# Spring面试题目

## Spring框架的优点

1.轻量级（不需要太多依赖）：Spring在大小和透明性方面绝对属于轻量级的，基础版本的Spring框架大约只有2MB。

2.控制反转(IOC)：Spring使用控制反转技术实现了松耦合。依赖被注入到对象，而不是创建或寻找依赖对象。

3.面向切面编程(AOP)： Spring支持面向切面编程，可以方便的实现对程序进行权限拦截、运行监控等功能。

4.事务管理：声明式事务的支持，只需要通过配置就可以完成对事务的管理，而无需手动编程。

5. 方便程序的测试：Spring对Junit4支持，可以通过注解方便的测试Spring程序。

6. 方便集成各种优秀框架：Spring不排斥各种优秀的开源框架，其内部提供了对各种优秀框架（如：Struts2、Hibernate、MyBatis、Quartz等）的直接支持。

7. 降低JavaEE API的使用难度： Spring 对JavaEE开发中非常难用的一些API（JDBC、JavaMail、远程调用等），都提供了封装，使这些API应用难度大大降低。

## Spring是一个分层的一站式框架

Spring针对javaee的三层结构，每一层都提供了解决技术。

**web层**：SpringMVC

**业务层（Service层）**：Bean管理（IOC)，Spring声明式事务

**持久层（Dao层）**：Spring的JDBC模板、ORM模板用于整合其他的持久层框架

## Spring的两个主要的特性

##### 1.AOP

AOP称为面向切面编程，在程序开发中常通过AOP来处理一些具有横切性质的系统级服务，如事务管理、安全检查、缓存、对象池管理等。

AOP是OOP的延伸，解决OOP开发遇到的问题。**AOP用横向抽取代替了纵向继承**，将封装好的对象剖开，找出其中对多个对象产生影响的公共行为，并将其封装为一个可重用的模块，这个模块被命名为“**切面**”（Aspect），切面将那些与业务无关，却被业务模块共同调用的逻辑提取并封装起来，减少了系统中的重复代码，降低了模块间的耦合度，同时提高了系统的可维护性。

AOP可以理解为一个拦截器框架，但是这个拦截器会非常武断，如果它拦截一个类，那么它就会拦截这个类中的所有方法。如对一个目标列的代理，增强了目标类的所有方法。

AOP的底层实现原理是动态代理。包括JDK动态代理和CGLIB动态代理。详细的讲解两种代理模式。

##### 2.IOC

IoC（Inversion of Control）是指容器控制程序对象之间的关系，而不是传统实现中，由程序代码直接操控。控制权由应用代码中转到了外部容器，控制权的转移是所谓反转。对于Spring而言，就是由Spring来控制对象的生命周期和对象之间的关系。**对于某个具体的对象而言，以前是它控制其他对象，现在是所有对象都被spring控制，所以这叫控制反转。**

此外，IOC还有一个特性--“依赖注入（Dependency Injection）”。从名字上理解，**所谓依赖注入，即组件之间的依赖关系由容器在运行期决定，即由容器动态地将某种依赖关系注入到组件之中**。依赖注入的思想是通过**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。

**总而言之，在传统的对象创建方式中，通常由调用者来创建被调用者的实例，而在Spring中创建被调用者的工作由Spring来完成，然后注入调用者，即所谓的控制反转和依赖注入。**

IoC的优点：Ioc通过**工厂模式+反射+配置文件**，降低了组件之间的耦合，降低了业务对象之间替换的复杂性，使之能够灵活的管理对象。

## 说说AOP实现原理

Spring底层实现aop的原理：动态代理。（增强类中的方法而不修改源码）

1.JDK动态代理：只能对实现了接口的类产生代理。

2.Cglib动态代理：对没有实现接口的类产生代理对象，生成子类对象。

##### 1.JDK动态代理



**步骤：**

* 通过实现InvocationHandlet接口创建自己的调用处理器。
* 首先调用代理工厂的createProxyIntance(Object obj)创建DoServiceImpl类的代理对象。
* 在该方法内，调用Proxy.newProxyInstance()方法创建代理对象。第一个参数是目标对象的类加载器，第二个参数是目标对象实现的接口，第三个参数传入一个InvocationHandler实例，该参数和回调有关系。
* 重写invoke(Object proxy, Method method, Object[] args)。该函数中有三个参数，分别是

proxy：最终生成的代理实例，很少用到。

method：被代理目标实例的某个具体方法，通过它可以发起目标实例方法的反射调用。

args：被代理实例某个方法的入参，在方法反射调用时使用。

* 每当调用目标对象的方法的时候，就会回调该InvocationHandler实例的方法，也就是public Object invoke()方法，我们就可以把限制的条件放在这里，条件符合的时候，就可以调用method.invoke()方法真正的调用目标对象的方法，否则，则可以在这里过滤掉不符合条件的调用。

java.lang.reflect包中的两个类：Proxy和InvocationHandler。

其中InvocationHandler只是一个接口，可以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，动态的将横切逻辑与业务逻辑织在一起。

