## ****Spring 概述****

### ****1. 什么是spring?****

Spring 是个java企业级应用的开源开发框架。Spring主要用来开发Java应用，但是有些扩展是针对构建J2EE平台的web应用。Spring 框架目标是简化Java企业级应用开发，并通过POJO为基础的编程模型促进良好的编程习惯。

### ****2. 使用Spring框架的好处是什么？****

* **轻量：**Spring 是轻量的，基本的版本大约2MB。
* **控制反转：**Spring通过控制反转实现了松散耦合，对象们给出它们的依赖，而不是创建或查找依赖的对象们。
* **面向切面的编程(AOP)：**Spring支持面向切面的编程，并且把应用业务逻辑和系统服务分开。
* **容器：**Spring 包含并管理应用中对象的生命周期和配置。
* **MVC框架**：Spring的WEB框架是个精心设计的框架，是Web框架的一个很好的替代品。
* **事务管理：**Spring 提供一个持续的事务管理接口，可以扩展到上至本地事务下至全局事务（JTA）。
* **异常处理：**Spring 提供方便的API把具体技术相关的异常（比如由JDBC，Hibernate or JDO抛出的）转化为一致的unchecked 异常。

### ****3.  Spring由哪些模块组成?****

以下是Spring 框架的基本模块：

* Core module
* Bean module
* Context module
* Expression Language module
* JDBC module
* ORM module
* OXM module
* Java Messaging Service(JMS) module
* Transaction module
* Web module
* Web-Servlet module
* Web-Struts module
* Web-Portlet module

### ****4. 核心容器（应用上下文) 模块。****

这是基本的Spring模块，提供spring 框架的基础功能，BeanFactory 是 任何以spring为基础的应用的核心。Spring 框架建立在此模块之上，它使Spring成为一个容器。

### ****5. BeanFactory – BeanFactory 实现举例。****

Bean 工厂是工厂模式的一个实现，提供了控制反转功能，用来把应用的配置和依赖从正真的应用代码中分离。

最常用的BeanFactory 实现是XmlBeanFactory 类。

### ****6. XMLBeanFactory****

最常用的就是org.springframework.beans.factory.xml.XmlBeanFactory ，它根据XML文件中的定义加载beans。该容器从XML 文件读取配置元数据并用它去创建一个完全配置的系统或应用。

### ****7. 解释AOP模块****

AOP模块用于发给我们的Spring应用做面向切面的开发， 很多支持由AOP联盟提供，这样就确保了Spring和其他AOP框架的共通性。这个模块将元数据编程引入Spring。

### ****8. 解释JDBC抽象和DAO模块。****

通过使用JDBC抽象和DAO模块，保证数据库代码的简洁，并能避免数据库资源错误关闭导致的问题，它在各种不同的数据库的错误信息之上，提供了一个统一的异常访问层。它还利用Spring的AOP 模块给Spring应用中的对象提供事务管理服务。

### ****9. 解释对象/关系映射集成模块。****

Spring 通过提供ORM模块，支持我们在直接JDBC之上使用一个对象/关系映射映射(ORM)工具，Spring 支持集成主流的ORM框架，如Hiberate,JDO和 iBATIS SQL Maps。Spring的事务管理同样支持以上所有ORM框架及JDBC。

### ****10.  解释WEB 模块。****

Spring的WEB模块是构建在application context 模块基础之上，提供一个适合web应用的上下文。这个模块也包括支持多种面向web的任务，如透明地处理多个文件上传请求和程序级请求参数的绑定到你的业务对象。它也有对Jakarta Struts的支持。

### ****12.  Spring配置文件****

Spring配置文件是个XML 文件，这个文件包含了类信息，描述了如何配置它们，以及如何相互调用。

### ****13.  什么是Spring IOC 容器？****

Spring IOC 负责创建对象，管理对象（通过依赖注入（DI），装配对象，配置对象，并且管理这些对象的整个生命周期。

### ****14.  IOC的优点是什么？****

IOC 或 依赖注入把应用的代码量降到最低。它使应用容易测试，单元测试不再需要单例和JNDI查找机制。最小的代价和最小的侵入性使松散耦合得以实现。IOC容器支持加载服务时的饿汉式初始化和懒加载。

### ****15. ApplicationContext通常的实现是什么?****

* **FileSystemXmlApplicationContext ：**此容器从一个XML文件中加载beans的定义，XML Bean 配置文件的全路径名必须提供给它的构造函数。
* **ClassPathXmlApplicationContext：**此容器也从一个XML文件中加载beans的定义，这里，你需要正确设置classpath因为这个容器将在classpath里找bean配置。
* **WebXmlApplicationContext：**此容器加载一个XML文件，此文件定义了一个WEB应用的所有bean。

### ****16. Bean 工厂和 Application contexts  有什么区别？****

Application contexts提供一种方法处理文本消息，一个通常的做法是加载文件资源（比如镜像），它们可以向注册为监听器的bean发布事件。另外，在容器或容器内的对象上执行的那些不得不由bean工厂以程序化方式处理的操作，可以在Application contexts中以声明的方式处理。Application contexts实现了MessageSource接口，该接口的实现以可插拔的方式提供获取本地化消息的方法。

### ****17. 一个Spring的应用看起来象什么？****

* 一个定义了一些功能的接口。
* 这实现包括属性，它的Setter ， getter 方法和函数等。
* Spring AOP。
* Spring 的XML 配置文件。
* 使用以上功能的客户端程序。

## ****依赖注入****

### ****18. 什么是Spring的依赖注入？****

依赖注入，是IOC的一个方面，是个通常的概念，它有多种解释。这概念是说你不用创建对象，而只需要描述它如何被创建。你不在代码里直接组装你的组件和服务，但是要在配置文件里描述哪些组件需要哪些服务，之后一个容器（IOC容器）负责把他们组装起来。

### ****19.  有哪些不同类型的IOC（依赖注入）方式？****

* **构造器依赖注入：**构造器依赖注入通过容器触发一个类的构造器来实现的，该类有一系列参数，每个参数代表一个对其他类的依赖。
* **Setter方法注入：**Setter方法注入是容器通过调用无参构造器或无参static工厂 方法实例化bean之后，调用该bean的setter方法，即实现了基于setter的依赖注入。

### ****20. 哪种依赖注入方式你建议使用，构造器注入，还是 Setter方法注入？****

你两种依赖方式都可以使用，构造器注入和Setter方法注入。最好的解决方案是用构造器参数实现强制依赖，setter方法实现可选依赖。

## ****Spring Beans****

### ****21.什么是Spring beans?****

Spring beans 是那些形成Spring应用的主干的java对象。它们被Spring IOC容器初始化，装配，和管理。这些beans通过容器中配置的元数据创建。比如，以XML文件中<bean/> 的形式定义。

Spring 框架定义的beans都是单件beans。在bean tag中有个属性”singleton”，如果它被赋为TRUE，bean 就是单件，否则就是一个 prototype bean。默认是TRUE，所以所有在Spring框架中的beans 缺省都是单件。

### ****22. 一个 Spring Bean 定义 包含什么？****

一个Spring Bean 的定义包含容器必知的所有配置元数据，包括如何创建一个bean，它的生命周期详情及它的依赖。

### ****23. 如何给Spring 容器提供配置元数据?****

这里有三种重要的方法给Spring 容器提供配置元数据。

XML配置文件。

基于注解的配置。

基于java的配置。

### ****24. 你怎样定义类的作用域?****

当定义一个<bean> 在Spring里，我们还能给这个bean声明一个作用域。它可以通过bean 定义中的scope属性来定义。如，当Spring要在需要的时候每次生产一个新的bean实例，bean的scope属性被指定为prototype。另一方面，一个bean每次使用的时候必须返回同一个实例，这个bean的scope 属性 必须设为 singleton。

### ****25. 解释Spring支持的几种bean的作用域。****

Spring框架支持以下五种bean的作用域：

* **singleton :**bean在每个Spring ioc 容器中只有一个实例。
* **prototype**：一个bean的定义可以有多个实例。
* **request**：每次http请求都会创建一个bean，该作用域仅在基于web的Spring ApplicationContext情形下有效。
* **session**：在一个HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。
* **global-session**：在一个全局的HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。

缺省的Spring bean 的作用域是Singleton.

### ****26. Spring框架中的单例bean是线程安全的吗?****

不，Spring框架中的单例bean不是线程安全的。

### ****27. 解释Spring框架中bean的生命周期。****

* Spring容器 从XML 文件中读取bean的定义，并实例化bean。
* Spring根据bean的定义填充所有的属性。
* 如果bean实现了BeanNameAware 接口，Spring 传递bean 的ID 到 setBeanName方法。
* 如果Bean 实现了 BeanFactoryAware 接口， Spring传递beanfactory 给setBeanFactory 方法。
* 如果有任何与bean相关联的BeanPostProcessors，Spring会在postProcesserBeforeInitialization()方法内调用它们。
* 如果bean实现IntializingBean了，调用它的afterPropertySet方法，如果bean声明了初始化方法，调用此初始化方法。
* 如果有BeanPostProcessors 和bean 关联，这些bean的postProcessAfterInitialization() 方法将被调用。
* 如果bean实现了 DisposableBean，它将调用destroy()方法。

### ****28.  哪些是重要的bean生命周期方法？ 你能重载它们吗？****

有两个重要的bean 生命周期方法，第一个是setup ， 它是在容器加载bean的时候被调用。第二个方法是 teardown  它是在容器卸载类的时候被调用。

The bean 标签有两个重要的属性（init-method和destroy-method）。用它们你可以自己定制初始化和注销方法。它们也有相应的注解（@PostConstruct和@PreDestroy）。

### ****29. 什么是Spring的内部bean？****

当一个bean仅被用作另一个bean的属性时，它能被声明为一个内部bean，为了定义inner bean，在Spring 的 基于XML的 配置元数据中，可以在 <property/>或 <constructor-arg/> 元素内使用<bean/> 元素，内部bean通常是匿名的，它们的Scope一般是prototype。

