## 什么是控制反转IOC(Inversion of control):

应用本身不负责依赖对象的创建和维护，脐带对象的创建及是由外部容器负责的，这样控制权就由应用转移到了外部的容器，控制权的转移就是所谓的反转。

## 什么是依赖注入DI（Dependency Injection）

在运行期，由外部容器动态地将依赖对象注入到组件中

## 使用Spring的好处/优点

* 降低组件之间的耦合度,实现软件各层之间的解耦。
* 可以使用容器提供的众多服务，如：事务管理服务、消息服务等等。当我们使用容器管理事务时，开发人员就不再需要手工控制事务.也不需处理复杂的事务传播。
* 容器提供单例模式支持，开发人员不再需要自己编写实现代码。
* 容器提供了AOP技术，利用它很容易实现如权限拦截、运行期监控等功能。
* 容器提供的众多辅作类，使用这些类能够加快应用的开发，如： JdbcTemplate、 HibernateTemplate。
* Spring对于主流的应用框架提供了集成支持，如：集成Hibernate、JPA、Struts等，这样更便于应用的开发。

## 轻量级与重量级概念的划分

轻量级还是重量级,主要看它使用了多少服务.使用的服务越多,容器要为普通java对象做的工作就越多,必然会影响到应用的发布时间或者是运行性能

对于spring容器，它提供了很多服务，但这些服务并不是默认为应用打开的，应用需要某种服务，还需要指明使用该服务，如果应用使用的服务很少，如:只使用了spring核心服务，那么我们可以认为此时应用属于轻量级的，如果应用使用了spring提供的大部分服务，这时应用就属于重量级。目前EJB容器就因为它默认为应用提供了EJB规范中所有的功能，所以它属于重量级。

## 实例化spring容器的两种方式：

方法一:

在类路径下寻找配置文件来实例化容器

ApplicationContext *ctx =* **new ClassPathXmlApplicationContext(new String[]{"beans.xml"});**

方法二:

在文件系统路径下寻找配置文件来实例化容器

ApplicationContext *ctx =* **new FileSystemXmlApplicationContext(new String[]{“d:\\beans.xml“});**

**Spring的配置文件可以指定多个，可以通过String数组传入。**

## 三种实例化bean的方式

1. 使用类构造器实例化

|  |
| --- |
| <bean id=“orderService" class="cn.itcast.OrderServiceBean"/> |

1. 使用静态工厂方法实例化

|  |
| --- |
| <bean id="personService" class="cn.itcast.service.OrderFactory" **factory-method="createOrder"**/> |

|  |
| --- |
| public class OrderFactory {  public **static** OrderServiceBean createOrder(){  return new OrderServiceBean();  }  } |

1. 使用实例工厂方法实例化

|  |
| --- |
| <bean id="personServiceFactory" class="cn.itcast.service.OrderFactory"/>  <bean id="personService" factory-bean="personServiceFactory" factory-method="createOrder"/> |

|  |
| --- |
| public class OrderFactory {  public OrderServiceBean createOrder(){  return new OrderServiceBean();  }  } |

## Bean的作用域

1. Singleton

在每个Spring IoC容器中一个bean定义只有一个对象实例。默认情况下会在容器启动时初始化bean，但我们可以指定Bean节点的lazy-init=“true”来延迟初始化bean，这时候，只有第一次获取bean会才初始化bean。如：

<bean id="xxx" class="cn.itcast.OrderServiceBean" lazy-init="true"/>

如果想对所有bean都应用延迟初始化，可以在根节点beans设置default-lazy-init=“true“，如下：

<beans default-lazy-init="true“ ...>

1. Prototype
2. Request
3. Session
4. Global session

## 指定Bean的初始化方法和销毁方法

指定Bean的初始化方法和销毁方法

<bean id="xxx" class="cn.itcast.OrderServiceBean" init-method="init" destroy-method="close"/>