而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。

##### 2.Cglib动态代理



* 在这一步中，我们使用一个Enhancer类来创建代理对象，不再使用Proxy。
* 使用Enhancer类，需要为其实例指定一个父类，也就是我们的目标对象。这样，我们新创建出来的对象就是目标对象的子类，有目标对象的一样。
* 除此之外，还要指定一个回调函数intercept，这个函数就和Proxy的 invoke()类似。intercept(Object object, Method method, Object[] args,MethodProxy proxy)，里面有四个参数，分别是：

object：目标类实例

method：目标类方法的反射对象

args：方法的动态入参

proxy：代理类实例

* 我们就可以把限制的条件放在这里，条件符合的时候，就可以调用method. intercept ()方法真正的调用目标对象的方法，否则，则可以在这里过滤掉不符合条件的调用。

## JDK动态代理和CGLIB动态代理的区别

JDK的动态代理主要涉及java.lang.reflect包中的两个类：Proxy和InvocationHandler。其中InvocationHandler只是一个接口，可以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，动态的将横切逻辑与业务逻辑织在一起。而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。 其代理对象必须是某个接口的实现，它是通过在运行期间创建一个接口的实现类来完成对目标对象的代理。只能针对实现接口的类生成代理，而不能针对类。

CGLib采用底层的字节码技术，为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类的调用方法，并顺势织入横切逻辑。它运行期间生成的代理对象是目标类的扩展子类，所以无法通知final的方法，因为它们不能被覆写，是针对类实现代理，主要是为指定的类生成一个子类覆盖其中方法。

在spring中默认情况下使用JDK动态代理实现AOP，如果proxy-target-class设置为true或者使用了优化策略那么会使用CGLIB来创建动态代理。SpringAOP在这两种方式的实现上基本一样。以JDK代理为例，会使用JdkDynamicAopProxy来创建代理，在invoke()方法首先需要织入到当前类的增强器封装到拦截器链中，然后递归的调用这些拦截器完成功能的织入．最终返回代理对象。

## cglib如何运用字节码技术实现AOP？

 cglib就是封装了asm，简化了asm的操作，实现了在运行期动态生成新的class。

它的原理就是用Enhancer生成一个原有类的子类，并且设置好callback，则原有类的每个方法调用都会转成调用实现了MethodInterceptor接口的proxy的intercept() 函数： 在intercept()函数里，你可以在执行Object result=proxy.invokeSuper(o,args);来执行原有函数，在执行前后加入自己的东西，改变它的参数，也可以瞒天过海，完全干别的。说白了，就是AOP中的around advice。

##### 如何使用Cglib

举个例子：比如DAO层有对表的增、删、改、查操作，如果要对原有的DAO层的增、删、改、查增加权限控制的话，修改代码是非常痛苦的。所以可以用AOP来实现。但是DAO层没有使用接口，动态代理不可用。这时候CGlib是个很好的选择。

**TableDao.java:**

package com.cglib;

public class TableDao {

public void create(){

System.out.println("create() is running...");

}

public void delete(){

System.out.println("delete() is running...");

}

public void update(){

System.out.println("update() is running...");

}

public void query(){

System.out.println("query() is running...");

}

}

**AuthProxy.java：**

**实现了MethodInterceptor接口的AuthProxy.java：用来对方法进行拦截，增加方法访问的权限控制，这里只允许张三访问。**

package com.cglib;

//方法拦截器

public class AuthProxy implements MethodInterceptor {

private String userName;

AuthProxy(String userName){

this.userName = userName;

}

//用来增强原有方法

public Object intercept(Object arg0, Method arg1, Object[] arg2,

MethodProxy arg3) throws Throwable {

//权限判断

if(!"张三".equals(userName)){

System.out.println("你没有权限！");

return null;

}

return arg3.invokeSuper(arg0, arg2);

}

}

**TableDAOFactory.java:**

用来创建TableDao的子类的工厂类

package com.cglib;

public class TableDAOFactory {

private static TableDao tDao = new TableDao();

public static TableDao getInstance(){

return tDao;

}

public static TableDao getAuthInstance(AuthProxy authProxy){

Enhancer en = new Enhancer(); //Enhancer用来生成一个原有类的子类

//进行代理

en.setSuperclass(TableDao.class);

//设置织入逻辑

en.setCallback(authProxy);

//生成代理实例

return (TableDao)en.create();

}

}

**测试类Client.java：**

package com.cglib;

public class Client {

public static void main(String[] args) {

// haveAuth();

haveNoAuth();

}

public static void doMethod(TableDao dao){

dao.create();

dao.query();

dao.update();

dao.delete();

}

//模拟有权限

public static void haveAuth(){

TableDao tDao = TableDAOFactory.getAuthInstance(new AuthProxy("张三"));

doMethod(tDao);

}

//模拟无权限

public static void haveNoAuth(){

TableDao tDao = TableDAOFactory.getAuthInstance(new AuthProxy("李四"));

doMethod(tDao);

