**Spring要点**

**spring bean 的生命周期**

1.Spring 容器根据配置中的 bean 定义中实例化 bean。

2. Spring 使用依赖注入填充所有属性，如 bean 中所定义的配置。

3. 如果 bean 实现 BeanNameAware 接口，则工厂通过传递 bean 的 ID 来调用 setBeanName()。

4. 如果 bean 实现 BeanFactoryAware 接口，工厂通过传递自身的实例来调用 setBeanFactory()。

5. 如果存在与 bean 关联的任何 BeanPostProcessors，则调用 postProcessBeforeInitialization() 方法。

6. 如果为 bean 指定了 init 方法（ 的 init-method 属性），那么将调用它。

7. 最后，如果存在与 bean 关联的任何 BeanPostProcessors，则将调用 postProcessAfterInitialization() 方法。

8. 如果 bean 实现 DisposableBean 接口，当 spring 容器关闭时，会调用 destory()。

9. 如果为 bean 指定了 destroy 方法（ 的 destroy-method 属性），那么将调用它。

**IOC底层实现原理**

IoC，控制反转 ，是一种设计思想，对于spring框架来说，就是由spring来负责控制对象的生命周期和对象间的关系。 是说创建对象的控制权进行转移，以前创建对象的主动权和创建时机是由自己把控的，而现在这种权力转移到第三方。**它是通过反射机制+工厂模式实现的，在实例化一个类时，它通过反射调用类中set方法将事先保存在Map中的类属性注入到类中。通过解析xml文件，获取到id属性和class属性里面的内容，利用反射原理获取到配置里面类的实例对象，存入到Spring的bean容器中。**

DI—Dependency Injection，即“依赖注入”由容器动态的将某个依赖关系注入到组件之中。

**让我们看看Spring 到底是怎么依赖注入的吧，其实依赖注入的思想也很简单，它是通过反射机制实现的，在实例化一个类时，它通过反射调用类中set方法将事先保存在HashMap中的类属性注入到类中**

**接下来Spring 就开始加载我们的配置文件了，将我们配置的信息保存在一个HashMap中，HashMap的key就是Bean 的 Id ，HasMap 的value是这个Bean，只有这样我们才能通过context.getBean("animal")这个方法获得Animal这个类。我们都知道Spirng可以注入基本类型，而且可以注入像List，Map这样的类型，接下来就让我们以Map为例看看Spring是怎么保存的吧**

**AOP底层实现原理**

面向方面编程，利用一种称为“横切”的技术，剖解开封装的对象内部。将那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来。便于减少系统的重复代码，降低模块间的耦合度。实现AOP的技术，主要分为两大类：一是采用动态代理技术，Spring默认使用Jdk动态代理，如果目标类不是接口选择cglib动态代理，二是采用静态织入的方式。

3、AOP的优点：  
  其实就是代理模式的优点，真实角色处理的业务更加纯粹，不用关注公共的业务。公共的业务由代理来完成，实现了业务的分工。公共业务发生改变（拓展）时，变得更加集中和方便。

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 。  
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 。

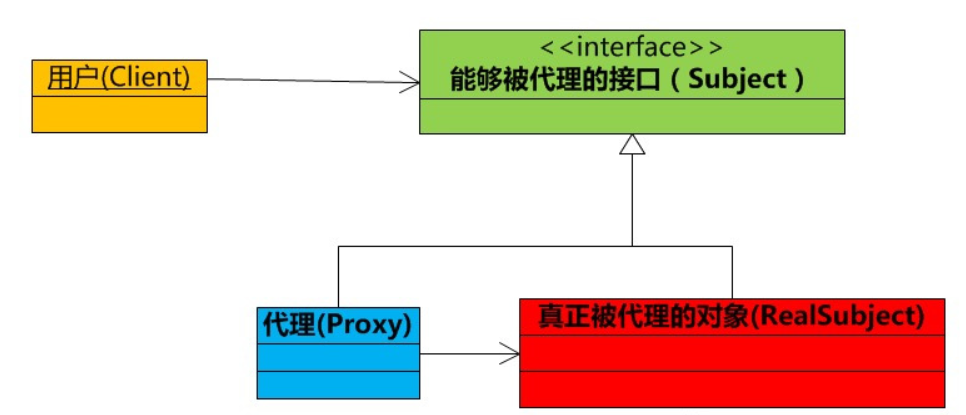
(1)当Bean实现接口时，Spring就会用JDK的动态代理

(2)当Bean没有实现接口时，Spring使用CGlib是实现

4、Spring AOP中的概念  
**关注点**：增加某个业务：如安全、缓存、异常等。  
**切面（Aspect）**：一个关注点的模块化。  
  **举个例子，日志一般都会在每个方法中都要添加，可以用一个方法将日志封装成一个类。这也就是模块化的理念。**

