Spring常见面试：

<https://blog.csdn.net/hrbeuwhw/article/details/79476988>

<https://blog.csdn.net/a745233700/article/details/80959716>

<https://www.cnblogs.com/huajiezh/p/5790946.html>

# Spring

## Spring是什么?

        Spring是一个轻量级的IoC和AOP容器框架。目的是解决企业应用开发的复杂性，使用基本的JavaBean来完成以前只可能由EJB完成的事情，并提供了更多的企业应用功能，Spring的用途不仅限于服务器端的开发，从简单性、可测试性和松耦合的角度而言，任何Java应用都可以从Spring中受益。

## Spring 的优点？

（1）spring属于低侵入式设计，代码的污染极低；

（2）spring的DI机制降低了业务对象替换的复杂性；

（3）容器提供了AOP技术，利用它很容易实现如权限拦截，运行期监控等功能；

（4）降低了组件之间的耦合性 ，实现了软件各层之间的解耦；

（5）容器提供单例模式支持；

（6）可以使用容器提供的众多服务，如事务管理，消息服务等；

（7）容器提供了众多的辅助类，能加快应用的开发；

（8）spring对于主流的应用框架提供了集成支持，如hibernate，JPA，Struts等

（9）独立于各种应用服务器

（10）Spring的高度开放性，并不强制应用完全依赖于Spring，开发者可以自由选择spring的部分或全部。

## 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。

## Spring的AOP理解：

        AOP，一般称为面向方面（切面）编程，作为面向对象的一种补充，用于解剖封装好的对象内部，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。可用于权限认证、日志、事务处理。

        AOP实现的关键在于AOP框架自动创建的AOP代理，AOP代理主要分为静态代理和动态代理。静态代理的代表为AspectJ；动态代理则以Spring AOP为代表。

（1）AspectJ是静态代理的增强，所谓静态代理，就是AOP框架会在编译阶段生成AOP代理类，因此也称为编译时增强，他会在编译阶段将AspectJ织入到Java字节码中，运行的时候就是增强之后的AOP对象。

（2）Spring AOP使用的动态代理，所谓的动态代理就是说AOP框架不会去修改字节码，而是每次运行时在内存中临时为方法生成一个AOP对象，这个AOP对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。

Spring AOP中的动态代理主要有两种方式，JDK动态代理和CGLIB动态代理：

        ①JDK动态代理通过反射来接收被代理的类，并且要求被代理的类必须实现一个接口。JDK动态代理的核心是InvocationHandler接口和Proxy类。生成的代理对象的方法调用都会委托到InvocationHandler.invoke()方法，当我们调用代理类对象的方法时，这个“调用”会转送到invoke方法中，代理类对象作为proxy参数传入，参数method标识了我们具体调用的是代理类的哪个方法，args为这个方法的参数。

        ②如果目标类没有实现接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成指定类的一个子类对象，并覆盖其中特定方法，覆盖方法时可以添加增强代码，从而实现AOP。CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动态代理的。

（3）静态代理与动态代理区别在于生成AOP代理对象的时机不同，相对来说AspectJ的静态代理方式具有更好的性能，但是AspectJ需要特定的编译器进行处理，而Spring AOP则无需特定的编译器处理。

Spring的IoC理解：

（1）IOC就是控制反转。就是对象的创建权反转交给Spring，由容器控制程序之间的依赖关系，作用是实现了程序的解耦合，而非传统实现中，由程序代码直接操控。(依赖)控制权由应用代码本身转到了外部容器，由容器根据配置文件去创建实例并管理各个实例之间的依赖关系，控制权的转移，是所谓反转，并且由容器动态的将某种依赖关系注入到组件之中。BeanFactory 是Spring IoC容器的具体实现与核心接口，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，用来包装和管理各种bean。

（2）最直观的表达就是，IOC让对象的创建不用去new了，可以由spring自动生产，这里用的就是java的反射机制，通过反射在运行时动态的去创建、调用对象。spring就是根据配置文件在运行时动态的去创建对象，并调用对象的方法的。