}

}

## AOP里面的名称（术语）

Joinpoint(连接点）：类里面可以被增强的方法，这些方法称为连接点；

pointcut(切入点)：在类里面有很多方法可以被增强，而在实际操作中实际增强的方法称为切入点；

advice(通知/增强）：增强的逻辑或者扩展的功能，称为增强；

aspect(切面)：把增强应用到具体方法上面的过程称为切面（把增强用到切入点的过程）；

 target(目标对象）：增强方法所在的类称为目标对象；

weaving(织入）：把advice应用到target的过程；

Proxy(代理）：一个类被织入增强后，产生的结果代理类。

## AOP的通知方式

前置通知：在方法之前执行

后置通知：在方法之后执

异常通知：方法出现异常

最终通知：在后置之后来执行

环绕通知：在方法之前和之后来执行

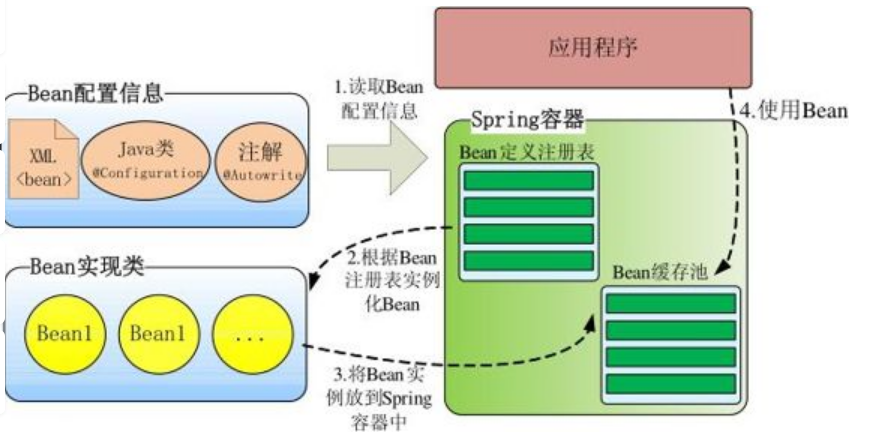
## spring的aop操作

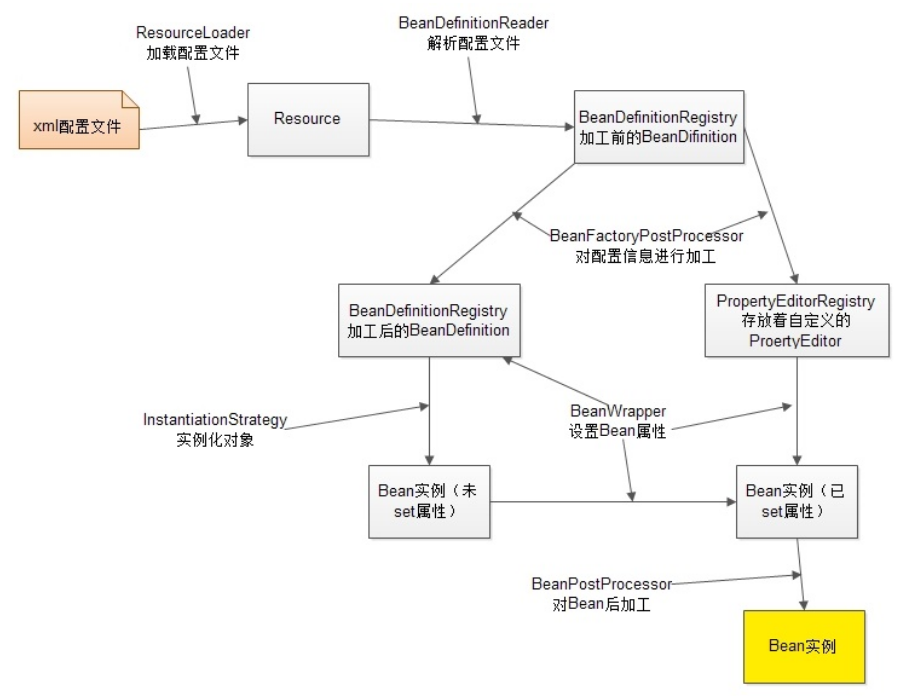
使用Aspectj来实现：AspectJ不是spring的一部分，只是一起使用，2.0以后增加了对他的支持。

使用aspectj实现aop有两种方式：1.基于aspectj的xml配置方式   2.基于aspectj的注解方式。

## Spring容器初始化过程

Spring 启动时读取应用程序提供的Bean配置信息，并在Spring容器中生成一份相应的Bean配置注册表，然后根据这张注册表实例化Bean，装配号Bean之间的依赖关系，为上层应用提供准备就绪的运行环境。





１、ResourceLoader从存储介质中加载Spring配置信息，并使用Resource表示这个配置文件的资源；

２、BeanDefinitionReader读取Resource所指向的配置文件资源，然后解析配置文件。配置文件中每一个<bean>解析成一个BeanDefinition对象，并保存到BeanDefinitionRegistry中；

３、容器扫描BeanDefinitionRegistry中的BeanDefinition，使用Java的反射机制自动识别出Bean工厂后处理后器（实现BeanFactoryPostProcessor接口）的Bean，然后调用这些Bean工厂后处理器对BeanDefinitionRegistry中的BeanDefinition进行加工处理。主要完成以下两项工作：

1）对使用到占位符的<bean>元素标签进行解析，得到最终的配置值，这意味对一些半成品式的BeanDefinition对象进行加工处理并得到成品的BeanDefinition对象；

2）对BeanDefinitionRegistry中的BeanDefinition进行扫描，通过Java反射机制找出所有属性编辑器的Bean（实现java.beans.PropertyEditor接口的Bean），并自动将它们注册到Spring容器的属性编辑器注册表中（PropertyEditorRegistry）；

4.Spring容器从BeanDefinitionRegistry中取出加工后的BeanDefinition，并调用InstantiationStrategy着手进行Bean实例化的工作；