AOP的重要性：使得业务方法变得更加纯粹。Spring的底层还是通过动态代理实现的。但是**通过Spring来实现的话还要比动态代理要简单得多**。只需要配置一下就可以让方法加到指定的业务方法上去了，并且分的很清楚，例如是前置通知、后置通知。Spring AOP 可以认为是动态代理的框架。  
**在不去改变原有代码的基础上改变新的功能**。  
  **其本质——将公共的业务（日志、安全等）和领域业务结合。当执行领域业务时就会把公共业务加进来，实现公共业务的重复利用。使得领域业务更加纯粹。**

**JDK 动态代理主要涉及到 java.lang.reflect 包中的两个类：Proxy 和 InvocationHandler。 InvocationHandler是一个接口，通过实现该接口定义横切逻辑，并通过反射机制调用目标类 的代码，动态将横切逻辑和业务逻辑编制在一起。Proxy 利用 InvocationHandler 动态创建 一个符合某一接口的实例，生成目标类的代理对象**



————————————————

原文链接：https://blog.csdn.net/Butterfly\_resting/article/details/89735792

## .BeanFactory 接口和 ApplicationContext 接口有什么区别 ？

1. ApplicationContext 接口继承BeanFactory接口，Spring核心工厂是BeanFactory ,BeanFactory采取延迟加载，第一次getBean时才会初始化Bean, ApplicationContext是会在加载配置文件时初始化Bean。

②ApplicationContext是对BeanFactory扩展，它可以进行国际化处理、事件传递和bean自动装配以及各种不同应用层的Context实现

①**.Application**

一.singleton 解析xml-->解析Bean 一个对象

二.prototype 解析Bean-->解析Bean 有几个getBean就有几个对象

②**.BeanFactory**

无论是singleton还是prototy都是在解析Bean的时候-->解析Bean

开发中基本都在使用ApplicationContext, web项目使用WebApplicationContext ，很少用到BeanFactory

**通过解析xml文件，获取到id属性和class属性里面的内容，利用反射原理获取到配置里面类的实例对象，存入到Spring的bean容器中。**

# spring三种实例化bean的方式

一、使用构造器实例化；

二、使用静态工厂方法实例化；

三、使用实例化工厂方法实例化。

**一、使用构造器实例化；**

这种实例化的方式可能在我们平时的开发中用到的是最多的，因为在xml文件中配置简单并且也不需要额外的工厂类来实现。

1. <!--applicationContext.xml配置：-->
2. **<bean** id="personService" class="cn.mytest.service.impl.PersonServiceBean"**></bean>**
3. id是对象的名称，class是要实例化的类，然后再通过正常的方式进调用实例化的类即可，比如：
4. **public** **void** instanceSpring(){
5. //加载spring配置文件
6. ApplicationContext ac = **new** ClassPathXmlApplicationContext(
7. **new** String[]{  "/conf/applicationContext.xml"   });
8. //调用getBean方法取得被实例化的对象。
9. PersonServiceBean psb = (PersonServiceBean) ac.getBean("personService");
10. psb.save();
11. }  采用这种实例化方式要注意的是：要实例化的类中如果有构造器的话，一定要有一个无参的构造器。

**二、使用静态工厂方法实例化；**

根据这个中实例化方法的名称就可以知道要想通过这种方式进行实例化就要具备两个条件：（一）、要有工厂类及其工厂方法；（二）、工厂方法是静态的。OK，知道这两点就好办了，首先创建工程类及其静态方法：