（3）Spring的IOC有三种注入方式 ：

        第一是根据属性注入，也叫set方法注入；

        第二种是根据构造方法进行注入；

        第三种是根据注解进行注入。

详细的说：

（4）IoC，控制反转：将对象交给容器管理，你只需要在spring配置文件总配置相应的bean，以及设置相关的属性，让spring容器生成类的实例对象以及管理对象。在spring容器启动的时候，spring会把你在配置文件中配置的bean都初始化以及装配好，然后在你需要调用的时候，就把它已经初始化好的那些bean分配给你需要调用这些bean的类。就是将对象的控制权反转给spring容器管理。

（5）DI机制（Dependency Injection，依赖注入）：可以说是IoC的其中一个内容，在容器实例化对象的时候主动的将被调用者（或者说它的依赖对象）注入给调用对象。比如对象A需要操作数据库，以前我们总是要在A中自己编写代码来获得一个Connection对象，有了 spring我们就只需要告诉spring，A中需要一个Connection，至于这个Connection怎么构造，何时构造，A不需要知道。在系统运行时，spring会在适当的时候制造一个Connection，然后像打针一样，注射到A当中，这样就完成了对各个对象之间关系的控制。

IoC让相互协作的组件保持松散的耦合，而AOP编程允许你把遍布于应用各层的功能分离出来形成可重用的功能组件。

## BeanFactory和ApplicationContext有什么区别？

        BeanFactory和ApplicationContext是Spring的两大核心接口，而其中ApplicationContext是BeanFactory的子接口。它们都可以当做Spring的容器，生成Bean实例的，并管理容器中的Bean。

（1）BeanFactory：是Spring里面最底层的接口，提供了最简单的容器的功能，负责读取bean配置文档，管理bean的加载与实例化，维护bean之间的依赖关系，负责bean的生命周期，但是无法支持spring的aop功能和web应用。

（2）ApplicationContext接口作为BeanFactory的派生，因而具有BeanFactory所有的功能。而且ApplicationContext还在功能上做了扩展，以一种更面向框架的方式工作以及对上下文进行分层和实现继承，相较于BeanFactorty，ApplicationContext还提供了以下的功能：

①默认初始化所有的Singleton，也可以通过配置取消预初始化。

②继承MessageSource，因此支持国际化。

③资源访问，比如访问URL和文件。

④事件机制。

⑤同时加载多个配置文件。

⑥以声明式方式启动并创建Spring容器。

⑦载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的web层。

（3）①BeanFactroy采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化，这样，我们就不能发现一些存在的Spring的配置问题。如果Bean的某一个属性没有注入，BeanFacotry加载后，直至第一次使用调用getBean方法才会抛出异常。

        ②而ApplicationContext则相反，它是在容器启动时，一次性创建了所有的Bean。这样，在容器启动时，我们就可以发现Spring中存在的配置错误，这样有利于检查所依赖属性是否注入。 ApplicationContext启动后预载入所有的单实例Bean，通过预载入单实例bean ,确保当你需要的时候，你就不用等待，因为它们已经创建好了。

        ③相对于基本的BeanFactory，ApplicationContext 唯一的不足是占用内存空间。当应用程序配置Bean较多时，程序启动较慢。

（4）BeanFactory通常以编程的方式被创建，ApplicationContext还能以声明的方式创建，如使用ContextLoader。

（5）BeanFactory和ApplicationContext都支持BeanPostProcessor、BeanFactoryPostProcessor的使用，但两者之间的区别是：BeanFactory需要手动注册，而ApplicationContext则是自动注册。

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：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。

## 请解释Spring Bean的生命周期？

 首先说一下Servlet的生命周期：实例化，初始init，接收请求service，销毁destroy；

 Spring上下文中的Bean生命周期也类似，如下：

（1）实例化一个Bean－－也就是我们常说的new；

（2）按照Spring上下文对实例化的Bean进行配置－－也就是IOC注入；

（3）如果这个Bean已经实现了BeanNameAware接口，会调用它实现的setBeanName(String)方法，此处传递的就是Spring配置文件中Bean的id值；

（4）如果这个Bean已经实现了BeanFactoryAware接口，会调用它实现的setBeanFactory(setBeanFactory(BeanFactory)传递的是Spring工厂自身（可以用这个方式来获取其它Bean，只需在Spring配置文件中配置一个普通的Bean就可以）；

（5）如果这个Bean已经实现了ApplicationContextAware接口，会调用setApplicationContext(ApplicationContext)方法，传入Spring上下文（同样这个方式也可以实现步骤4的内容，但比4更好，因为ApplicationContext是BeanFactory的子接口，有更多的实现方法）；