5.在实例化Bean时，Spring容器使用BeanWrapper对Bean进行封装，BeanWrapper提供了很多以Java反射机制操作Bean的方法，它将结合该Bean的BeanDefinition以及容器中属性编辑器，完成Bean属性的设置工作；

6．利用容器中注册的Bean后处理器（实现BeanPostProcessor接口的Bean）对已经完成属性设置工作的Bean进行后续加工，直接装配出一个准备就绪的Bean。

## Spring的bean管理方式

1.XML方式管理

2.注解方式管理

## Spring的bean管理中常用的注解

##### 类的注解：

1.**@component** :组件。修饰一个类，将这个类交给Spring管理。

这个注解有三个衍生注解（功能类似）：

**@Controller**：web层

**@Service**：对应的是业务层Bean

**@Repository**：DAO层

##### 属性注入的注解：

普通属性：

**@value**  : 设置普通属性的类型

对象属性类型：

**@Autowired**:设置对象类型的属性的值。但是按照类型完成属性的注入。

我们习惯按照名称完成属性的注入：

必须让@Autowired和**@Qualifier**一起使用完成按照名称的属性注入。

**@Resource**:完成对象类型的属性的注入。按照名称完成属性的注入。

##### 生命周期注解：

**@PostConstruct**:初始化方法。

**@PreDestroy**:销毁方法。

##### Bean的作用范围的注解

**@scope**:作用域的注解。

## Spring的注解

1）

类的注解：

**@Component**：注解可以放在类的头上，将这个类交给Spring管理。

@Component不推荐使用。

Comeponent的衍生注解：

**@Service**：对应的是业务层Bean。

**@controller**: web层。由DispatcherServlet分发的请求，它把用户请求的数据经过业务处理层处理之后封装成一个Model，然后再把该Model返回给对应的View进行展示。

**@Resposity**:Dao层

2）

属性注入的注解：

**@Autowired** ：对象类型的注入，按照类型进行注入

**@Qualifier 和@Autowired** ：一起用按照属性进行注入

**@Value**   ：设置普通属性的值

**@Resource**：和@Autowired注解都是用来实现依赖注入的

只是@Autowired按byType自动注入，而@Resource默认按byName自动注入。

3）

生命周期的注解：

**@PostConstruct**：初始化方法

**@PreDestroy**：销毁方法

4）

作用域注解：

**@scope**

5）

Controller相关的注解：

类注解： @RestController、@RequestMapping

成员注解： @Autowired

方法注解： @GetMapping、@RequestMapping

@RequestMapping   ：可以声明到类或方法上。用来处理请求地址映射的注解

@RequestParam：完成参数绑定的一个注解

@SessionAttributes：作用在方法上或者方法的参数上，将被注解的方法的返回值或者是被注解的参数作为Model的属性加入到Model中，然后Spring框架自会将这个Model传递给ViewResolver。

@ModelAttribute：绑定请求参数到指定对象

@Required：用来检查bean在初始化时其声明的set方法是否被执行

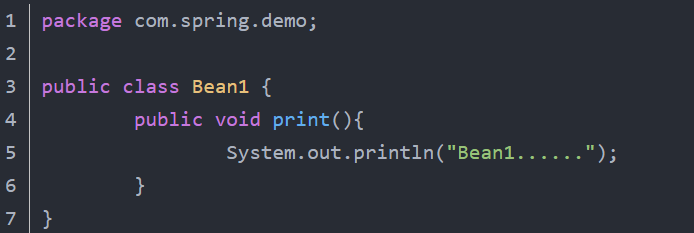
## Spring的Bean的实例化

Bean的实例化有三种方式：构造器实例化、静态工厂方式实例化、实例工厂方式实例化。

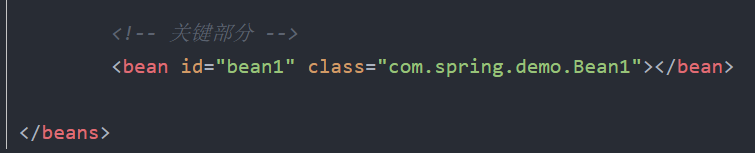
##### 一、构造器实例化

**1.首先定义一个 bean：**

bean1.java：

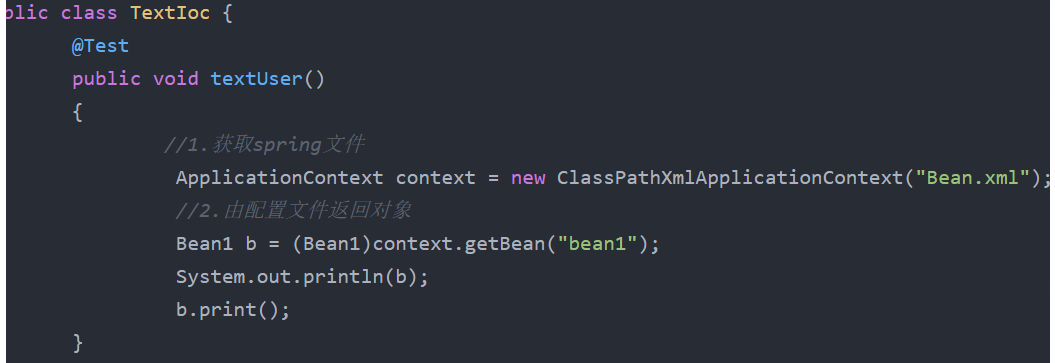


**2.在Spring配置文件Bean.xml 中 配置 bean：**



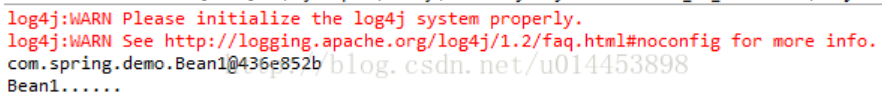
<bean id = “bean1” class="com.spring.demo.Bean1"></bean>：