## 注入依赖对象

**基本类型对象注入：**

<bean id="orderService" class="cn.itcast.service.OrderServiceBean">

<constructor-arg index=“0” type=“java.lang.String” value=“xxx”/>//构造器注入

<property name=“name” value=“zhao/>//属性setter方法注入

</bean>

**注入其他bean：**

**方式一**

<bean id="orderDao" class="cn.itcast.service.OrderDaoBean"/>

<bean id="orderService" class="cn.itcast.service.OrderServiceBean">

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

</bean>

**方式二(使用内部bean,但该bean不能被其他bean使用)**

<bean id="orderService" class="cn.itcast.service.OrderServiceBean">

<property name="orderDao">

<bean class="cn.itcast.service.OrderDaoBean"/>

</property>

</bean>

## 集合类型的装配

|  |
| --- |
| **<bean id="order" class="cn.itcast.service.OrderServiceBean">**  **<property name="lists">**  **<list>**  **<value>lihuoming</value>**  **</list>**  **</property>**  **<property name="sets">**  **<set>**  **<value>set</value>**  **</set>**  **</property>**  **<property name="maps">**  **<map>**  **<entry key="lihuoming" value="28"/>**  **</map>**  **</property>**  **<property name="properties">**  **<props>**  **<prop key="12">sss</prop>**  **</props>**  **</property>**  **</bean>** |

## 依赖注入

* 使用构造器注入
* 使用属性setter方法注入
* 使用Field注入（用于注解方式）

注入依赖对象可以采用手工装配或自动装配，在实际应用中建议使用手工装配，因为自动装配会产生未知情况,开发人员无法预见最终的装配结果。

1.手工装配依赖对象

2.自动装配依赖对象

## 依赖注入--手工装配

**手工装配依赖对象，在这种方式中又有两种编程方式**

1. 在xml配置文件中，通过在bean节点下配置，如

<bean id="orderService" class="cn.itcast.service.OrderServiceBean">

<constructor-arg index=“0” type=“java.lang.String” value=“xxx”/>//构造器注入

<property name=“name” value=“zhao/>//属性setter方法注入

</bean>

2. 在java代码中使用@Autowired或@Resource注解方式进行装配。但我们需要在xml配置文件中配置以下信息：

<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-2.5.xsd

**http://www.springframework.org/schema/context**

**http://www.springframework.org/schema/context/spring-context-2.5.xsd**">

**<context:annotation-config/>**

</beans>

这个配置隐式注册了多个对注释进行解析处理的处理器:AutowiredAnnotationBeanPostProcessor，CommonAnnotationBeanPostProcessor，PersistenceAnnotationBeanPostProcessor，RequiredAnnotationBeanPostProcessor

注： @Resource注解在spring安装目录的lib\j2ee\common-annotations.jar

## 依赖注入--手工装配

**在java代码中使用@Autowired或@Resource注解方式进行装配，这两个注解的区别是：@Autowired 默认按类型装配，@Resource默认按名称装配，当找不到与名称匹配的bean才会按类型装配。**

@Autowired

private PersonDao personDao;//用于字段上

@Autowired

public void setOrderDao(OrderDao orderDao) {//用于属性的setter方法上

this.orderDao = orderDao;

}

@Autowired注解是按类型装配依赖对象，默认情况下它要求依赖对象必须存在，如果允许null值，可以设置它required属性为false。如果我们想使用按名称装配，可以结合@Qualifier注解一起使用。如下：

**@Autowired @Qualifier("personDaoBean")**

private PersonDao personDao;

@Resource注解和@Autowired一样，也可以标注在字段或属性的setter方法上，但它默认按名称装配。名称可以通过@Resource的name属性指定，如果没有指定name属性，当注解标注在字段上，即默认取字段的名称作为bean名称寻找依赖对象，当注解标注在属性的setter方法上，即默认取属性名作为bean名称寻找依赖对象。