### ****30. 在 Spring中如何注入一个java集合？****

Spring提供以下几种集合的配置元素：

* <list>类型用于注入一列值，允许有相同的值。
* <set> 类型用于注入一组值，不允许有相同的值。
* <map> 类型用于注入一组键值对，键和值都可以为任意类型。
* <props>类型用于注入一组键值对，键和值都只能为String类型。

### ****31. 什么是bean装配?****

装配，或bean 装配是指在Spring 容器中把bean组装到一起，前提是容器需要知道bean的依赖关系，如何通过依赖注入来把它们装配到一起。

### ****32. 什么是bean的自动装配？****

Spring 容器能够自动装配相互合作的bean，这意味着容器不需要<constructor-arg>和<property>配置，能通过Bean工厂自动处理bean之间的协作。

### ****33. 解释不同方式的自动装配 。****

有五种自动装配的方式，可以用来指导Spring容器用自动装配方式来进行依赖注入。

* **no**：默认的方式是不进行自动装配，通过显式设置ref 属性来进行装配。
* **byName：**通过参数名 自动装配，Spring容器在配置文件中发现bean的autowire属性被设置成byname，之后容器试图匹配、装配和该bean的属性具有相同名字的bean。
* **byType:：**通过参数类型自动装配，Spring容器在配置文件中发现bean的autowire属性被设置成byType，之后容器试图匹配、装配和该bean的属性具有相同类型的bean。如果有多个bean符合条件，则抛出错误。
* **constructor：这个方式类似于**byType， 但是要提供给构造器参数，如果没有确定的带参数的构造器参数类型，将会抛出异常。
* **autodetect：**首先尝试使用constructor来自动装配，如果无法工作，则使用byType方式。

### ****34.自动装配有哪些局限性 ?****

自动装配的局限性是：

* **重写**： 你仍需用 <constructor-arg>和 <property> 配置来定义依赖，意味着总要重写自动装配。
* **基本数据类型**：你不能自动装配简单的属性，如基本数据类型，String字符串，和类。
* **模糊特性：**自动装配不如显式装配精确，如果有可能，建议使用显式装配。

### ****35. 你可以在Spring中注入一个null 和一个空字符串吗？****

**可以。**

## ****Spring注解****

### ****36. 什么是基于Java的Spring注解配置? 给一些注解的例子.****

基于Java的配置，允许你在少量的Java注解的帮助下，进行你的大部分Spring配置而非通过XML文件。

以@Configuration 注解为例，它用来标记类可以当做一个bean的定义，被Spring IOC容器使用。另一个例子是@Bean注解，它表示此方法将要返回一个对象，作为一个bean注册进Spring应用上下文。

### ****37. 什么是基于注解的容器配置?****

相对于XML文件，注解型的配置依赖于通过字节码元数据装配组件，而非尖括号的声明。

开发者通过在相应的类，方法或属性上使用注解的方式，直接组件类中进行配置，而不是使用xml表述bean的装配关系。

### ****38. 怎样开启注解装配？****

注解装配在默认情况下是不开启的，为了使用注解装配，我们必须在Spring配置文件中配置 <context:annotation-config/>元素。

### ****39. @Required  注解****

这个注解表明bean的属性必须在配置的时候设置，通过一个bean定义的显式的属性值或通过自动装配，若@Required注解的bean属性未被设置，容器将抛出BeanInitializationException。

### ****40. @Autowired 注解****

@Autowired 注解提供了更细粒度的控制，包括在何处以及如何完成自动装配。它的用法和@Required一样，修饰setter方法、构造器、属性或者具有任意名称和/或多个参数的PN方法。

### ****41. @Qualifier 注解****

当有多个相同类型的bean却只有一个需要自动装配时，将@Qualifier 注解和@Autowire 注解结合使用以消除这种混淆，指定需要装配的确切的bean。

## ****Spring数据访问****

### ****42.在Spring框架中如何更有效地使用JDBC?****

使用SpringJDBC 框架，资源管理和错误处理的代价都会被减轻。所以开发者只需写statements 和 queries从数据存取数据，JDBC也可以在Spring框架提供的模板类的帮助下更有效地被使用，这个模板叫JdbcTemplate （例子见这里[here](http://examples.javacodegeeks.com/enterprise-java/spring/jdbc/spring-jdbctemplate-example/" \t "http://www.importnew.com/_blank)）

### ****43. JdbcTemplate****

JdbcTemplate 类提供了很多便利的方法解决诸如把数据库数据转变成基本数据类型或对象，执行写好的或可调用的数据库操作语句，提供自定义的数据错误处理。

### ****44. Spring对DAO的支持****

Spring对数据访问对象（DAO）的支持旨在简化它和数据访问技术如JDBC，Hibernate or JDO 结合使用。这使我们可以方便切换持久层。编码时也不用担心会捕获每种技术特有的异常。

### ****45. 使用Spring通过什么方式访问Hibernate?****

在Spring中有两种方式访问Hibernate：

* 控制反转  Hibernate Template和 Callback。
* 继承 HibernateDAOSupport提供一个AOP 拦截器。

### ****46. Spring支持的ORM****

Spring支持以下ORM：

* Hibernate
* iBatis
* JPA (Java Persistence API)
* TopLink
* JDO (Java Data Objects)
* OJB

### ****47.如何通过HibernateDaoSupport将Spring和Hibernate结合起来？****

用Spring的 SessionFactory 调用 LocalSessionFactory。集成过程分三步：

* 配置the Hibernate SessionFactory。
* 继承HibernateDaoSupport实现一个DAO。
* 在AOP支持的事务中装配。

### ****48. Spring支持的事务管理类型****

Spring支持两种类型的事务管理：

* **编程式事务管理**：这意味你通过编程的方式管理事务，给你带来极大的灵活性，但是难维护。
* **声明式事务管理：**这意味着你可以将业务代码和事务管理分离，你只需用注解和XML配置来管理事务。

### ****49. Spring框架的事务管理有哪些优点？****

* 它为不同的事务API  如 JTA，JDBC，Hibernate，JPA 和JDO，提供一个不变的编程模式。
* 它为编程式事务管理提供了一套简单的API而不是一些复杂的事务API如
* 它支持声明式事务管理。
* 它和Spring各种数据访问抽象层很好得集成。

### ****50. 你更倾向用那种事务管理类型？****

大多数Spring框架的用户选择声明式事务管理，因为它对应用代码的影响最小，因此更符合一个无侵入的轻量级容器的思想。声明式事务管理要优于编程式事务管理，虽然比编程式事务管理（这种方式允许你通过代码控制事务）少了一点灵活性。

## ****Spring面向切面编程（AOP）****

### ****51.  解释AOP****

面向切面的编程，或AOP， 是一种编程技术，允许程序模块化横向切割关注点，或横切典型的责任划分，如日志和事务管理。

### ****52. Aspect 切面****

AOP核心就是切面，它将多个类的通用行为封装成可重用的模块，该模块含有一组API提供横切功能。比如，一个日志模块可以被称作日志的AOP切面。根据需求的不同，一个应用程序可以有若干切面。在Spring AOP中，切面通过带有@Aspect注解的类实现。

### ****52. 在Spring AOP 中，关注点和横切关注的区别是什么？****

关注点是应用中一个模块的行为，一个关注点可能会被定义成一个我们想实现的一个功能。  
横切关注点是一个关注点，此关注点是整个应用都会使用的功能，并影响整个应用，比如日志，安全和数据传输，几乎应用的每个模块都需要的功能。因此这些都属于横切关注点。

### ****54. 连接点****

连接点代表一个应用程序的某个位置，在这个位置我们可以插入一个AOP切面，它实际上是个应用程序执行Spring AOP的位置。

### ****55. 通知****

通知是个在方法执行前或执行后要做的动作，实际上是程序执行时要通过SpringAOP框架触发的代码段。

Spring切面可以应用五种类型的通知：

* **before**：前置通知，在一个方法执行前被调用。
* **after:**在方法执行之后调用的通知，无论方法执行是否成功。
* **after-returning:**仅当方法成功完成后执行的通知。
* **after-throwing:**在方法抛出异常退出时执行的通知。
* **around:**在方法执行之前和之后调用的通知。

### ****56. 切点****

切入点是一个或一组连接点，通知将在这些位置执行。可以通过表达式或匹配的方式指明切入点。

### ****57. 什么是引入?****

引入允许我们在已存在的类中增加新的方法和属性。

### ****58. 什么是目标对象?****

被一个或者多个切面所通知的对象。它通常是一个代理对象。也指被通知（advised）对象。

### ****59. 什么是代理?****

代理是通知目标对象后创建的对象。从客户端的角度看，代理对象和目标对象是一样的。

### ****60. 有几种不同类型的自动代理？****

BeanNameAutoProxyCreator

DefaultAdvisorAutoProxyCreator

Metadata autoproxying

### ****61. 什么是织入。什么是织入应用的不同点？****

织入是将切面和到其他应用类型或对象连接或创建一个被通知对象的过程。

织入可以在编译时，加载时，或运行时完成。

### ****62. 解释基于XML Schema方式的切面实现。****

在这种情况下，切面由常规类以及基于XML的配置实现。

### ****63. 解释基于注解的切面实现****

在这种情况下(基于@AspectJ的实现)，涉及到的切面声明的风格与带有java5标注的普通java类一致。

## ****Spring 的MVC****

### ****64. 什么是Spring的MVC框架？****

Spring 配备构建Web 应用的全功能MVC框架。Spring可以很便捷地和其他MVC框架集成，如Struts，Spring 的MVC框架用控制反转把业务对象和控制逻辑清晰地隔离。它也允许以声明的方式把请求参数和业务对象绑定。

### ****65. DispatcherServlet****

Spring的MVC框架是围绕DispatcherServlet来设计的，它用来处理所有的HTTP请求和响应。

### ****66. WebApplicationContext****

WebApplicationContext 继承了ApplicationContext  并增加了一些WEB应用必备的特有功能，它不同于一般的ApplicationContext ，因为它能处理主题，并找到被关联的servlet。