**3.测试代码从配置文件中取出Bean1对象：**



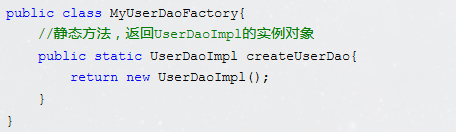
getBean() 返回的就是由spring 实例化的对象。

结果是：打印返回的对象b，和调用 b 的print()方法。

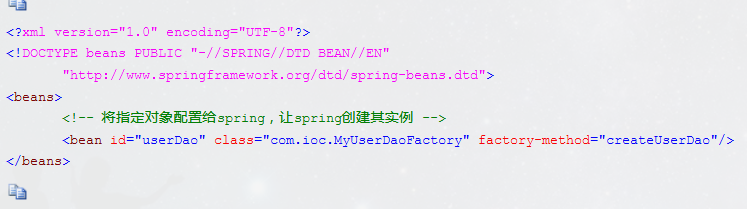


##### 二、静态工厂方式实例化

该方式首先要求创建一个静态工厂类，然后在类中定义一个静态方法来创建Bean实例，静态工厂类及静态方法的代码如下：



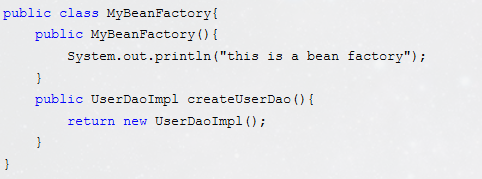
然后是xml配置文件的内容如下：



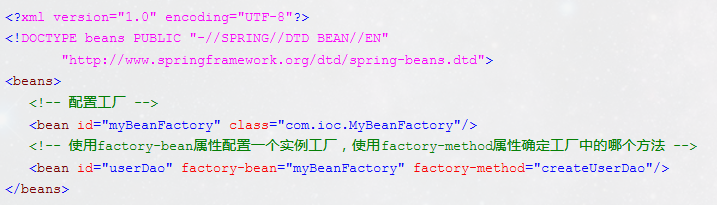
##### 三、实例工厂方式实例化

该种方式的工厂类中，不再使用静态方法创建Bean实例，而是采用直接创建Bean实例的方式。同时在配置文件中，需要实例化的Bean也不是通过class属性直接指向其实例化的类，而是通过factory-bean属性配置一个实例工厂，然后使用factory-method属性确定使用工厂中哪个方法。

工厂类方法代码如下：



xml配置文件如下：



## BeanFactory与ApplicationContext的区别

1.Bean工厂(BeanFactory)是Spring框架最核心的接口，提供了高级Ioc的配置机制；

2.应用上下文(ApplicationContext)建立在BeanFacotry基础之上，提供了更多面向应用的功能，如国际化，属性编辑器，事件等等；

3.beanFactory是spring框架的基础设施，是面向spring本身，ApplicationContext是面向使用Spring框架的开发者，几乎所有场合都会用到ApplicationContext。

## Bean的初始化过程（生命周期）

各种接口方法分类

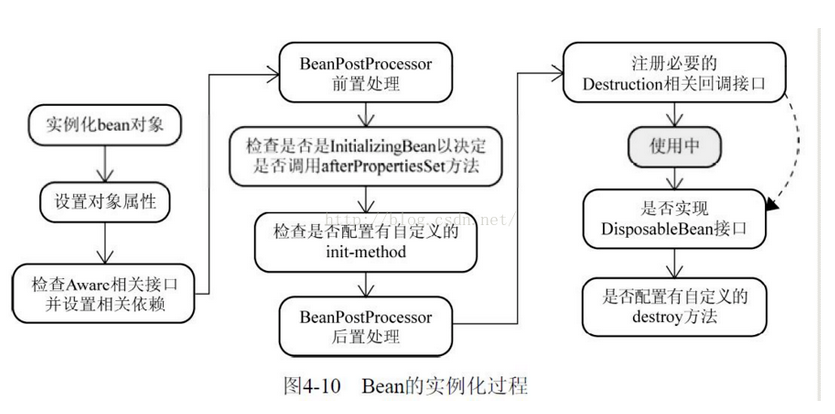
Bean的完整生命周期经历了各种方法调用，这些方法可以划分为以下几类：

1、Bean自身的方法 ： 这个包括了Bean本身调用的方法和通过配置文件中的init-method和destroy-method指定的方法。

2、Bean级生命周期接口方法 ： 这个包括了BeanNameAware、BeanFactoryAware、InitializingBean和DiposableBean这些接口的方法。

3、容器级生命周期接口方法 ： 这个包括了InstantiationAwareBeanPostProcessor 和 BeanPostProcessor 这两个接口实现，一般称它们的实现类为“后处理器”。

4、工厂后处理器接口方法 ： 这个包括了BeanFactoryPostProcessor等等非常有用的工厂后处理器接口的方法。工厂后处理器也是容器级的。在应用上下 文装配配置文件之后立即调用。



