       }

     ②用途——实施依赖注入

   (a)标注成员变量，

          注入的对象，默认名称为字段名；

   (b)标注方法(参考spring06\_hibernate\_tran2->anno.dao.HibernateUserDao.java)

          通过方法参数注入的对象，默认名称为JavaBean的属性名称；

   (c)标注类

          没有默认值，必须指明所注入对象的名称

     ③在Spring框架中使用

    Spring将@Resource注解的name元素的值解析为bean的名字，而type元素的值解析为bean的类型。

        所以如果使用name元素，则使用byName的自动注入策略，而使用type元素时则使用byType自动注入策略。

        默认情况下使用byName自动注入策略。

  (2)@PostConstruct

     ①定义

       @Documented

       @Retention(value=RUNTIME)

       @Target(value=METHOD)

       public @interface PostConstruct

     ②用途：

       参照：spring04\_annotation->anno.demo3.UseBean.java

  (3)@PreDestroy

     ①定义：

       @Documented

       @Retention(value=RUNTIME)

       @Target(value=METHOD)

       public @interface PreDestroy

      ②用途：

      在容器销毁对象前所执行的方法，目的是为了释放资源。要求被标注的方法不能有参数，而且无返回值。

-------------------------------------------------------------------------------------------------------------------------

#Spring注解

一、Spring框架定义的注解

分类：组件、依赖注入、容器管理、事务

1.组件对象(位于org.springframework.stereotype包下)；

                                ==========

  (1)@Component (通用性组件)

     ①定义：

        @Target(value=TYPE)

       @Retention(value=RUNTIME)

       @Documented

       public @interface Component{

          String value();

       }

     ②用途：

       (a)通用性的标注，可以标注任意类

       (b)所标注的类产生指定名称的对象；对象的默认名称为所标注类的类名首字母小写

  (2)@Controller (组件进一步细分之控制器)

     ①定义：

       @Target(value=TYPE)

       @Retention(value=RUNTIME)

       @Documented

       @Component //说明了该注解实质上是Component

       public @interface Controller{

           String value();

       }

     ②应用：

       (a)专用性标注，通常标注控制器

       (b)所标注的类产生指定名称的对象；对象的默认名称为所标注类的类名首字母小写

  (3)@Service (组件进一步细分之业务组件)

     ①定义：

       @Target(value=TYPE)

       @Retention(value=RUNTIME)

       @Documented

       @Component

       public @interface Service{

              String value();

       }

     ②应用：

       (a)专用性标注，通常标注业务组件

       (b)所标注的类产生指定名称的对象；对象的默认名称为所标注类的类名首字母小写

  (4)@Repository (组件进一步细分之DAO组件)

     ①定义：

       @Target(value=TYPE)

       @Retention(value=RUNTIME)

       @Documented

       @Component

       public @interface Repository{

            String value();

       }

      ②应用：

       (a)专用性标注，通常标注DAO组件

       (b)所标注的类产生指定名称的对象；对象的默认名称为所标注类的类名首字母小写

2.依赖注入(位于org.springframework.beans.factory.annotation包下)：

                                      ==================

  (1)@Autowired

     ①定义：

       @Retention(value=RUNTIME)

       @Target(value={CONSTRUCTOR,FIELD,METHOD})

       public @interface Autowired{

           boolean required() default true

       }

     ②应用：

       实施自动绑定。如果不提供@Qualifier("beanName"),将按照类型实施绑定；

  (2)@Qualifier("beanName")

          ①定义：

       @Retention(value=RUNTIME)

         @Target(value={FIELD,METHOD,PARAMETER,TYPE,ANNOTATION\_TYPE})

         @Inherited

         @Documented

       public @interface Qualifier{

         String value default ""

       }

          ②应用：

            按照标注所指定的名称，实施自动绑定。

3.容器管理(位于org.springframework.context.annotation包下)：

                                ===================

  (1)@Scope

          ①定义：

       @Target(value={TYPE,METHOD})

       @Retention(value=RUNTIME)