### ****67. 什么是Spring MVC框架的控制器？****

控制器提供一个访问应用程序的行为，此行为通常通过服务接口实现。控制器解析用户输入并将其转换为一个由视图呈现给用户的模型。Spring用一个非常抽象的方式实现了一个控制层，允许用户创建多种用途的控制器。

### ****68. @Controller 注解****

该注解表明该类扮演控制器的角色，Spring不需要你继承任何其他控制器基类或引用Servlet API。

### ****69. @RequestMapping 注解****

该注解是用来映射一个URL到一个类或一个特定的方处理法上。

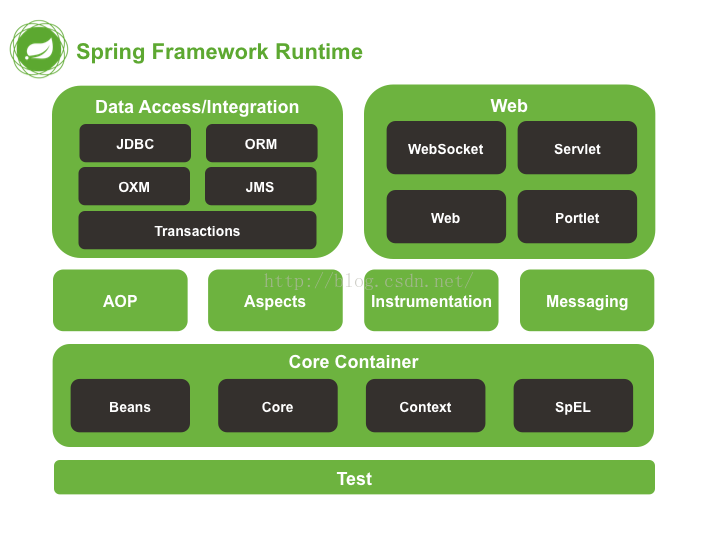
### 、什么是Spring框架？Spring框架有哪些主要模块？

Spring框架是一个为Java应用程序的开发提供了综合、广泛的基础性支持的Java平台。

Spring帮助开发者解决了开发中基础性的问题，使得开发人员可以专注于应用程序的开发。

Spring框架本身亦是按照[设计模式](http://www.amazon.cn/gp/product/B001130JN8/ref=as_li_qf_sp_asin_il_tl?ie=UTF8&tag=importnew-23&linkCode=as2&camp=536&creative=3200&creativeASIN=B001130JN8" \o "设计模式:可复用面向对象软件的基础" \t "https://blog.csdn.net/u014079773/article/details/_blank)精心打造，这使得我们可以在开发环境中安心的集成Spring框架，不必担心Spring是如何在后台进行工作的。

Spring框架至今已集成了20多个模块。这些模块主要被分如下图所示的核心容器、数据访问/集成,、Web、AOP（面向切面编程）、工具、消息和测试模块。



### 2、使用Spring框架能带来哪些好处？

下面列举了一些使用Spring框架带来的主要好处：

* [Dependency Injection(DI)](http://howtodoinjava.com/2013/03/19/inversion-of-control-ioc-and-dependency-injection-di-patterns-in-spring-framework-and-related-interview-questions/" \o "Inversion of control (IoC) and dependency injection (DI) patterns in spring framework and related interview questions" \t "https://blog.csdn.net/u014079773/article/details/_blank) 方法使得构造器和JavaBean properties文件中的依赖关系一目了然。
* 与EJB容器相比较，IoC容器更加趋向于轻量级。这样一来IoC容器在有限的内存和CPU资源的情况下进行应用程序的开发和发布就变得十分有利。
* Spring并没有闭门造车，Spring利用了已有的技术比如ORM框架、logging框架、J2EE、Quartz和JDK Timer，以及其他视图技术。
* Spring框架是按照模块的形式来组织的。由包和类的编号就可以看出其所属的模块，开发者仅仅需要选用他们需要的模块即可。
* 要[测试一项用Spring开发的应用程序](http://howtodoinjava.com/2013/04/19/how-to-unit-test-spring-security-authentication-with-junit/" \t "https://blog.csdn.net/u014079773/article/details/_blank)十分简单，因为测试相关的环境代码都已经囊括在框架中了。更加简单的是，利用JavaBean形式的POJO类，可以很方便的利用依赖注入来写入测试数据。
* Spring的Web框架亦是一个精心设计的Web MVC框架，为开发者们在web框架的选择上提供了一个除了主流框架比如Struts、过度设计的、不流行web框架的以外的有力选项。
* Spring提供了一个便捷的事务管理接口，适用于小型的本地事物处理（比如在单DB的环境下）和复杂的共同事物处理（比如利用JTA的复杂DB环境）。

### 3、什么是控制反转(IOC)？什么是依赖注入？

控制反转是应用于软件工程领域中的，在运行时被装配器对象来绑定耦合对象的一种编程技巧，对象之间耦合关系在编译时通常是未知的。在传统的编程方式中，业务逻辑的流程是由应用程序中的早已被设定好关联关系的对象来决定的。在使用控制反转的情况下，业务逻辑的流程是由对象关系图来决定的，该对象关系图由装配器负责实例化，这种实现方式还可以将对象之间的关联关系的定义抽象化。而绑定的过程是通过“依赖注入”实现的。

控制反转是一种以给予应用程序中目标组件更多控制为目的设计范式，并在我们的实际工作中起到了有效的作用。

依赖注入是在编译阶段尚未知所需的功能是来自哪个的类的情况下，将其他对象所依赖的功能对象实例化的模式。这就需要一种机制用来激活相应的组件以提供特定的功能，所以依赖注入是控制反转的基础。否则如果在组件不受框架控制的情况下，框架又怎么知道要创建哪个组件？

在Java中依然注入有以下三种实现方式：

1. 构造器注入
2. Setter方法注入
3. 接口注入

### 4、请解释下Spring框架中的IoC？

Spring中的 org.springframework.beans 包和 org.springframework.context包构成了Spring框架IoC容器的基础。

BeanFactory 接口提供了一个先进的配置机制，使得任何类型的对象的配置成为可能。ApplicationContex接口对BeanFactory（是一个子接口）进行了扩展，在BeanFactory的基础上添加了其他功能，比如与[Spring的AOP](http://howtodoinjava.com/category/frameworks/java-spring-tutorials/spring-aop/" \t "https://blog.csdn.net/u014079773/article/details/_blank)更容易集成，也提供了处理[message resource的机制](http://howtodoinjava.com/2015/02/10/spring-mvc-internationalization-i18n-and-localization-i10n-example/" \t "https://blog.csdn.net/u014079773/article/details/_blank)（用于国际化）、事件传播以及应用层的特别配置，比如针对Web应用的WebApplicationContext。

org.springframework.beans.factory.BeanFactory 是Spring IoC容器的具体实现，用来包装和管理前面提到的各种bean。BeanFactory接口是Spring IoC 容器的核心接口。

IOC:把对象的创建、初始化、销毁交给spring来管理，而不是由开发者控制，实现控制反转。

### 5、BeanFactory和ApplicationContext有什么区别？

BeanFactory 可以理解为含有bean集合的工厂类。BeanFactory 包含了种bean的定义，以便在接收到客户端请求时将对应的bean实例化。

BeanFactory还能在实例化对象的时生成协作类之间的关系。此举将bean自身与bean客户端的配置中解放出来。BeanFactory还包含了bean生命周期的控制，调用客户端的初始化方法（initialization methods）和销毁方法（destruction methods）。

从表面上看，application context如同bean factory一样具有bean定义、bean关联关系的设置，根据请求分发bean的功能。但applicationcontext在此基础上还提供了其他的功能。

1. 提供了支持国际化的文本消息
2. 统一的资源文件读取方式
3. 已在监听器中注册的bean的事件

以下是三种较常见的 ApplicationContext 实现方式：

1、ClassPathXmlApplicationContext：从classpath的XML配置文件中读取上下文，并生成上下文定义。应用程序上下文从程序环境变量中取得。

**[html]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. ApplicationContext context = new ClassPathXmlApplicationContext(“bean.xml”);

ApplicationContext context = new ClassPathXmlApplicationContext(“bean.xml”);

2、FileSystemXmlApplicationContext ：由文件系统中的XML配置文件读取上下文。

**[html]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. ApplicationContext context = new FileSystemXmlApplicationContext(“bean.xml”);

ApplicationContext context = new FileSystemXmlApplicationContext(“bean.xml”);

3、XmlWebApplicationContext：由Web应用的XML文件读取上下文。

### 6、Spring有几种配置方式？

将Spring配置到应用开发中有以下三种方式：

1. 基于XML的配置
2. 基于注解的配置
3. 基于Java的配置

### 7、如何用基于XML配置的方式配置Spring？

在Spring框架中，依赖和服务需要在专门的配置文件来实现，我常用的XML格式的配置文件。这些配置文件的格式通常用<beans>开头，然后一系列的bean定义和专门的应用配置选项组成。

SpringXML配置的主要目的时候是使所有的Spring组件都可以用xml文件的形式来进行配置。这意味着不会出现其他的Spring配置类型（比如声明的方式或基于Java Class的配置方式）

Spring的XML配置方式是使用被Spring命名空间的所支持的一系列的XML标签来实现的。Spring有以下主要的命名空间：context、beans、jdbc、tx、aop、mvc和aso。

如：

**[html]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. <beans>
2. <!-- JSON Support -->
3. <bean name="viewResolver" class="org.springframework.web.servlet.view.BeanNameViewResolver"/>
4. <bean name="jsonTemplate" class="org.springframework.web.servlet.view.json.MappingJackson2JsonView"/>
5. <bean id="restTemplate" class="org.springframework.web.client.RestTemplate"/>
6. </beans>

<beans>

<!-- JSON Support -->

<bean name="viewResolver" class="org.springframework.web.servlet.view.BeanNameViewResolver"/>

<bean name="jsonTemplate" class="org.springframework.web.servlet.view.json.MappingJackson2JsonView"/>