* Spring对bean进行实例化，默认bean是单例；
* Spring对bean进行依赖注入；
* 如果bean实现了BeanNameAware接口，spring将bean的id传给setBeanName()方法；
* 如果bean实现了BeanFactoryAware接口，spring将调用setBeanFactory方法，将BeanFactory实例传进来；
* 如果bean实现了ApplicationContextAware接口，它的setApplicationContext()方法将被调用，将应用上下文的引用传入到bean中；
* 如果bean实现了BeanPostProcessor接口，它的postProcessBeforeInitialization方法将被调用；
* 如果bean实现了InitializingBean接口，spring将调用它的afterPropertiesSet接口方法，类似的如果bean使用了init-method属性声明了初始化方法，该方法也会被调用；
* 如果bean实现了BeanPostProcessor接口，它的postProcessAfterInitialization接口方法将被调用；
* 此时bean已经准备就绪，可以被应用程序使用了，他们将一直驻留在应用上下文中，直到该应用上下文被销毁；
* 若bean实现了DisposableBean接口，spring将调用它的distroy()接口方法。同样的，如果bean使用了destroy-method属性声明了销毁方法，则该方法被调用；

## Spring的bean作用域

使用Scope属性来进行设置。代表Bean的作用范围。

五种作用域中，request、session和global session三种作用域仅在基于web的应用中使用，singleton和prototype只能用在基于web的Spring ApplicationContext环境。

当通过Spring容器创建一个Bean实例时，不仅可以完成Bean实例的实例化，还可以为Bean指定特定的作用域。Spring支持如下五种作用域：

1. singleton: 单例模式，在整个Spring IoC容器中，singleton作用域的Bean将只生成一个实例
2. prototype: 每次通过容器的getBean()方法获取prototype作用域的Bean时，都将产生一个新的Bean实例
3. request: 对于一次HTTP请求，request作用域的Bean将只生成一个实例，这意味着，在同一次HTTP请求内，程序每次请求该Bean，得到的总是同一个实例。只有在Web应用中使用Spring时，该作用域才真正有效
4. session：对于一次HTTP会话，session作用域的Bean将只生成一个实例，这意味着，在同一次HTTP会话内，程序每次请求该Bean，得到的总是同一个实例。只有在Web应用中使用Spring时，该作用域才真正有效
5. global session: 每个全局的HTTP Session对应一个Bean实例。在典型的情况下，仅在使用portlet context的时候有效，同样只在Web应用中有效

如果不指定Bean的作用域，Spring默认使用singleton作用域。prototype作用域的Bean的创建、销毁代价比较大。而singleton作用域的Bean实例一旦创建成果，就可以重复使用。因此，应该尽量避免将Bean设置成prototype作用域。

## Spring的Bean加载机制

比如我们这里定义了一个IOC容器，BeanFactroy的子类ClassXMLPathApplicationContext,在他的构造函数中我们会把xml路径写进去以此完成资源定位步骤，接下来就是BeanDefiniton的载入，在构造函数当中有一个refresh()的函数，这个就是载入BeanDefinition的接口，这个方法进去之后是一个同步代码块，把之前的容器销毁和关闭并且创建了一个BeanFatroy，就像对我们的容器重新启动一样，然后我们对BeanDefiniton载入和解析解析完毕之后会把beanDefinition和beanName放入BeanFactory的HashMap中维护。在这里Bean已经被创建完成，然后我们就像IOC容器索要Bean，如果是第一次索要会触发依赖注入，会递归的调用gebBean实现依赖出入。

## Spring 框架中都用到的设计模式

1. 单例模式：spring中bean默认是单例模式。Spring默认将所有的Bean设置成单例模式，即对所有的相同id的Bean的请求，都将返回同一个共享的Bean实例。这样就可以大大降低Java创建对象和销毁时的系统开销。使用Spring将Bean设置称为单例行为，则无需自己完成单例模式。

2.工厂模式：在各种BeanFactory以及ApplicationContext创建中都用到了。spring中的BeanFactory就是简单工厂模式的体现，根据传入一个唯一的标识来获得bean对象。

3.代理模式：在Aop实现中用到了动态代理。比如JdkDynamicAopProxy和Cglib2AopProxy。

4.模版方法模式：用来解决代码重复的问题. 比如. RestTemplate, JmsTemplate, JpaTemplate。

5.观察者模式：定义对象间的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得到通知并被自动更新。spring中Observer模式常用的地方是listener的实现。如ApplicationListener。

6.策略者模式：Spring 中策略模式使用有多个地方，如 Bean 定义对象的创建以及代理对象的创建等。这里主要看一下代理对象创建的策略模式的实现。前面已经了解 Spring 的代理方式有两个 Jdk 动态代理和CGLIB代理。这两个代理方式的使用正是使用了策略模式。

## Spring的依赖注入的方式

构造方法注入  
构造器依赖注入在容器触发构造器的时候完成，该构造器有一系列的参数，每个参数代表注入的对象。

##### set方法的属性注入

首先容器会触发一个无参构造函数或无参静态工厂方法实例化对象，之后容器调用bean中的setter方法完成Setter方法依赖注入。

##### 接口方法的属性注入

接口注入的意思是通过接口来实现信息的注入，而其它的类要实现该接口时，就可以实现了注入。

集合注入重要是对数组、List、Set、map的注入。



## 构造函数注入能不能和设值注入一块用？

不能，有优先级问题。

使用构造函数依赖注入时，Spring保证一个对象所有依赖的对象先实例化后，才实例化这个对象。使用set方法依赖注入时，Spring首先实例化对象，然后才实例化所有依赖的对象。