       @Documented

       public @interface Scope{

          String value() default "singleton" ;

       }

          ②应用：

             指明被标注的类所产生的对象是:单例对象还是多例对象

  (2)@Lazy

          ①定义：

       @Target(value={TYPE,METHOD})

         @Retention(value=RUNTIME)

         @Inherited

         @Documented

         public @interface Lazy{

               boolean value() default true;

         }

       ②应用：

           指明Spring容器是否延迟创建对象，默认是延迟

4.事务(位于org.springframework.transaction.annotation包下)：

                              ======================

  (1)@Transactional

          ①定义

       @Target({ElementType.METHOD，ElementType.TYPE})

       @Retention(RetentionPolicy.RUNTIME)

       @Inherited

       @Documented

       public @interface Transactional{

       \*Propagation propagation() default Propagation.REQUIRED;

          Isolation isolation() default Isolation.DEFAULT;//由底层的数据库决定

          int timeout() default TransactionDefinition.TIMEOUT\_DEFAULT;

        \*boolean readOnly() default false;

         Class<? extends Throwable>[] rollbackFor() default {};

         String[] rollbackForClassName() default {};

         Class<? extends Throwable>[] noRollbackFor() default {};

         String[] noRollbackForClassName() default {};

       }

            ②标注类，标注方法

               参考案例：Spring06\_hibernate06\_tran2->anno.service.UserServiceImpl.java

   (2)enum Isolation所定义的常量

      DEFAULT

          Use the default isolation level of the underlying datastore.

        READ\_COMMITTED

          A constant indicating that dirty reads are prevented; non-repeatable

          reads and phantom reads can occur.

        READ\_UNCOMMITTED

          A constant indicating that dirty reads, non-repeatable reads and

          phantom reads can occur.

        REPEATABLE\_READ

          A constant indicating that dirty reads and non-repeatable reads are

          prevented; phantom reads can occur.

        SERIALIZABLE

          A constant indicating that dirty reads, non-repeatable reads and

          phantom reads are prevented.

   (3)enum Propagation所定义的常量

      MANDATORY

          Support a current transaction, throw an exception if none exists.

        NESTED

          Execute within a nested transaction if a current transaction exists,

          behave like PROPAGATION\_REQUIRED else.

        NEVER

          Execute non-transactionally, throw an exception if a transaction

          exists.

        NOT\_SUPPORTED

          Execute non-transactionally, suspend the current transaction if one

          exists.

        REQUIRED

          Support a current transaction, create a new one if none exists.

        REQUIRES\_NEW

          Create a new transaction, suspend the current transaction if one

          exists.

        SUPPORTS

          Support a current transaction, execute non-transactionally if none

          exists.

   (4)Spring的事务抽象，包括3个主要接口：

           ①PlatformTransactionManager

              负责界定事务边界

           ②TransactionDefinition

              定义了:

      (a)事务的传播行为；

      (b)事务的只读提示；

      (c)事务的隔离级别；

      (d)事务的超时设定

          ③TransactionStatus定义了事务的状态

二、AspectJ所定义的注解

1.切面组件(Aspect)：

    ①定义：

    @Retention(value=RUNTIME)

      @Target(value=TYPE)

      public @interface Aspect{

             String value() default "";

      }

    ②应用：

       用来标注类的，指明该类对象是切面对象

2.切入点(Pointcut)：

    ①定义：

     @Retention(value=RUNTIME)

      @Target(value=METHOD)

      public @interface Pointcut{

            String value() default ""; //指明Pointcut表达式

    }

   ②应用：

   (a)@Pointcut是方法级别的注解，因此不能脱离某个方法单独声明；

   (b)为@Pointcut定义的一个空方法是：Pointcut Signature。

          在Advice中可以通过该方法签名引用切入点表达式。参考案例：

        spring04\_aop2\_anno2->anno.aop.OptLogger.java

3.通知(Advice):

  (1)BeforeAdvice

         ①定义：

       @Retention(value=RUNTIME)