@Resource(name=“personDaoBean”)

private PersonDao personDao;//用于字段上

**注意：如果没有指定name属性，并且按照默认的名称仍然找不到依赖对象时， @Resource注解会回退到按类型装配。但一旦指定了name属性，就只能按名称装配了。**

**另外@Resource是J2EE里面的注解，而@Autowired是Spring里的注解，从解耦的角度来看，推荐使用前者**

## 依赖注入--自动装配依赖对象

对于自动装配，大家了解一下就可以了，实在不推荐大家使用。例子：

<bean id="..." class="..." autowire="byType"/>

autowire属性取值如下：

* byType：按类型装配，可以根据属性的类型，在容器中寻找跟该类型匹配的bean。如果发现多个，那么将会抛出异常。如果没有找到，即属性值为null。
* byName：按名称装配，可以根据属性的名称，在容器中寻找跟该属性名相同的bean，如果没有找到，即属性值为null。
* 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。
* 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。

## 通过在classpath自动扫描方式把组件纳入spring容器中管理

前面的例子我们都是使用XML的bean定义来配置组件。在一个稍大的项目中，通常会有上百个组件，如果这些这组件采用xml的bean定义来配置，显然会增加配置文件的体积，查找及维护起来也不太方便。spring2.5为我们引入了组件自动扫描机制，他可以在类路径底下寻找标注了@Component、@Service、@Controller、@Repository注解的类，并把这些类纳入进spring容器中管理。它的作用和在xml文件中使用bean节点配置组件是一样的。要使用自动扫描机制，我们需要打开以下配置信息:

<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-2.5.xsd

**http://www.springframework.org/schema/context**

**http://www.springframework.org/schema/context/spring-context-2.5.xsd**">

**<context:component-scan base-package="cn.itcast"/>**

</beans>

其中base-package为需要扫描的包(含子包)。

@Service用于标注业务层组件、 @Controller用于标注控制层组件（如struts中的action）、@Repository用于标注数据访问组件，即DAO组件。而@Component泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注。

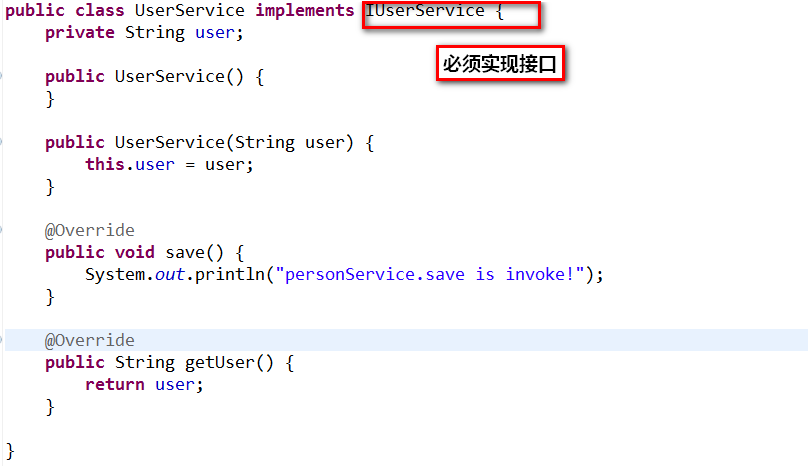
到Spring2.5.6为止，Spring并且没有对这四个注解作特别的处理，即是这四个注解都是一样的，都会把对应的类进行自己的配置，但不排除以后会特别处理。