<bean id="restTemplate" class="org.springframework.web.client.RestTemplate"/>

</beans>

下面这个web.xml仅仅配置了DispatcherServlet，这件最简单的配置便能满足应用程序配置运行时组件的需求。

**[html]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. <web-app>
2. <display-name>Archetype Created Web Application</display-name>
3. <servlet>
4. <servlet-name>spring</servlet-name>
5. <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
6. <load-on-startup>1</load-on-startup>
7. </servlet>
8. <servlet-mapping>
9. <servlet-name>spring</servlet-name>
10. <url-pattern>/</url-pattern>
11. </servlet-mapping>
12. </web-app>

<web-app>

<display-name>Archetype Created Web Application</display-name>

<servlet>

<servlet-name>spring</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>spring</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

### 8、如何用基于Java配置的方式配置Spring？

Spring对Java配置的支持是由@Configuration注解和@Bean注解来实现的。由@Bean注解的方法将会实例化、配置和初始化一个新对象，这个对象将由Spring的IoC容器来管理。@Bean声明所起到的作用与<bean/> 元素类似。被@Configuration所注解的类则表示这个类的主要目的是作为bean定义的资源。被@Configuration声明的类可以通过在同一个类的内部调用@bean方法来设置嵌入bean的依赖关系。

最简单的@Configuration 声明类请参考下面的代码：

**[java]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. @Configuration
2. public class AppConfig{
3. @Bean
4. public MyService myService() {
5. return new MyServiceImpl();
6. }
7. }

@Configuration

public class AppConfig{

@Bean

public MyService myService() {

return new MyServiceImpl();

}

}

对于上面的@Beans配置文件相同的XML配置文件如下：

**[html]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. <beans>
2. <bean id="myService" class="com.somnus.services.MyServiceImpl"/>
3. </beans>

<beans>

<bean id="myService" class="com.somnus.services.MyServiceImpl"/>

</beans>

上述配置方式的实例化方式如下：利用AnnotationConfigApplicationContext 类进行实例化

**[java]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. public static void main(String[] args) {
2. ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);
3. MyService myService = ctx.getBean(MyService.class);
4. myService.doStuff();
5. }

public static void main(String[] args) {

ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);

MyService myService = ctx.getBean(MyService.class);

myService.doStuff();

}

要使用组件组建扫描，仅需用@Configuration进行注解即可：

**[java]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. @Configuration
2. @ComponentScan(basePackages = "com.somnus")
3. public class AppConfig  {
4. ...
5. }

@Configuration

@ComponentScan(basePackages = "com.somnus")

public class AppConfig {

...

}

在上面的例子中，com.acme包首先会被扫到，然后再容器内查找被@Component 声明的类，找到后将这些类按照Sring bean定义进行注册。

如果你要在你的web应用开发中选用上述的配置的方式的话，需要用AnnotationConfigWebApplicationContext 类来读取配置文件，可以用来配置Spring的Servlet监听器ContextLoaderListener或者Spring MVC的DispatcherServlet。

**[html]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. <web-app>
2. <!-- Configure ContextLoaderListener to use AnnotationConfigWebApplicationContext
3. instead of the default XmlWebApplicationContext -->
4. <context-param>
5. <param-name>contextClass</param-name>
6. <param-value>
7. org.springframework.web.context.support.AnnotationConfigWebApplicationContext
8. </param-value>
9. </context-param>
11. <!-- Configuration locations must consist of one or more comma- or space-delimited
12. fully-qualified @Configuration classes. Fully-qualified packages may also be
13. specified for component-scanning -->
14. <context-param>
15. <param-name>contextConfigLocation</param-name>
16. <param-value>com.howtodoinjava.AppConfig</param-value>
17. </context-param>
19. <!-- Bootstrap the root application context as usual using ContextLoaderListener -->
20. <listener>
21. <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
22. </listener>
24. <!-- Declare a Spring MVC DispatcherServlet as usual -->
25. <servlet>
26. <servlet-name>dispatcher</servlet-name>
27. <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
28. <!-- Configure DispatcherServlet to use AnnotationConfigWebApplicationContext
29. instead of the default XmlWebApplicationContext -->
30. <init-param>
31. <param-name>contextClass</param-name>
32. <param-value>
33. org.springframework.web.context.support.AnnotationConfigWebApplicationContext
34. </param-value>
35. </init-param>
36. <!-- Again, config locations must consist of one or more comma- or space-delimited
37. and fully-qualified @Configuration classes -->
38. <init-param>
39. <param-name>contextConfigLocation</param-name>
40. <param-value>com.howtodoinjava.web.MvcConfig</param-value>
41. </init-param>
42. </servlet>
44. <!-- map all requests for /app/\* to the dispatcher servlet -->
45. <servlet-mapping>
46. <servlet-name>dispatcher</servlet-name>
47. <url-pattern>/app/\*</url-pattern>
48. </servlet-mapping>
49. </web-app

<web-app>

<!-- Configure ContextLoaderListener to use AnnotationConfigWebApplicationContext

instead of the default XmlWebApplicationContext -->

<context-param>

<param-name>contextClass</param-name>

<param-value>

org.springframework.web.context.support.AnnotationConfigWebApplicationContext

</param-value>

</context-param>

<!-- Configuration locations must consist of one or more comma- or space-delimited

fully-qualified @Configuration classes. Fully-qualified packages may also be

specified for component-scanning -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>com.howtodoinjava.AppConfig</param-value>

</context-param>

<!-- Bootstrap the root application context as usual using ContextLoaderListener -->

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<!-- Declare a Spring MVC DispatcherServlet as usual -->

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<!-- Configure DispatcherServlet to use AnnotationConfigWebApplicationContext

instead of the default XmlWebApplicationContext -->

<init-param>

<param-name>contextClass</param-name>

<param-value>

org.springframework.web.context.support.AnnotationConfigWebApplicationContext

</param-value>

</init-param>

<!-- Again, config locations must consist of one or more comma- or space-delimited

and fully-qualified @Configuration classes -->

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>com.howtodoinjava.web.MvcConfig</param-value>

</init-param>

</servlet>

<!-- map all requests for /app/\* to the dispatcher servlet -->

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>/app/\*</url-pattern>

</servlet-mapping>

</web-app

### 9、怎样用注解的方式配置Spring？

Spring在2.5版本以后开始支持用注解的方式来配置依赖注入。可以用注解的方式来替代XML方式的bean描述，可以将bean描述转移到组件类的内部，只需要在相关类上、方法上或者字段声明上使用注解即可。注解注入将会被容器在XML注入之前被处理，所以后者会覆盖掉前者对于同一个属性的处理结果。

注解装配在Spring中是默认关闭的。所以需要在Spring文件中配置一下才能使用基于注解的装配模式。如果你想要在你的应用程序中使用关于注解的方法的话，请参考如下的配置。

**[html]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. <beans>
2. <context:annotation-config/>
3. <!-- bean definitions go here -->
4. </beans>

<beans>

<context:annotation-config/>

<!-- bean definitions go here -->

</beans>

在 <context:annotation-config/>标签配置完成以后，就可以用注解的方式在Spring中向属性、方法和构造方法中自动装配变量。

下面是几种比较重要的注解类型：

1. @Required：该注解应用于设值方法。
2. @Autowired：该注解应用于有值设值方法、非设值方法、构造方法和变量。
3. @Qualifier：该注解和@Autowired注解搭配使用，用于消除特定bean自动装配的歧义。
4. JSR-250 Annotations：Spring支持基于JSR-250 注解的以下注解，@Resource、@PostConstruct 和 @PreDestroy。

### 10、请解释Spring Bean的生命周期？

Spring Bean的生命周期简单易懂。在一个bean实例被初始化时，需要执行一系列的初始化操作以达到可用的状态。同样的，当一个bean不在被调用时需要进行相关的析构操作，并从bean容器中移除。

Spring bean factory 负责管理在spring容器中被创建的bean的生命周期。Bean的生命周期由两组回调（call back）方法组成。

1. 初始化之后调用的回调方法。
2. 销毁之前调用的回调方法。

Spring框架提供了以下四种方式来管理bean的生命周期事件：

* InitializingBean和DisposableBean回调接口
* 针对特殊行为的其他Aware接口
* Bean配置文件中的Custom init()方法和destroy()方法
* @PostConstruct和@PreDestroy注解方式

使用customInit()和 customDestroy()方法管理bean生命周期的代码样例如下：

**[html]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. <beans>
2. <bean id="demoBean" class="com.somnus.task.DemoBean" init-method="customInit" destroy-method="customDestroy"></bean>
3. </beans>

<beans>

<bean id="demoBean" class="com.somnus.task.DemoBean" init-method="customInit" destroy-method="customDestroy"></bean>

</beans>

### 11、Spring Bean的作用域之间有什么区别？

Spring容器中的bean可以分为5个范围。所有范围的名称都是自说明的，但是为了避免混淆，还是让我们来解释一下：

1. singleton：这种bean范围是默认的，这种范围确保不管接受到多少个请求，每个容器中只有一个bean的实例，单例的模式由bean factory自身来维护。
2. prototype：原形范围与单例范围相反，为每一个bean请求提供一个实例。
3. request：在请求bean范围内会每一个来自客户端的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：与请求范围类似，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：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。

### 12、什么是Spring inner beans？

在Spring框架中，无论何时bean被使用时，当仅被调用了一个属性。一个明智的做法是将这个bean声明为内部bean。内部bean可以用setter注入“属性”和构造方法注入“构造参数”的方式来实现。

比如，在我们的应用程序中，一个Customer类引用了一个Person类，我们的要做的是创建一个Person的实例，然后在Customer内部使用。

**[java]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. public class Customer{
2. private Person person;
3. //Setters and Getters
4. }

public class Customer{

private Person person;

//Setters and Getters

}

**[java]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. public class Person{
2. private String name;
3. private String address;
4. private int age;
5. //Setters and Getters
6. }

public class Person{

private String name;

private String address;

private int age;