         @Target(value=METHOD)

         public @interface Before{

               String value() default "";

         }

         ②应用：

           标注方法，指明该方法定义了前置通知的横切逻辑

  (2)AfterReturningAdvice

          ①定义：

       @Retention(value=RUNTIME)

         @Target(value=METHOD)

       public @interface AfterReturning{

          String value() default "";//用于默认指定切入点表达式

        √ String pointcut() default ""; //用于显示指定切入点表达式

        √ String returning() default "";//引用目标对象的返回值；

       }

         ②应用：

            标注方法，指明该方法定义了后置通知的横切逻辑

  (3)AfterThrowingAdvice

         ①定义

       @Retention(value=RUNTIME)

       @Target(value=METHOD)

       public @interface AfterThrowing{

          String value() default "";//用于默认指定切入点表达式

        √ String pointcut() default ""; //用于显示指定切入点表达式

        √ String throwing() default "";//引用目标对象执行方法所抛出的异常对象

       }

         ②应用：

            标注方法，指明该方法定义了异常通知的横切逻辑

  (4)AfterAdvice

          ①定义

       @Retention(value=RUNTIME)

         @Target(value=METHOD)

         public @interface After{

               String value() default "";

         }

          ②应用：

             标注方法，指明该方法定义了最终通知的横切逻辑

  (5)AroundAdvice

          ①定义

       @Retention(value=RUNTIME)

         @Target(value=METHOD)

       public @interface Around{

          String value() default "";

       }

         ②应用：

            标注方法，指明该方法定义了环绕通知的横切逻辑

三、使用Spring注解时xml文件的配置：

  (1)<context:component-scan base-package="...">

         ①扫描指定包下的所有类，寻找标注了相应注解的类，并添加到IoC容器中；

         ②扫描的注解类型是:

       @Component及其衍生的细化类型@Controller、@Service、@Repository；

         ③【注意】！！

             扫描的注解类型不包括@Aspect，因此需要使用单独的bean标签注册@Aspect标注的类

|  |
| --- |
| <beans  xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:**context**=*"http://www.springframework.org/schema/***context***"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=  *"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans-2.5.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/***context***/spring-***context***-2.5.xsd"*  >      <!-- 扫描指定包下程序，根据注解创建对象，并注册到容器中 -->  **<context:component-scan** **base-package**=*"anno.demo2"* />  </beans> |

  (2)<aop:aspectj-autoproxy />

          在spring容器中，自动创建切面并为目标对象创建代理对象。

  (3)<tx:annotation-driven transaction-manager="txManager" />

          事务的注解驱动，需要依赖于PlatformTransactionManager

-------------------------------------------------------------------------------------------------------------------------

总结：

1.利用Spring注解创建对象

  @Component()

  @Component(value="xxx")

  @Component("xxx")

2.利用Spring注解设置依赖关系（\*）

  @Autowired

  @Qualifier("xxx")

3.利用Spring注解设置IoC容器的对象提供方式：

  @Scope("singleton")

  @Scope("prototype")

4.利用JDK注解指定初始化方法和销毁对象时执行的方法

  @PostConstruct

  @PreDestroy

|  |
| --- |
| **@Component**  //所标注的类产生指定名称的对象  **@Scope**("prototype") //销毁对象前调用方法，只适用于singleton模式）  **@Lazy**   // 指明Spring容器是否延迟创建对象，默认是延迟  **public** **class** UseBean {        //**@Resource**(name="zhHelloBean")  //如果使用name元素，则使用byName的自动注入策略，而使用type元素时则使用byType自动注入策略  **@Autowired**  //按类型自动装配  **@Qualifier**("zhHelloBean")  //按名称自动装配  **private** HelloBean hello;    **@PostConstruct**   //指定初始化方法  **public** **void** myinit(){          System.*out*.println("init....");      }  **@PreDestroy**  //销毁对象时执行(方法为无参无返回值)  **public** **void** mydestroy(){          System.*out*.println("destroy.....");      }  } |