## 什么是循环依赖，Spring是怎样解决循环依赖的？

循环依赖：循环依赖就是N个类中循环嵌套引用。

**第一种：构造器参数循环依赖**

表示通过构造器注入构成的循环依赖，此依赖是无法解决的，只能抛出BeanCurrentlyIn CreationException异常表示循环依赖。

如在创建TestA类时，构造器需要TestB类，那将去创建TestB，在创建TestB类时又发现需要TestC类，则又去创建TestC，最终在创建TestC时发现又需要TestA，从而形成一个环，没办法创建。

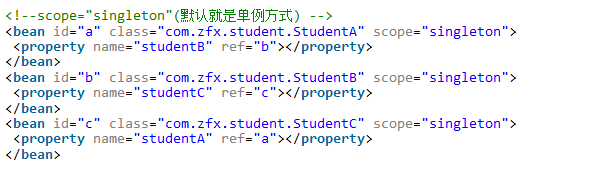
Spring容器会将每一个正在创建的Bean 标识符放在一个“当前创建Bean池”中，Bean标识符在创建过程中将一直保持。在这个池中，因此如果在创建Bean过程中发现自己已经在“当前创建Bean池”里时将抛出BeanCurrentlyInCreationException异常表示循环依赖；而对于创建完毕的Bean将从“当前创建Bean池”中清除掉。

例如：StudentA有参构造是StudentB。StudentB的有参构造是StudentC，StudentC的有参构造是StudentA ，这样就产生了一个循环依赖的情况，我们都把这三个Bean交给Spring管理，并用有参构造实例化。结果就会报错。

原因：Spring容器先创建单例StudentA，StudentA依赖StudentB，然后将A放在“当前创建Bean池”中，此时创建StudentB,StudentB依赖StudentC ,然后将B放在“当前创建Bean池”中,此时创建StudentC，StudentC又依赖StudentA， 但是，此时Student已经在池中，所以会报错，，因为在池中的Bean都是未初始化完的，所以会依赖错误 ，（初始化完的Bean会从池中移除）。

**第二种：setter方式单例，默认方式**

**结合**Spring中Bean实例化可知，Spring是先将Bean对象实例化之后再设置对象属性的。



（1） Spring容器创建单例“StudentA”Bean。首先依据无參构造器创建“StudentA”Bean， 并暴露一个exposedObject用于返回提前暴露的Bean。并将“StudentA”Bean放到Catch中。然后进行setter注入“StudentB”;  
 （2） Spring容器创建单例“StudentB " Bean。首先依据无參构造器创建“StudentB " Bean，并暴露一个exposedObject用于返回提前暴露的Bean。并将“StudentB”Bean放到Catch中，然后进行setter注入“StudentC”;  
 （3） Spring容器创建单例“StudentC”Bean，首先依据无參构造器创建“StudentC” Bean，并暴露一个exposedObject用于返回暴露的Bean。并将“StudentC”Bean放入Catch中， 然后进行setter注入“circularityA”。进行注入“circularityA”时因为步骤1提前暴露了exposedObject所以从之前的catch里面拿Bean不用反复创建。  
 （4） 最后在依赖注入“circularityB”和“circularityA”也是从catch里面拿提前暴露的bean。完毕setter注入。

**第三种：setter方式原型，prototype**

**同样使用上面的例子，改成prototype模式。程序就会报错。**

对于"prototype"作用域bean，Spring容器无法完成依赖注入，因为Spring容器不进行缓存"prototype"作用域的bean，因此无法提前暴露一个创建中的bean。

scope="prototype" 意思是每次请求都会创建一个实例对象。两者的区别是：有状态的bean都使用Prototype作用域，无状态的一般都使用singleton单例作用域。

## springmvc初始化过程

1.向服务器发送Http request请求，请求被**前端控制器（DispatcherServlet）**捕获。

2.前端控制器根据xml文件中的配置（或者注解）对请求的URL进行解析，得到请求资源标识符（URI）。然后根据该URI，调用**处理器映射器（HandlerMapping）**获得处理该请求的Handler以及Handler对应的拦截器，最后以 HandlerExecutionChain对象的形式返回到前端控制器。

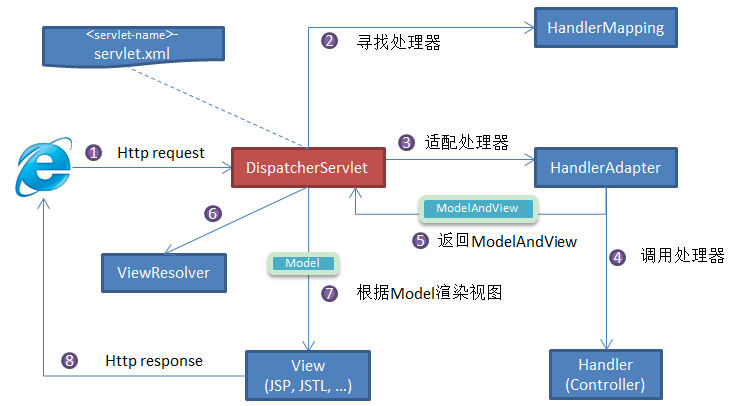
3.前端控制器根据获得的Handler，选择一个合适的**处理器适配器（HandlerAdapter）**去执行该Handler。