//Setters and Getters

}

内部bean的声明方式如下：

**[java]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. <bean id="CustomerBean" class="com.somnus.common.Customer">
2. <property name="person">
3. <!-- This is inner bean -->
4. <bean class="com.howtodoinjava.common.Person">
5. <property name="name" value="lokesh" />
6. <property name="address" value="India" />
7. <property name="age" value="34" />
8. </bean>
9. </property>
10. </bean>

<bean id="CustomerBean" class="com.somnus.common.Customer">

<property name="person">

<!-- This is inner bean -->

<bean class="com.howtodoinjava.common.Person">

<property name="name" value="lokesh" />

<property name="address" value="India" />

<property name="age" value="34" />

</bean>

</property>

</bean>

### 13、Spring框架中的单例Beans是线程安全的么？

Spring框架并没有对[单例](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/" \t "https://blog.csdn.net/u014079773/article/details/_blank)bean进行任何多线程的封装处理。关于单例bean的[线程安全](http://howtodoinjava.com/2014/06/02/what-is-thread-safety/" \t "https://blog.csdn.net/u014079773/article/details/_blank)和并发问题需要开发者自行去搞定。但实际上，大部分的Spring bean并没有可变的状态(比如Serview类和DAO类)，所以在某种程度上说Spring的单例bean是线程安全的。如果你的bean有多种状态的话（比如 View Model 对象），就需要自行保证线程安全。  
最浅显的解决办法就是将多态bean的作用域由“singleton”变更为“prototype”。

### 14、请举例说明如何在Spring中注入一个Java Collection？

Spring提供了以下四种集合类的配置元素：

* <list> :   该标签用来装配可重复的list值。
* <set> :    该标签用来装配没有重复的set值。
* <map>:   该标签可用来注入键和值可以为任何类型的键值对。
* <props> : 该标签支持注入键和值都是字符串类型的键值对。

下面看一下具体的例子：

**[html]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. <beans>
2. <!-- Definition for javaCollection -->
3. <bean id="javaCollection" class="com.howtodoinjava.JavaCollection">
4. <!-- java.util.List -->
5. <property name="customList">
6. <list>
7. <value>INDIA</value>
8. <value>Pakistan</value>
9. <value>USA</value>
10. <value>UK</value>
11. </list>
12. </property>
14. <!-- java.util.Set -->
15. <property name="customSet">
16. <set>
17. <value>INDIA</value>
18. <value>Pakistan</value>
19. <value>USA</value>
20. <value>UK</value>
21. </set>
22. </property>
24. <!-- java.util.Map -->
25. <property name="customMap">
26. <map>
27. <entry key="1" value="INDIA"/>
28. <entry key="2" value="Pakistan"/>
29. <entry key="3" value="USA"/>
30. <entry key="4" value="UK"/>
31. </map>
32. </property>
34. <!-- java.util.Properties -->
35. <property name="customProperies">
36. <props>
37. <prop key="admin">admin@nospam.com</prop>
38. <prop key="support">support@nospam.com</prop>
39. </props>
40. </property>
42. </bean>
43. </beans>

<beans>

<!-- Definition for javaCollection -->

<bean id="javaCollection" class="com.howtodoinjava.JavaCollection">

<!-- java.util.List -->

<property name="customList">

<list>

<value>INDIA</value>

<value>Pakistan</value>

<value>USA</value>

<value>UK</value>

</list>

</property>

<!-- java.util.Set -->

<property name="customSet">

<set>

<value>INDIA</value>

<value>Pakistan</value>

<value>USA</value>

<value>UK</value>

</set>

</property>

<!-- java.util.Map -->

<property name="customMap">

<map>

<entry key="1" value="INDIA"/>

<entry key="2" value="Pakistan"/>

<entry key="3" value="USA"/>

<entry key="4" value="UK"/>

</map>

</property>

<!-- java.util.Properties -->

<property name="customProperies">

<props>

<prop key="admin">admin@nospam.com</prop>

<prop key="support">support@nospam.com</prop>

</props>

</property>

</bean>

</beans>

### 15、如何向Spring Bean中注入一个Java.util.Properties？

第一种方法是使用如下面代码所示的<props> 标签：

**[html]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. <bean id="adminUser" class="com.somnus.common.Customer">
3. <!-- java.util.Properties -->
4. <property name="emails">
5. <props>
6. <prop key="admin">admin@nospam.com</prop>
7. <prop key="support">support@nospam.com</prop>
8. </props>
9. </property>
11. </bean>

<bean id="adminUser" class="com.somnus.common.Customer">

<!-- java.util.Properties -->

<property name="emails">

<props>

<prop key="admin">admin@nospam.com</prop>

<prop key="support">support@nospam.com</prop>

</props>

</property>

</bean>

也可用”util:”命名空间来从properties文件中创建出一个propertiesbean，然后利用setter方法注入bean的引用。

### 16、请解释Spring Bean的自动装配？

在Spring框架中，在配置文件中设定bean的依赖关系是一个很好的机制，Spring容器还可以自动装配合作关系bean之间的关联关系。这意味着Spring可以通过向Bean Factory中注入的方式自动搞定bean之间的依赖关系。自动装配可以设置在每个bean上，也可以设定在特定的bean上。

下面的XML配置文件表明了如何根据名称将一个bean设置为自动装配：

**[html]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. <bean id="employeeDAO" class="com.howtodoinjava.EmployeeDAOImpl" autowire="byName" />

<bean id="employeeDAO" class="com.howtodoinjava.EmployeeDAOImpl" autowire="byName" />

除了bean配置文件中提供的自动装配模式，还可以使用@Autowired注解来自动装配指定的bean。在使用@Autowired注解之前需要在按照如下的配置方式在Spring配置文件进行配置才可以使用。

**[html]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. <context:annotation-config />

<context:annotation-config />

也可以通过在配置文件中配置AutowiredAnnotationBeanPostProcessor 达到相同的效果。

**[html]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. <bean class ="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"/>

<bean class ="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"/>

配置好以后就可以使用@Autowired来标注了。

**[java]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. @Autowired
2. public EmployeeDAOImpl ( EmployeeManager manager ) {
3. this.manager = manager;
4. }

@Autowired

public EmployeeDAOImpl ( EmployeeManager manager ) {

this.manager = manager;

}

### 17、请解释自动装配模式的区别？

在Spring框架中共有5种自动装配，让我们逐一分析。

1. no：这是Spring框架的默认设置，在该设置下自动装配是关闭的，开发者需要自行在bean定义中用标签明确的设置依赖关系。
2. byName：该选项可以根据bean名称设置依赖关系。当向一个bean中自动装配一个属性时，容器将根据bean的名称自动在在配置文件中查询一个匹配的bean。如果找到的话，就装配这个属性，如果没找到的话就报错。
3. byType：该选项可以根据bean类型设置依赖关系。当向一个bean中自动装配一个属性时，容器将根据bean的类型自动在在配置文件中查询一个匹配的bean。如果找到的话，就装配这个属性，如果没找到的话就报错。
4. constructor：造器的自动装配和byType模式类似，但是仅仅适用于与有构造器相同参数的bean，如果在容器中没有找到与构造器参数类型一致的bean，那么将会抛出异常。
5. autodetect：该模式自动探测使用构造器自动装配或者byType自动装配。首先，首先会尝试找合适的带参数的构造器，如果找到的话就是用构造器自动装配，如果在bean内部没有找到相应的构造器或者是无参构造器，容器就会自动选择byTpe的自动装配方式。

### 18、如何开启基于注解的自动装配？

要使用 @Autowired，需要注册 AutowiredAnnotationBeanPostProcessor，可以有以下两种方式来实现：

1、引入配置文件中的<bean>下引入 <context:annotation-config>

**[html]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. <beans>
2. <context:annotation-config />
3. </beans>

<beans>

<context:annotation-config />

</beans>

2、在bean配置文件中直接引入AutowiredAnnotationBeanPostProcessor

**[html]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. <beans>
2. <bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"/>
3. </beans>

<beans>

<bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"/>

</beans>

### 19、请举例解释@Required注解？

在产品级别的应用中，IoC容器可能声明了数十万了bean，bean与bean之间有着复杂的依赖关系。设值注解方法的短板之一就是验证所有的属性是否被注解是一项十分困难的操作。可以通过在<bean>中设置“dependency-check”来解决这个问题。

在应用程序的生命周期中，你可能不大愿意花时间在验证所有bean的属性是否按照上下文文件正确配置。或者你宁可验证某个bean的特定属性是否被正确的设置。即使是用“dependency-check”属性也不能很好的解决这个问题，在这种情况下，你需要使用@Required 注解。

需要用如下的方式使用来标明bean的设值方法。

**[java]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. public class EmployeeFactoryBean extends AbstractFactoryBean<Object>{
2. private String designation;
3. public String getDesignation() {
4. return designation;
5. }
6. @Required
7. public void setDesignation(String designation) {
8. this.designation = designation;
9. }
10. //more code here
11. }

public class EmployeeFactoryBean extends AbstractFactoryBean<Object>{

private String designation;

public String getDesignation() {

return designation;

}

@Required

public void setDesignation(String designation) {

this.designation = designation;

}

//more code here

}

RequiredAnnotationBeanPostProcessor是Spring中的后置处理用来验证被@Required 注解的bean属性是否被正确的设置了。在使用RequiredAnnotationBeanPostProcesso来验证bean属性之前，首先要在IoC容器中对其进行注册：

**[html]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. <bean class="org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor" />

<bean class="org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor" />

但是如果没有属性被用 @Required 注解过的话，后置处理器会抛出一个BeanInitializationException 异常。

### 20、请举例解释@Autowired注解？

@Autowired注解对自动装配何时何处被实现提供了更多细粒度的控制。@Autowired注解可以像@Required注解、构造器一样被用于在bean的设值方法上自动装配bean的属性，一个参数或者带有任意名称或带有多个参数的方法。

比如，可以在设值方法上使用@Autowired注解来替代配置文件中的 <property>元素。当Spring容器在setter方法上找到@Autowired注解时，会尝试用byType 自动装配。