## AOP

AOP在企业中多用于权限和日记记录模块

### 动态代理的JDK实现方式

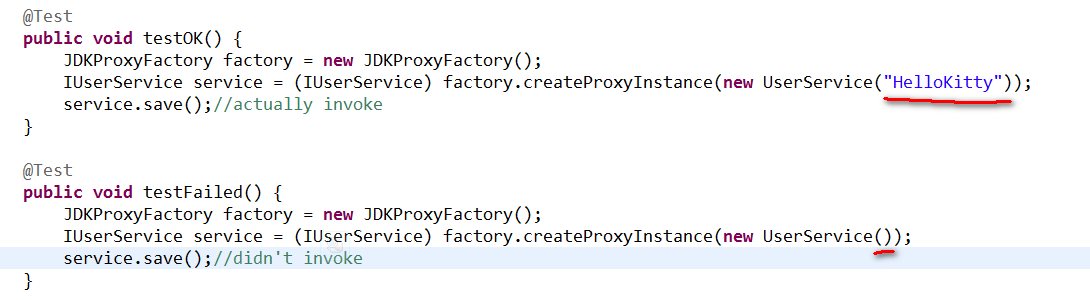
**Service:**



**Proxy factory:**



**Test class:**

****

**Note:**

1. The proxy target must have implemented from interface!

### CGLIB的AOP实现方式



AOP的概念：

**Aspect(切面)**:指横切性关注点的抽象即为切面,它与类相似,只是两者的关注点不一样,类是对物体特征的抽象,而切面横切性关注点的抽象.（类JDKProxyFactory就可以理解为一个切面）

**joinpoint(连接点)**:所谓连接点是指那些被拦截到的点。在spring中,这些点指的是方法,因为spring只支持方法类型的连接点,实际上joinpoint还可以是field或类构造器)（UserService里的save方法）

**Pointcut(切入点):**所谓切入点是指我们要对那些joinpoint进行拦截的定义.

**Advice(通知)**:所谓通知是指拦截到joinpoint之后所要做的事情就是通知.通知分为前置通知,后置通知,异常通知,最终通知,环绕通知

**Target(目标对象):**代理的目标对象 （JDKProxyFactory创建出来的代理对象）

**Weave(织入):**指将aspects应用到target对象并导致proxy对象创建的过程称为织入.

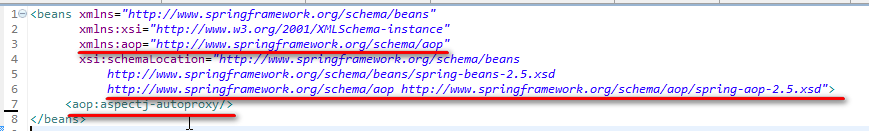
**Introduction(引入):**在不修改类代码的前提下, Introduction可以在运行期为类动态地添加一些方法或Field.

### Spring的AOP实现方式

当发现有AOP的配置的时候，Spring首先会检查配置的类是否有实现某些接口，如果已经实现了，那么会使用JDK的方式来创建代理对象，如果没有实现接口，那么就使用CGLIB的方式来实现

### Spring AOP的使用

首先启动对@AspectJ注解的支持(蓝色部分)和AOP的命名空间：



注：@AspectJ注解依赖于aspectj项目

A: 基于注解方式声明切面

|  |
| --- |
| @Aspect  public class LogPrint {//这个类必须交给spring管理，AOP才会起作用  @Pointcut("execution(\* cn.itcast.service..\*.\*(..))")  private void anyMethod() {}//声明一个切入点  @Before("anyMethod() && args(userName)")//定义前置通知  public void doAccessCheck(String userName) {  }  @AfterReturning(pointcut="anyMethod()",returning="revalue")//定义后置通知  public void doReturnCheck(String revalue) {  }  @AfterThrowing(pointcut="anyMethod()", throwing="ex")//定义例外通知  public void doExceptionAction(Exception ex) {  }  @After("anyMethod()")//定义最终通知  public void doReleaseAction() {  }  @Around("anyMethod()")//环绕通知  public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {  return pjp.proceed();  }  } |

B: 基于基于XML配置方式声明切面

|  |
| --- |
| <bean id="orderservice" class="cn.itcast.service.OrderServiceBean"/>  <bean id="log" class="cn.itcast.service.LogPrint"/>  <aop:config>  <aop:aspect id="myaop" ref="log">  <aop:pointcut id="mycut" expression="execution(\* cn.itcast.service..\*.\*(..))"/>  <aop:before pointcut-ref="mycut" method="doAccessCheck"/>  <aop:after-returning pointcut-ref="mycut" method="doReturnCheck "/>  <aop:after-throwing pointcut-ref="mycut" method="doExceptionAction"/>  <aop:after pointcut-ref="mycut" method=“doReleaseAction"/>  <aop:around pointcut-ref="mycut" method="doBasicProfiling"/>  </aop:aspect>  </aop:config> |

## FAQ

1. Spring是怎么处理依赖注入时的依赖情况的？读XML文件时的初始化顺序冲突是怎样解决的（如一个bean要待注入的property又依赖于其它的bean的初始化时）？