4.处理器适配器提取request中的模型数据，填充Handler入参，执行**处理器（Handler）**（也称之为Controller）。

5.Handler(Controller)执行完成后，向处理器适配器返回一个**ModelAndView**对象，处理器适配器再向前端控制器返回该ModelAndView对象（ModelAndView只是一个逻辑视图）。

6.根据返回的ModelAndView，前端控制器请求一个适合的**视图解析器（ViewResolver）**（必须是已经注册到Spring容器中的ViewResolver）去进行视图解析，然后视图解析器向前端控制器返回一个真正的视图View（jsp）。

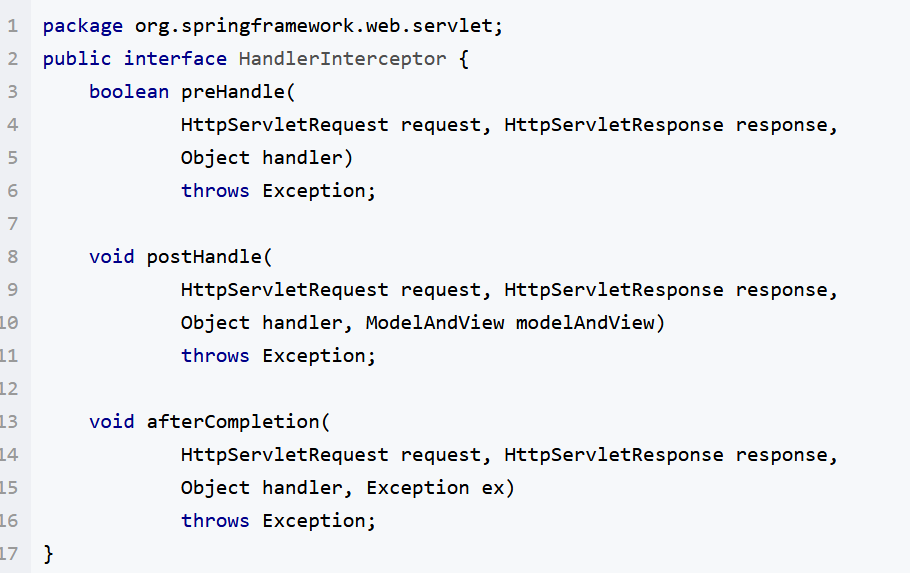
7.前端控制器通过Model解析出ModelAndView中的参数进行解析，最终展现出完整的View并通过Http response返回给客户端。



## Springmvc中拦截器的使用

在springmvc中，定义拦截器要实现HandlerInterceptor接口，并实现该接口中提供的三个方法。

1、拦截器接口HandlerInterceptor



* **preHandle**

预处理回调方法，实现处理器的预处理（如登录检查），第三个参数为响应的处理器（如具体的Controller实现）；   
返回值：true表示继续流程（如调用下一个拦截器或处理器）；false表示流程中断（如登录检查失败），不会继续调用其他的拦截器或处理器，此时我们需要通过response来产生响应；

* **postHandle**

后处理回调方法，实现处理器的后处理（但在渲染视图之前），此时我们可以通过modelAndView（模型和视图对象）对模型数据进行处理或对视图进行处理，modelAndView也可能为null。

* **afterCompletion**

整个请求处理完毕回调方法，即在视图渲染完毕时回调，如性能监控中我们可以在此记录结束时间并输出消耗时间，还可以进行一些资源清理，类似于try-catch-finally中的finally，但仅调用处理器执行链中preHandle返回true的拦截器的afterCompletion。

2、拦截器适配器

有时候我们可能只需要实现三个回调方法中的某一个，如果实现HandlerInterceptor接口的话，三个方法必须实现，不管你需不需要。此时spring提供了一个HandlerInterceptorAdapter适配器（一种适配器设计模式的实现），允许我们只实现需要的回调方法。



## Spring整合web项目

1.演示问题

（1）action调用service，service调用dao。每次访问action时候，都会加载spring配置文件。

2.解决方案：

（1）在服务器启动时候，创建对象加载配置文件

（2）底层使用监听器、ServletContext对象

3.具体使用：

在服务器启动时，为每个项目创建一个ServletContext对象；

在ServletContext对象创建时候，可以使用监听器监听到；

当监听到的时候，加载Spring配置文件，应用配置文件创建对象；

把创建好的对象放到ServletContext域对象里面（setAttribute方法）；

获取对象时候，到ServletContext域得到 （getAttribute方法）。

## spring监听器

监听器的作用：监听器是一个实现特定接口的普通java程序，这个程序专门用于监听另一个java对象的方法调用或者属性改变，当被监听对象发生上述事件后，监听器某个方法将立即执行。

系统启动的时候，系统自动加载org.springframework.web.context.ContextLoaderListener这个类，然后由它负责创建Spring容器即WebApplicationContext，系统会把bean放在其中，一般情况下，创建完成后，负责IOC操作，你不需要获取它，如果非要在程序中调用它，可以通过WebApplicationContext ctx = WebApplicationContextUtils.getWebApplicationContext(servletContext);来获取这个容器。