1. **package** cn.mytest.service.impl;
3. /\*\*
4. \*创建工厂类
5. \*
6. \*/
7. **public** **class** PersonServiceFactory {
8. //创建静态方法
9. **public** **static** PersonServiceBean createPersonServiceBean(){
10. //返回实例化的类的对象
11. **return** **new** PersonServiceBean();
12. }
13. }
14. 然后再去配置spring配置文件，配置的方法和上面有点不同，这里也是关键所在
15. <!--applicationContext.xml配置：-->
16. **<bean** id="personService1" class="cn.mytest.service.impl.PersonServiceFactory" factory-method="createPersonServiceBean"**></bean>**
17. **public** **void** instanceSpring(){
18. //加载spring配置文件
19. ApplicationContext ac = **new** ClassPathXmlApplicationContext(
20. **new** String[]{   "/conf/applicationContext.xml"   });
21. //调用getBean方法取得被实例化的对象。
22. PersonServiceBean psb = (PersonServiceBean) ac.getBean("personService1");
24. psb.save();
25. }

**三、使用实例化工厂方法实例化。**

这个方法和上面的方法不同之处在与使用该实例化方式工厂方法不需要是静态的，但是在spring的配置文件中需要配置更多的内容，，首先创建工厂类及工厂方法：

1. **package** cn.mytest.service.impl;
3. /\*\*
4. \*创建工厂类
5. \*
6. \*/
7. **public** **class** PersonServiceFactory {
8. //创建静态方法
9. **public** PersonServiceBean createPersonServiceBean1(){
10. //返回实例化的类的对象
11. **return** **new** PersonServiceBean();
12. }
13. }
14. <!--applicationContext.xml配置：-->
15. **<bean** id="personServiceFactory" class="cn.mytest.service.impl.PersonServiceFactory"**></bean>**
16. **<bean** id="personService2" factory-bean="personServiceFactory" factory-method="createPersonServiceBean1"**></bean>**
17. 这里需要配置两个bean，第一个bean使用的构造器方法实例化工厂类，第二个bean中的id是实例化对象的名称，factory-bean对应的被实例化的工厂类的对象名称，也就是第一个bean的id，factory-method是非静态工厂方法。
18. **public** **void** instanceSpring(){
19. //加载spring配置文件
20. ApplicationContext ac = **new** ClassPathXmlApplicationContext(
21. **new** String[]{ "/conf/applicationContext.xml"   });
22. //调用getBean方法取得被实例化的对象。
23. PersonServiceBean psb = (PersonServiceBean) ac.getBean("personService2");
25. psb.save();
26. }

#### 9.介绍一下Spring的事物管理

事务就是对一系列的数据库操作（比如插入多条数据）进行统一的提交或回滚操作，如果插入成功，那么一起成功，如果中间有一条出现异常，那么回滚之前的所有操作。这样可以防止出现脏数据，防止数据库数据出现问题。

开发中为了避免这种情况一般都会进行事务管理。Spring中也有自己的事务管理机制，一般是使用TransactionMananger进行管 理，可以通过Spring的注入来完成此功能。spring提供了几个关于事务处理的类：

TransactionDefinition //事务属性定义

TranscationStatus //代表了当前的事务，可以提交，回滚。

**7、 解释Spring支持的几种bean的作用域。**

Spring容器中的bean可以分为5个范围

（1）singleton：默认，每个容器中只有一个bean的实例，单例的模式由BeanFactory自身来维护。

（2）prototype：为每一个bean请求提供一个实例。

（3）request：为每一个网络请求创建一个实例，在请求完成以后，bean会失效并被垃圾回收器回收。

（4）session：与request范围类似，确保每个session中有一个bean的实例，在session过期后，bean会随之失效。

（5）global-session：全局作用域，global-session和Portlet应用相关。当你的应用部署在Portlet容器中工作时，它包含很多portlet。如果你想要声明让所有的portlet共用全局的存储变量的话，那么这全局变量需要存储在global-session中。全局作用域与Servlet中的session作用域效果相同。

**9、Spring如何处理线程并发问题？**

在一般情况下，只有无状态的Bean才可以在多线程环境下共享，在Spring中，绝大部分Bean都可以声明为singleton作用域，因为Spring对一些Bean中非线程安全状态采用ThreadLocal进行处理，解决线程安全问题。

ThreadLocal和线程同步机制都是为了解决多线程中相同变量的访问冲突问题。同步机制采用了“时间换空间”的方式，仅提供一份变量，不同的线程在访问前需要获取锁，没获得锁的线程则需要排队。而ThreadLocal采用了“空间换时间”的方式。

ThreadLocal会为每一个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进ThreadLocal。