当然我们也可以在构造方法上使用@Autowired 注解。带有@Autowired 注解的构造方法意味着在创建一个bean时将会被自动装配，即便在配置文件中使用<constructor-arg> 元素。

**[java]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. public class TextEditor {
2. private SpellChecker spellChecker;
3. @Autowired
4. public TextEditor(SpellChecker spellChecker){
5. System.out.println("Inside TextEditor constructor." );
6. this.spellChecker = spellChecker;
7. }
8. public void spellCheck(){
9. spellChecker.checkSpelling();
10. }
11. }

public class TextEditor {

private SpellChecker spellChecker;

@Autowired

public TextEditor(SpellChecker spellChecker){

System.out.println("Inside TextEditor constructor." );

this.spellChecker = spellChecker;

}

public void spellCheck(){

spellChecker.checkSpelling();

}

}

下面是没有构造参数的配置方式：

**[html]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. <beans>
3. <context:annotation-config/>
5. <!-- Definition for textEditor bean without constructor-arg  -->
6. <bean id="textEditor" class="com.howtodoinjava.TextEditor"/>
8. <!-- Definition for spellChecker bean -->
9. <bean id="spellChecker" class="com.howtodoinjava.SpellChecker"/>
11. </beans>

<beans>

<context:annotation-config/>

<!-- Definition for textEditor bean without constructor-arg -->

<bean id="textEditor" class="com.howtodoinjava.TextEditor"/>

<!-- Definition for spellChecker bean -->

<bean id="spellChecker" class="com.howtodoinjava.SpellChecker"/>

</beans>

### 21、请举例说明@Qualifier注解？

@Qualifier注解意味着可以在被标注bean的字段上可以自动装配。Qualifier注解可以用来取消Spring不能取消的bean应用。

下面的示例将会在Customer的person属性中自动装配person的值。

**[java]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. public class Customer{
2. @Autowired
3. private Person person;
4. }

public class Customer{

@Autowired

private Person person;

}

下面我们要在配置文件中来配置Person类。

**[html]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. <bean id="customer" class="com.somnus.common.Customer" />
3. <bean id="personA" class="com.somnus.common.Person" >
4. <property name="name" value="lokesh" />
5. </bean>
7. <bean id="personB" class="com.somnus.common.Person" >
8. <property name="name" value="alex" />
9. </bean>

<bean id="customer" class="com.somnus.common.Customer" />

<bean id="personA" class="com.somnus.common.Person" >

<property name="name" value="lokesh" />

</bean>

<bean id="personB" class="com.somnus.common.Person" >

<property name="name" value="alex" />

</bean>

Spring会知道要自动装配哪个person bean么？不会的，但是运行上面的示例时，会抛出下面的异常：

**[html]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. Caused by: org.springframework.beans.factory.NoSuchBeanDefinitionException:
2. No unique bean of type [com.howtodoinjava.common.Person] is defined:
3. expected single matching bean but found 2: [personA, personB]

Caused by: org.springframework.beans.factory.NoSuchBeanDefinitionException:

No unique bean of type [com.howtodoinjava.common.Person] is defined:

expected single matching bean but found 2: [personA, personB]

要解决上面的问题，需要使用 @Quanlifier注解来告诉Spring容器要装配哪个bean：

**[java]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. public class Customer{
2. @Autowired
3. @Qualifier("personA")
4. private Person person;
5. }

public class Customer{

@Autowired

@Qualifier("personA")

private Person person;

}

### 22、构造方法注入和设值注入有什么区别？

请注意以下明显的区别：

1. 在设值注入方法支持大部分的依赖注入，如果我们仅需要注入int、string和long型的变量，我们不要用设值的方法注入。对于基本类型，如果我们没有注入的话，可以为基本类型设置默认值。在构造方法注入不支持大部分的依赖注入，因为在调用构造方法中必须传入正确的构造参数，否则的话为报错。
2. 设值注入不会重写构造方法的值。如果我们对同一个变量同时使用了构造方法注入又使用了设置方法注入的话，那么构造方法将不能覆盖由设值方法注入的值。很明显，因为构造方法尽在对象被创建时调用。
3. 在使用设值注入时有可能还不能保证某种依赖是否已经被注入，也就是说这时对象的依赖关系有可能是不完整的。而在另一种情况下，构造器注入则不允许生成依赖关系不完整的对象。
4. 在设值注入时如果对象A和对象B互相依赖，在创建对象A时Spring会抛出sObjectCurrentlyInCreationException异常，因为在B对象被创建之前A对象是不能被创建的，反之亦然。所以Spring用设值注入的方法解决了循环依赖的问题，因对象的设值方法是在对象被创建之前被调用的。

### 23、Spring框架中有哪些不同类型的事件？

Spring的ApplicationContext 提供了支持事件和代码中监听器的功能。

我们可以创建bean用来监听在ApplicationContext 中发布的事件。ApplicationEvent类和在ApplicationContext接口中处理的事件，如果一个bean实现了ApplicationListener接口，当一个ApplicationEvent 被发布以后，bean会自动被通知。

**[java]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. public class AllApplicationEventListener implements ApplicationListener < ApplicationEvent >{
2. @Override
3. public void onApplicationEvent(ApplicationEvent applicationEvent)
4. {
5. //process event
6. }
7. }

public class AllApplicationEventListener implements ApplicationListener < ApplicationEvent >{

@Override

public void onApplicationEvent(ApplicationEvent applicationEvent)

{

//process event

}

}

Spring 提供了以下5中标准的事件：

1. 上下文更新事件（ContextRefreshedEvent）：该事件会在ApplicationContext被初始化或者更新时发布。也可以在调用ConfigurableApplicationContext 接口中的refresh()方法时被触发。
2. 上下文开始事件（ContextStartedEvent）：当容器调用ConfigurableApplicationContext的Start()方法开始/重新开始容器时触发该事件。
3. 上下文停止事件（ContextStoppedEvent）：当容器调用ConfigurableApplicationContext的Stop()方法停止容器时触发该事件。
4. 上下文关闭事件（ContextClosedEvent）：当ApplicationContext被关闭时触发该事件。容器被关闭时，其管理的所有单例Bean都被销毁。
5. 请求处理事件（RequestHandledEvent）：在Web应用中，当一个http请求（request）结束触发该事件。

除了上面介绍的事件以外，还可以通过扩展ApplicationEvent 类来开发自定义的事件。

**[java]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. public class CustomApplicationEvent extends ApplicationEvent{
2. public CustomApplicationEvent ( Object source, final String msg ){
3. super(source);
4. System.out.println("Created a Custom event");
5. }
6. }

public class CustomApplicationEvent extends ApplicationEvent{

public CustomApplicationEvent ( Object source, final String msg ){

super(source);

System.out.println("Created a Custom event");

}

}

为了监听这个事件，还需要创建一个监听器：

**[java]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. public class CustomEventListener implements ApplicationListener < CustomApplicationEvent >{
2. @Override
3. public void onApplicationEvent(CustomApplicationEvent applicationEvent) {
4. //handle event
5. }
6. }

public class CustomEventListener implements ApplicationListener < CustomApplicationEvent >{

@Override

public void onApplicationEvent(CustomApplicationEvent applicationEvent) {

//handle event

}

}

之后通过applicationContext接口的publishEvent()方法来发布自定义事件。

**[java]** [view plain](https://blog.csdn.net/u014079773/article/details/52453002" \o "view plain) [copy](https://blog.csdn.net/u014079773/article/details/52453002" \o "copy) [print](https://blog.csdn.net/u014079773/article/details/52453002" \o "print)[?](https://blog.csdn.net/u014079773/article/details/52453002" \o "?)

1. CustomApplicationEvent customEvent = new CustomApplicationEvent(applicationContext, "Test message");
2. applicationContext.publishEvent(customEvent);

CustomApplicationEvent customEvent = new CustomApplicationEvent(applicationContext, "Test message");

applicationContext.publishEvent(customEvent);

### 24、FileSystemResource和ClassPathResource有何区别？

在FileSystemResource 中需要给出spring-config.xml文件在你项目中的相对路径或者绝对路径。在ClassPathResource中spring会在ClassPath中自动搜寻配置文件，所以要把ClassPathResource 文件放在ClassPath下。