（6）如果这个Bean关联了BeanPostProcessor接口，将会调用postProcessBeforeInitialization(Object obj, String s)方法，BeanPostProcessor经常被用作是Bean内容的更改，并且由于这个是在Bean初始化结束时调用那个的方法，也可以被应用于内存或缓存技术；

（7）如果Bean在Spring配置文件中配置了init-method属性会自动调用其配置的初始化方法。

（8）如果这个Bean关联了BeanPostProcessor接口，将会调用postProcessAfterInitialization(Object obj, String s)方法、；

注：以上工作完成以后就可以应用这个Bean了，那这个Bean是一个Singleton的，所以一般情况下我们调用同一个id的Bean会是在内容地址相同的实例，当然在Spring配置文件中也可以配置非Singleton。

（9）当Bean不再需要时，会经过清理阶段，如果Bean实现了DisposableBean这个接口，会调用那个其实现的destroy()方法；

（10）最后，如果这个Bean的Spring配置中配置了destroy-method属性，会自动调用其配置的销毁方法。

 另外我们这里描述的是应用Spring上下文Bean的生命周期，如果应用Spring的工厂也就是BeanFactory的话去掉第5步就Ok了。

## Spring中bean的加载过程：

（1）获取配置文件资源；

（2）对获取的xml资源进行一定的处理检验；

（3）处理包装资源；

（4）解析处理包装过后的资源；

（5）加载提取bean并注册(添加到beanDefinitionMap中)。

## Spring自动装配模式的区别？

在Spring框架中共有5种自动装配：

（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：构造器的自动装配和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。

## Spring 框架中都用到了哪些设计模式？

（1）代理模式—在AOP和remoting中被用的比较多。

（2）单例模式—在spring配置文件中定义的bean默认为单例模式。

（3）工厂模式—BeanFactory用来创建对象的实例。

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

（5）前端控制器—Spring提供了DispatcherServlet来对请求进行分发。

（6）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。

（7）依赖注入—贯穿于BeanFactory / ApplicationContext接口的核心理念。

## Spring事务的种类和各自的区别：

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或者直接使用底层的PlatformTransactionManager。对于编程式事务管理，spring推荐使用TransactionTemplate。

（2）声明式事务管理建立在AOP之上的。其本质是对方法前后进行拦截，然后在目标方法开始之前创建或者加入一个事务，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通过编程的方式管理事务，这样就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(或通过基于@Transactional注解的方式)，便可以将事务规则应用到业务逻辑中。

（3）显然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。声明式事务管理使业务代码不受污染，一个普通的POJO对象，只要加上注解就可以获得完全的事务支持。和编程式事务相比，声明式事务唯一不足地方是，后者的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。

## spring的事务传播行为：

spring事务的传播行为说的是当一个方法调用另一个方法时，事务该如何操作。

（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，则执行与PROPAGATION\_REQUIRED类似的操作。

## 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。

## Spring AOP里面的几个名词：

（1）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。

（2）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。

（3）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。

（4）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。

（5）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。

（6）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。

（7）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。

切入点（pointcut）和连接点（join point）匹配的概念是AOP的关键，这使得AOP不同于其它仅仅提供拦截功能的旧技术。 切入点使得定位通知（advice）可独立于OO层次。 例如，一个提供声明式事务管理的around通知可以被应用到一组横跨多个对象中的方法上（例如服务层的所有业务操作）。

## 通知有哪些类型？

（1）前置通知（Before advice）：在某连接点（join point）之前执行的通知，但这个通知不能阻止连接点前的执行（除非它抛出一个异常）。

（2）返回后通知（After returning advice）：在某连接点（join point）正常完成后执行的通知：例如，一个方法没有抛出任何异常，正常返回。

（3）抛出异常后通知（After throwing advice）：在方法抛出异常退出时执行的通知。

（4）后通知（After (finally) advice）：当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。

（5）环绕通知（Around Advice）：包围一个连接点（join point）的通知，如方法调用。这是最强大的一种通知类型。 环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回它们自己的返回值或抛出异常来结束执行。

环绕通知是最常用的一种通知类型。大部分基于拦截的AOP框架，例如Nanning和JBoss4，都只提供环绕通知。

## 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。

## @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。

## @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。

## @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。