**11、Spring 框架中都用到了哪些设计模式？**<https://blog.csdn.net/chao821/article/details/92400186>

（1）工厂模式：BeanFactory就是简单工厂模式的体现，用来创建对象的实例；

（2）单例模式：Bean默认为单例模式。

（3）代理模式：Spring的AOP功能用到了JDK的动态代理和CGLIB字节码生成技术；

（4）模板方法：用来解决代码重复的问题。比如. RestTemplate, JmsTemplate, JpaTemplate。

（5）观察者模式：定义对象键一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都会得到通知被制动更新，如Spring中listener的实现--ApplicationListener。

**2、Spring事务的实现方式和实现原理：**

Spring事务的本质其实就是数据库对事务的支持，没有数据库的事务支持，spring是无法提供事务功能的。真正的数据库层的事务提交和回滚是通过binlog或者redo log实现的。

（1）Spring事务的种类：

**spring支持编程式事务管理和声明式事务管理两种方式：**

①编程式事务管理使用TransactionTemplate。

②声明式事务管理建立在AOP之上的。其本质是通过AOP功能，对方法前后进行拦截，将事务处理的功能编织到拦截的方法中，也就是在目标方法开始之前加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。

声明式事务最大的优点就是不需要在业务逻辑代码中掺杂事务管理的代**码，只需在配置文件中做相关的事务规则声明或通过@Transactional注解的方式，便可以将事务规则应用到业务逻辑中。**

声明式事务管理要优于编程式事务管理，这正是spring倡导的非侵入式的开发方式，使业务代码不受污染，只要加上注解就可以获得完全的事务支持。唯一不足地方是，最细粒度只能作用到方法级别，无法做到像编程式事务那样可以作用到代码块级别。

**（2）spring的事务传播行为：**

spring事务的传播行为说的是，当多个事务同时存在的时候，spring如何处理这些事务的行为。

① PROPAGATION\_REQUIRED：如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务，该设置是最常用的设置。

② PROPAGATION\_SUPPORTS：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就以非事务执行。‘

③ PROPAGATION\_MANDATORY：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就抛出异常。

④ PROPAGATION\_REQUIRES\_NEW：创建新事务，无论当前存不存在事务，都创建新事务。

⑤ PROPAGATION\_NOT\_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

⑥ PROPAGATION\_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

⑦ 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，则按REQUIRED属性执行。

**ServiceA {**

**void methodA() {**

**ServiceB.methodB();**

**}**

**}**

**ServiceB {**

**void methodB() {**

**}**

**}**

案例1，ServiceB.methodB的事务级别定义为PROPAGATION\_REQUIRED,

  1、如果ServiceA.methodA已经起了事务，这时调用ServiceB.methodB，会共用同一个事务，如果出现异常，ServiceA.methodA和ServiceB.methodB作为一个整体都将一起回滚。

  2、如果ServiceA.methodA没有事务，ServiceB.methodB就会为自己分配一个事务。ServiceA.methodA中是不受事务控制的。如果出现异常，ServiceB.methodB不会引起ServiceA.methodA的回滚。

案例2，ServiceA.methodA的事务级别PROPAGATION\_REQUIRED，ServiceB.methodB的事务级别PROPAGATION\_REQUIRES\_NEW，

调用ServiceB.methodB，ServiceA.methodA所在的事务就会挂起，ServiceB.methodB会起一个新的事务。

1、如果ServiceB.methodB已经提交，那么ServiceA.methodA失败回滚，ServiceB.methodB是不会回滚的。

2、如果ServiceB.methodB失败回滚，如果他抛出的异常被ServiceA.methodA的try..catch捕获并处理，ServiceA.methodA事务仍然可能提交；如果他抛出的异常未被ServiceA.methodA捕获处理，ServiceA.methodA事务将回滚。

案例3，ServiceA.methodA的事务级别为PROPAGATION\_REQUIRED，ServiceB.methodB的事务级别为PROPAGATION\_NESTED，

调用ServiceB.methodB的时候，ServiceA.methodA所在的事务就会挂起，ServiceB.methodB会起一个**新的子事务并设置savepoint**

1、如果ServiceB.methodB已经提交，那么ServiceA.methodA失败回滚，ServiceB.methodB也将回滚。

2、如果ServiceB.methodB失败回滚，如果他抛出的异常被ServiceA.methodA的try..catch捕获并处理，ServiceA.methodA事务仍然可能提交；如果他抛出的异常未被ServiceA.methodA捕获处理，ServiceA.methodA事务将回滚。