如果将spring-config.xml保存在了src文件夹下的话，只需给出配置文件的名称即可，因为src文件夹是默认。

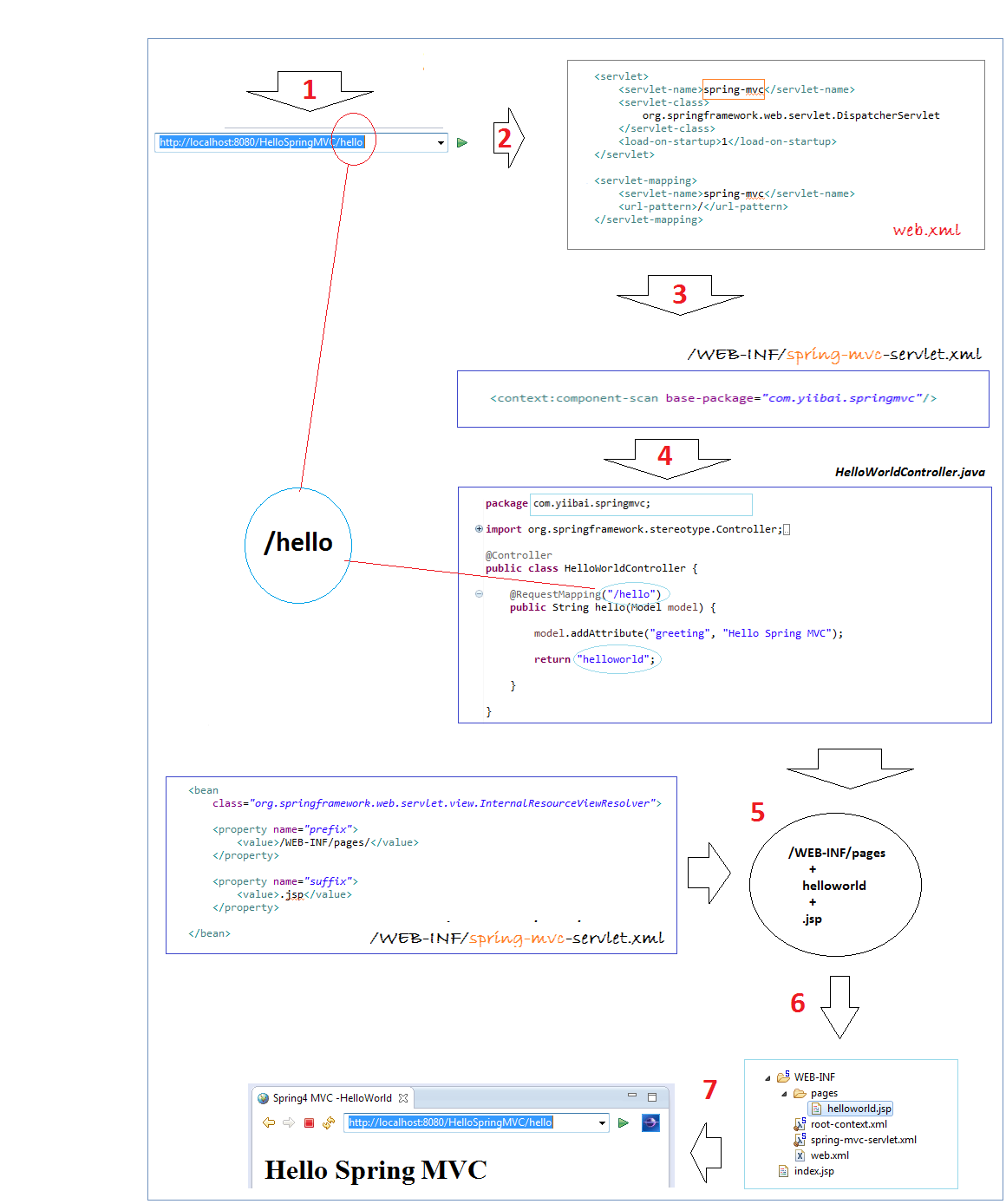
简而言之，ClassPathResource在环境变量中读取配置文件，FileSystemResource在配置文件中读取配置文件。

### 25、Spring 框架中都用到了哪些设计模式？

Spring框架中使用到了大量的设计模式，下面列举了比较有代表性的：

* 代理模式—在AOP和remoting中被用的比较多。
* 单例模式—在spring配置文件中定义的bean默认为单例模式。
* 模板方法—用来解决代码重复的问题。比如. [RestTemplate](http://howtodoinjava.com/2015/02/20/spring-restful-client-resttemplate-example/" \o "Spring RESTFul Client – RestTemplate Example" \t "https://blog.csdn.net/u014079773/article/details/_blank), JmsTemplate, JpaTemplate。
* 前端控制器—Spring提供了DispatcherServlet来对请求进行分发。
* 视图帮助(View Helper )—Spring提供了一系列的JSP标签，高效宏来辅助将分散的代码整合在视图里。
* 依赖注入—贯穿于BeanFactory / ApplicationContext接口的核心理念。
* 工厂模式—BeanFactory用来创建对象的实例。

### 应用程序流程



**@ResponseBody**

**@PathVariable**

参数

**@RequestParam**

redirect:

### Spring 容器创建对象

****IOC 让程序员不在关注怎么去创建对象，而是关注与对象创建之后的操作，把对象的创建、初始化、销毁等工作交给spring容器来做。****

**第一种方法：利用默认的构造方法**

　　在 src 目录下新建 applicationContext.xml 文件，这是 spring 配置文件，添加如下代码：

[?](https://www.cnblogs.com/ysocean/p/7466217.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"      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.xsd">      <!--      创建对象的第一种方式：利用无参构造器      id:唯一标识符      class:类的全类名        -->      <bean id="helloIoc" class="com.ys.ioc.HelloIoc" ></bean>      <!-- 别名属性  name：和bean的 id 属性对应 -->      <alias name="helloIoc" alias="helloIoc2"/>    </beans> |

　　测试代码：

[?](https://www.cnblogs.com/ysocean/p/7466217.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | /\*\*       \*  Spring 容器利用构造函数创建对象       \*/      @Test      public void testCreateObjectByConstrutor(){          //1、启动 spring 容器          ApplicationContext context =                  new ClassPathXmlApplicationContext("applicationContext.xml");          //2、从 spring 容器中取出数据          HelloIoc IOC = (HelloIoc) context.getBean("helloIoc");          //3、通过对象调用方法          IOC.sayHello();            //利用配置文件 alias 别名属性创建对象          HelloIoc IOC2 = (HelloIoc) context.getBean("helloIoc2");          IOC2.sayHello();      } |

　　我们可以在测试类 HelloIoc.java 中手动添加无参的构造方法，然后执行上面的测试代码，会发现构造方法会在 sayHello()方法执行之前调用。

**第二种方法：利用静态工厂方法**

首先创建静态工厂类 HelloStaticFactory.java

[?](https://www.cnblogs.com/ysocean/p/7466217.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | package com.ys.ioc;    public class HelloStaticFactory {      public HelloStaticFactory(){          System.out.println("HelloStaticFactory constructor");      }      //静态工厂方法      public static HelloIoc getInstances(){          return new HelloIoc();      }    } |

　　接着在 applicationContext.xml 中进行如下配置：

[?](https://www.cnblogs.com/ysocean/p/7466217.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <!--          创建对象的第二种方式：利用静态工厂方法          factory-method：静态工厂类的获取对象的静态方法          class:静态工厂类的全类名        -->      <bean id="helloStaticFactory" factory-method="getInstances" class="com.ys.ioc.HelloStaticFactory"></bean> |

　　编写测试类：

[?](https://www.cnblogs.com/ysocean/p/7466217.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | /\*\*       \* Spring 容器利用静态工厂方法创建对象       \*/      @Test      public void createObjectStaticFactory(){          ApplicationContext context =                  new ClassPathXmlApplicationContext("applicationContext.xml");          HelloIoc staticFactory =                  (HelloIoc) context.getBean("helloStaticFactory");          staticFactory.sayHello();      } |

**注意：spring容器只负责调用静态工厂方法，而这个静态工厂方法内部实现由程序员完成**

**第三种方法：利用实例工厂方法**

首先创建实例工厂类 HelloInstanceFactory .java

[?](https://www.cnblogs.com/ysocean/p/7466217.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | package com.ys.ioc;    public class HelloInstanceFactory {      public HelloInstanceFactory(){          System.out.println("实例工厂方法构造函数");      }        //利用实例工厂方法创建对象      public HelloIoc getInstance(){          HelloIoc instanceIoc = new HelloIoc();          return instanceIoc;      }  } |

　　接着在 applicationContext.xml 中进行如下配置：

[?](https://www.cnblogs.com/ysocean/p/7466217.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <!--          创建对象的第三种方式：利用实例工厂方法          factory-bean:指定当前Spring中包含工厂方法的beanID          factory-method:工厂方法名称        -->      <bean id="instanceFactory" class="com.ys.ioc.HelloInstanceFactory"></bean>      <bean id="instance" factory-bean="instanceFactory" factory-method="getInstance"></bean> |

　　最后编写测试类：

[?](https://www.cnblogs.com/ysocean/p/7466217.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | /\*\*       \* Spring 容器利用实例工厂方法创建对象       \*/      @Test      public void createObjectInstanceFactory(){          ApplicationContext context =                  new ClassPathXmlApplicationContext("applicationContext.xml");          HelloIoc staticFactory =                  (HelloIoc) context.getBean("instance");          staticFactory.sayHello();      } |

### Spring 容器创建对象的时机

****第一种：默认情况下，启动 spring 容器便创建对象（遇到bean便创建对象）****

**第二种：在spring的配置文件bean中有一个属性 lazy-init="default/true/false"**

**①、如果lazy-init为"default/false"在启动spring容器时创建对象（默认情况）**

**②、如果lazy-init为"true",在context.getBean时才要创建对象**

**比较：**

在第一种情况下可以在启动spring容器的时候，检查spring容器配置文件的正确性，如果再结合tomcat,如果spring容器不能正常启动，整个tomcat就不能正常启动。但是这样的缺点是把一些bean过早的放在了内存中，如果有数据，则对内存来是一个消耗。

反过来，在第二种情况下，可以减少内存的消耗，但是不容易发现错误

##### Spring bean的scope:

##### singleton/prototype/request/session/global session

**一、默认scope的值是singleton，即产生的对象是单例的**

 　　applicationContext.xml 文件中配置：

|  |  |
| --- | --- |
| 1 | <bean id="helloIoc" scope="singleton" class="com.ys.ioc.HelloIoc" ></bean> |

**二、scope=“prototype”**

**多例模式，并且spring容器启动的时候并不会创建对象，而是在得到 bean 的时候才会创建对象**

 　　applicationContext.xml 文件中配置：

|  |  |
| --- | --- |
| 1 | <bean id="helloIoc" scope="prototype" class="com.ys.ioc.HelloIoc" ></bean> |

****总结：在单例模式下，启动 spring 容器，便会创建对象；在多例模式下，启动容器并不会创建对象，获得 bean 的时候才会创建对象****

##### Spring 容器的生命周期

applicationContext.xml

|  |  |
| --- | --- |
| 1  2 | <!-- 生命周期 -->      <bean id="springLifeCycle" init-method="init" destroy-method="destroy" class="com.ys.ioc.SpringLifeCycle"></bean> |

　　测试：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | //spring 容器的初始化和销毁      @Test      public void testSpringLifeCycle(){          ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");          SpringLifeCycle hello = (SpringLifeCycle) context.getBean("springLifeCycle");            hello.sayHello();            //销毁spring容器          ClassPathXmlApplicationContext classContext = (ClassPathXmlApplicationContext) context;          classContext.close();      } |