**（3）Spring中的隔离级别：**

① ISOLATION\_DEFAULT：这是个 PlatfromTransactionManager 默认的隔离级别，使用数据库默认的事务隔离级别。

② ISOLATION\_READ\_UNCOMMITTED：读未提交，允许另外一个事务可以看到这个事务未提交的数据。

③ ISOLATION\_READ\_COMMITTED：读已提交，保证一个事务修改的数据提交后才能被另一事务读取，而且能看到该事务对已有记录的更新。

④ ISOLATION\_REPEATABLE\_READ：可重复读，保证一个事务修改的数据提交后才能被另一事务读取，但是不能看到该事务对已有记录的更新。

⑤ ISOLATION\_SERIALIZABLE：一个事务在执行的过程中完全看不到其他事务对数据库所做的更新。

# [Spring中声明式事务的几个属性的解释](https://www.cnblogs.com/convict/p/9886690.html)

声明式事务 @Transactional （通常用在service层）  
事务属性：传播行为，隔离级别，回滚，只读，过期

1，spring支持事务传播行为：propagation（常用以下两个）  
    ① required 使用调用方法的事务，即在当前事务进行，买多本书，但是钱不够，这样的话整个事务回滚，一本书都不会买得到，钱也不会扣。  
    ② requires\_new 开启新的事务，而调用方法的事务会被挂起，即每次动作都会开启新的事务，即使出错也可以回滚到当前事务，不会影响到其他事务。

2，隔离级别：isolation  
事务并发容易产生问题，比如脏读，不可重复读，幻读等，指定事务隔离级别解决  
spring事务隔离级别：读已提交，读未提交，可重复读，序列化等

3，回滚：rollbackFor，noRollbackFor等  
默认时spring对所有异常进行回滚，可指定某种异常回滚或者不回滚

4，只读：readOnly  
指定事务只能读取数据而不更新数据，可以帮助数据库引擎优化事务

5，过期：timeout  
事务回滚前最多占用多长时间

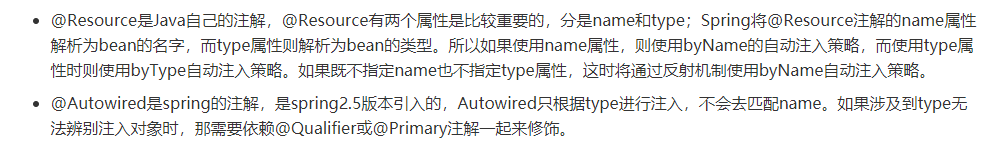
**Spring注解**

@Autowired可用于：构造函数、成员变量、Setter方法

**注：@Autowired和@Resource之间的区别**

(1) @Autowired默认是按照类型装配注入的，默认情况下它要求依赖对象必须存在（可以设置它required属性为false）。

(2) @Resource默认是按照名称来装配注入的，只有当找不到与名称匹配的bean才会按照类型来装配注入。



@Configuration作用在类上，声明一个class需要被spring解析以扩充beanDefinition。

@ComponentScan  
作用在类上，声明需要对包路径下资源进行扫描，效果等同于<context:component-scan>

# 注解service和component的区别

@service引用了@component注解，也就是component注解实现的功能@service都能实现，而@service是对@component进一步拓展，被@service注解标注的类会被spring认定是业务逻辑层

**(3).为什么要把这种方式叫做控制反转呢？**

— 软件系统在没有引入IoC容器之前，对象A依赖对象B，那么A对象在实例化或者运行到某一点的时候，自己必须主动创建对象B或者使用已经创建好的对象B，其中不管是创建还是使用已创建的对象B，控制权都在我们自己手上。

— 如果软件系统引入了Ioc容器之后，对象A和对象B之间失去了直接联系，所以，当对象A实例化和运行时，如果需要对象B的话，IoC容器会主动创建一个对象B注入到对象A所需要的地方。

— 通过前面的对比，可以看到对象A获得依赖对象B的过程，由主动行为变成了被动行为，即把创建对象交给了IoC容器处理，控制权颠倒过来了，这就是控制反转的由来！

先来看一下没有ioc容器的时候，对象A依赖对象B，A在运行到某一时刻的时候会去创建B的对象，在这里A具有主动权，它控制了对象B的创建。

引入ioc以后对象A和对象B之间没有了直接联系，当A运行的时候由ioc容器创建B对象在适当的时候注入到A中，在这里，控制权由A对象转移到了ioc容器。这也就是控制反转名称的由来。