分析：spring 容器的声明周期

　　 　   1、spring容器创建对象  
        　　2、执行init方法  
        　　3、调用自己的方法  
        　　4、当spring容器关闭的时候执行destroy方法

**注意：当scope为"prototype"时，调用 close（） 方法时是不会调用 destroy 方法的**

**Spring DI（依赖注入）**

spring动态的向某个对象提供它所需要的其他对象。这一点是通过DI（Dependency Injection，依赖注入）来实现的。Java 1.3之后一个重要特征是反射（reflection），它允许程序在运行的时候动态的生成对象、执行对象的方法、改变对象的属性，spring就是通过反射来实现注入的。

　简单来说什么是依赖注入，就是给属性赋值（包括基本数据类型和引用数据类型）

Spring中对象赋值：

1. 利用set方法给属性赋值

<!--

    property是用来描述一个类的属性

    基本类型的封装类、String等需要值的类型用value赋值

    引用类型用ref赋值

-->

<bean id="person" class="com.ys.di.Person">

    <property name="pid" value="1"></property>

    <property name="pname" value="vae"></property>

    <property name="students">

        <ref bean="student"/>

    </property>

    <property name="lists">

        <list>

            <value>1</value>

            <ref bean="student"/>

            <value>vae</value>

        </list>

    </property>

    <property name="sets">

        <set>

            <value>1</value>

            <ref bean="student"/>

            <value>vae</value>

        </set>

    </property>

    <property name="maps">

        <map>

            <entry key="m1" value="1"></entry>

            <entry key="m2" >

                <ref bean="student"/>

            </entry>

        </map>

    </property>

    <property name="properties">

        <props>

            <prop key="p1">p1</prop>

            <prop key="p2">p2</prop>

        </props>

    </property>

</bean>

<bean id="student" class="com.ys.di.Student"></bean>

1. 利用构造函数给属性赋值

第一步：在实体类 Per'son.java 中添加两个构造方法：有参和无参

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | //默认构造函数      public Person(){}      //带参构造函数      public Person(Long pid,Student students){          this.pid = pid;          this.students = students;      } |

　　第二步：在 applicationContext.xml 中进行赋值

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | <!-- 根据构造函数赋值 -->      <!--          index  代表参数的位置  从0开始计算          type   指的是参数的类型,在有多个构造函数时，可以用type来区分，要是能确定是那个构造函数，可以不用写type          value  给基本类型赋值          ref    给引用类型赋值        -->      <bean id="person\_con" class="com.ys.di.Person">          <constructor-arg index="0" type="java.lang.Long" value="1">          </constructor-arg>          <constructor-arg index="1" type="com.ys.di.Student" ref="student\_con"></constructor-arg>      </bean>      <bean id="student\_con" class="com.ys.di.Student"></bean> |

总结：

　　1、如果spring的配置文件中的bean中没有<constructor-arg>该元素，则调用默认的构造函数

　　2、如果spring的配置文件中的bean中有<constructor-arg>该元素，则该元素确定唯一的构造函数

##### Spring AOP（面向切面编程）

AOP（Aspect Oriented Programming），通常称为面向切面编程。它利用一种称为"横切"的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其命名为"Aspect"，即切面。所谓"切面"，简单说就是那些与业务无关，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，降低模块之间的耦合度，并有利于未来的可操作性和可维护性。

什么是切面，什么是公共模块，那么我们概念少说，直接通过一个实例来看看 AOP 到底是什么。

**AOP 实现原理也是动态代理**

##### AOP关键术语：

1.target：目标类，需要被代理的类。例如：UserService

　　2.Joinpoint(连接点):所谓连接点是指那些可能被拦截到的方法。例如：所有的方法

　　3.PointCut 切入点：已经被增强的连接点。例如：addUser()

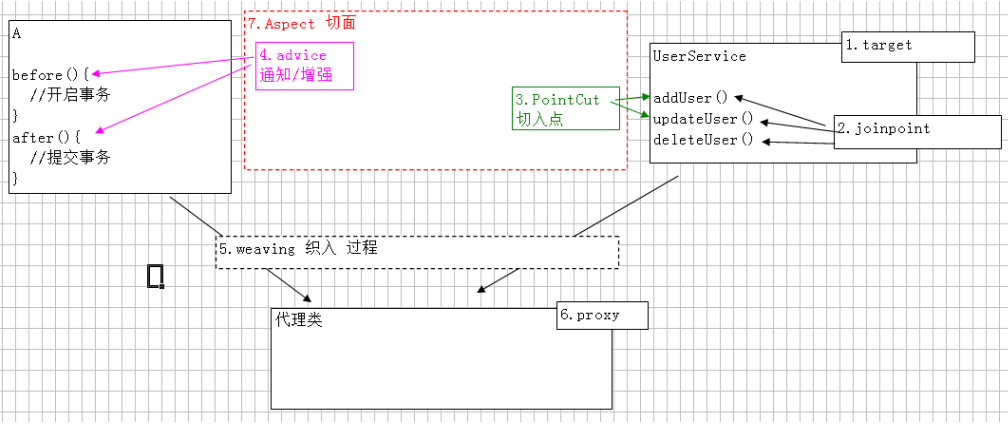
　　4.advice 通知/增强，增强代码。例如：after、before

　　5. Weaving(织入):是指把增强advice应用到目标对象target来创建新的代理对象proxy的过程.

　　6.proxy 代理类：通知+切入点

　　7. Aspect(切面): 是切入点pointcut和通知advice的结合

　　具体可以根据下面这张图来理解：



**Spring框架的7个模块**

****核心容器：****核心容器提供 Spring 框架的基本功能。核心容器的主要组件是 BeanFactory，它是工厂模式的实现。BeanFactory 使用控制反转 （IOC）模式将应用程序的配置和依赖性规范与实际的应用程序代码分开。  
     ****Spring 上下文：****Spring 上下文是一个配置文件，向 Spring 框架提供上下文信息。Spring 上下文包括企业服务，例如 JNDI、EJB、电子邮件、国际化、校验和调度功能。  
     ****Spring AOP：****通过配置管理特性，Spring AOP 模块直接将面向方面的编程功能集成到了 Spring 框架中。所以，可以很容易地使 Spring 框架管理的任何对象支持 AOP。Spring AOP 模块为基于 Spring 的应用程序中的对象提供了事务管理服务。通过使用 Spring AOP，不用依赖 EJB 组件，就可以将声明性事务管理集成到应用程序中。  
     ****Spring DAO：****JDBC DAO 抽象层提供了有意义的异常层次结构，可用该结构来管理异常处理和不同数据库供应商抛出的错误消息。异常层次结构简化了错误处理，并且极大地降低了需要编写的异常代码数量（例如打开和关闭连接）。Spring DAO 的面向 JDBC 的异常遵从通用的 DAO 异常层次结构。  
     ****Spring ORM：****Spring 框架插入了若干个 ORM 框架，从而提供了 ORM 的对象关系工具，其中包括 JDO、Hibernate 和 iBatis SQL Map。所有这些都遵从 Spring 的通用事务和 DAO 异常层次结构。  
     ****Spring Web 模块：****Web 上下文模块建立在应用程序上下文模块之上，为基于 Web 的应用程序提供了上下文。所以，Spring 框架支持与 Jakarta Struts 的集成。Web 模块还简化了处理多部分请求以及将请求参数绑定到域对象的工作。  
     ****Spring MVC 框架：****MVC 框架是一个全功能的构建 Web 应用程序的 MVC 实现。通过策略接口，MVC 框架变成为高度可配置的，MVC 容纳了大量视图技术，其中包括 JSP、Velocity、Tiles、iText 和 POI。

## **15.解释一下Spring AOP里面的几个名词**

切面（Aspect）：一个关注点的模块化，这个关注点可能会横切多个对象。事务管理是J2EE应用中一个关于横切关注点的很好的例子。 在Spring AOP中，切面可以使用通用类（基于模式的风格） 或者在普通类中以 @Aspect 注解（@AspectJ风格）来实现。

连接点（Joinpoint）：在程序执行过程中某个特定的点，比如某方法调用的时候或者处理异常的时候。 在Spring AOP中，一个连接点 总是 代表一个方法的执行。 通过声明一个org.aspectj.lang.JoinPoint类型的参数可以使通知（Advice）的主体部分获得连接点信息。

通知（Advice）：在切面的某个特定的连接点（Joinpoint）上执行的动作。通知有各种类型，其中包括“around”、“before”和“after”等通知。 通知的类型将在后面部分进行讨论。许多AOP框架，包括Spring，都是以拦截器做通知模型， 并维护一个以连接点为中心的拦截器链。

切入点（Pointcut）：匹配连接点（Joinpoint）的断言。通知和一个切入点表达式关联，并在满足这个切入点的连接点上运行（例如，当执行某个特定名称的方法时）。 切入点表达式如何和连接点匹配是AOP的核心：Spring缺省使用AspectJ切入点语法。

引入（Introduction）：（也被称为内部类型声明（inter-type declaration））。声明额外的方法或者某个类型的字段。 Spring允许引入新的接口（以及一个对应的实现）到任何被代理的对象。例如，你可以使用一个引入来使bean实现 IsModified 接口，以便简化缓存机制。

目标对象（Target Object）： 被一个或者多个切面（aspect）所通知（advise）的对象。也有人把它叫做 被通知（advised） 对象。 既然Spring AOP是通过运行时代理实现的，这个对象永远是一个 被代理（proxied） 对象。

AOP代理（AOP Proxy）： AOP框架创建的对象，用来实现切面契约（aspect contract）（包括通知方法执行等功能）。 在Spring中，AOP代理可以是JDK动态代理或者CGLIB代理。 注意：Spring 2.0最新引入的基于模式（schema-based）风格和@AspectJ注解风格的切面声明，对于使用这些风格的用户来说，代理的创建是透明的。

织入（Weaving）：把切面（aspect）连接到其它的应用程序类型或者对象上，并创建一个被通知（advised）的对象。 这些可以在编译时（例如使用AspectJ编译器），类加载时和运行时完成。 Spring和其他纯Java AOP框架一样，在运行时完成织入。

