# 69个spring面试题和答案

**Spring容器(Bean工厂)可简单分成两种**

BeanFactory

这是最基础、面向Spring的

ApplicationContext

这是在BeanFactory基础之上，面向使用Spring框架的开发者。提供了一系列的功能！

## BeanFactory

BeanFactory 体系:BeanFactory 是一个非常纯粹的 bean 容器，它是 IOC 必备的数据结构，其中 BeanDefinition 是她的基本结构，它内部维护着一个 BeanDefinition map ，并可根据 BeanDefinition 的描述进行 bean 的创建和管理。

Beandefinition 体系；BeanDefinition 用来描述 Spring 中的 Bean 对象。

BeandefinitionReader体系:BeanDefinitionReader 的作用是读取 Spring 的配置文件的内容，并将其转换成 Ioc 容器内部的数据结构：BeanDefinition。

ApplicationContext体系:它继承 BeanFactory

由于 ApplicationContext 的结构就决定了它与 BeanFactory 的不同，其主要区别有：

继承 MessageSource，提供国际化的标准访问策略。

继承 ApplicationEventPublisher ，提供强大的事件机制。

扩展 ResourceLoader，可以用来加载多个 Resource，可以灵活访问不同的资源。

对 Web 应用的支持。

**ApplicationContext和BeanFactory不同之处在于**

ApplicationContext会利用Java反射机制自动识别出配置文件中定义的BeanPostProcessor、 InstantiationAwareBeanPostProcesso 和BeanFactoryPostProcessor后置器，并自动将它们注册到应用上下文中。而BeanFactory需要在代码中通过手工调用addBeanPostProcessor()方法进行注册

ApplicationContext在初始化应用上下文的时候就实例化所有单实例的Bean。而BeanFactory在初始化容器的时候并未实例化Bean，直到第一次访问某个Bean时才实例化目标Bean。

## Spring IOC容器是怎么实现对象的创建和依赖的：

容器启动时BeanDefinitionReader会读取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；

容器扫描BeanDefinitionRegistry中的BeanDefinition；通过反射进行Bean实例化的工作；使用BeanWrapper完成Bean属性的设置工作；

单例Bean缓存池：Spring 在DefaultSingletonBeanRegistry类中提供了一个用于缓存单实例 Bean的缓存器，它是一个用HashMap实现的缓存器，单实例的Bean以beanName为键保存在这个HashMap中。

原文：Spring IOC知识点一网打尽！ https://mp.weixin.qq.com/s/TYftOpe6J1Y4EDNDxKKqXg

**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由哪些模块组成？**

以下是Spring 框架的基本模块：Core module;Bean moduleContext 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;

**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。

**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。

**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。

**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。

**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。

**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。

## 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。

**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。

**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。

## 解释Spring框架中bean的生命周期

1、Spring Ioc容器找到关于Bean的定义并实例化Bean

2、Spring Ioc容器对Bean进行依赖注入

3、如果Bean实现了BeanNameAware接口，则将该Bean的id传给setBeanName方法

4、如果Bean实现了BeanFactoryAware接口，则将BeanFactroy对象传给setBeanFactory方法

5、如果Bean实现了BeanPostProcessor接口，则调用其postProcessBeforeInitialization方法

6、如果Bean实现了InitializingBean接口，则调用其afterPropertySet方法

7、如果有和Bean关联的BeanPostProcessors对象，则这些对象的postProcessAfterInitialization方法被调用

8、如果销毁Bean实例时如果bean实现了DisposableBean接口，则调用其destroy方法

**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）。

**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。

**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。

**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。

**你可以在Spring中注入一个null 和一个空字符串吗？**

可以。

## 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。

Spring事务的传播属性

1) **REQUIRED** ，这个是默认的属性,如果存在一个事务，则支持当前事务。如果没有事务则开启一个新的事务。

2) **MANDATORY** 支持当前事务，如果当前没有事务，就抛出异常。

3) **NEVER** 以非事务方式执行，如果当前存在事务，则抛出异常。

4) **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。

5) **REQUIRES\_NEW** 新建事务，如果当前存在事务，把当前事务挂起。

6) **SUPPORTS** 支持当前事务，如果当前没有事务，就以非事务方式执行。

7) **NESTED** 支持当前事务，新增Savepoint点，与当前事务同步提交或回滚。嵌套事务一个非常重要的概念就是内层事务依赖于外层事务。外层事务失败时，会回滚内层事务所做的动作。而内层事务操作失败并不会引起外层事务的回滚。

PROPAGATION\_NESTED 与PROPAGATION\_REQUIRES\_NEW的区别

它们非常 类似,都像一个嵌套事务，如果不存在一个活动的事务，都会开启一个新的事务。使用PROPAGATION\_REQUIRES\_NEW时，内层事务与外层事务就像两个独立的事务一样，一旦内层事务进行了提交后，外层事务不能对其进行回滚。两个事务互不影响。两个事务不是一个真正的嵌套事务。同时它需要JTA 事务管理器的支持。

使用PROPAGATION\_NESTED时，外层事务的回滚可以引起内层事务的回滚。而内层事务的异常并不会导致外层事务的回滚，它是一个真正的嵌套事务。

## 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。

**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。

## 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。

**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。

**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。

**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。

**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

**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。

**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。

**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。

**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。

**有几种不同类型的自动代理？**

BeanNameAutoProxyCreator

DefaultAdvisorAutoProxyCreator

Metadata autoproxying

**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。

**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。

**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。

**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。

# 源码解析

BBP的全称叫做：BeanPostProcessor，一般我们俗称对象后处理器

简单来说，通过BeanPostProcessor可以对我们的对象进行“加工处理”。

## \*Bean的生命周期

**Spring管理Bean(或者说Bean的生命周期)步骤：**

ResouceLoader加载配置信息

BeanDefintionReader解析配置信息，生成一个一个的BeanDefintion

BeanDefintion由BeanDefintionRegistry管理起来

BeanFactoryPostProcessor对配置信息进行加工(也就是处理配置的信息，一般通过PropertyPlaceholderConfigurer来实现)

实例化Bean

如果该Bean配置/实现了InstantiationAwareBean，则调用对应的方法

使用BeanWarpper来完成对象之间的属性配置(依赖)

如果该Bean配置/实现了Aware接口，则调用对应的方法

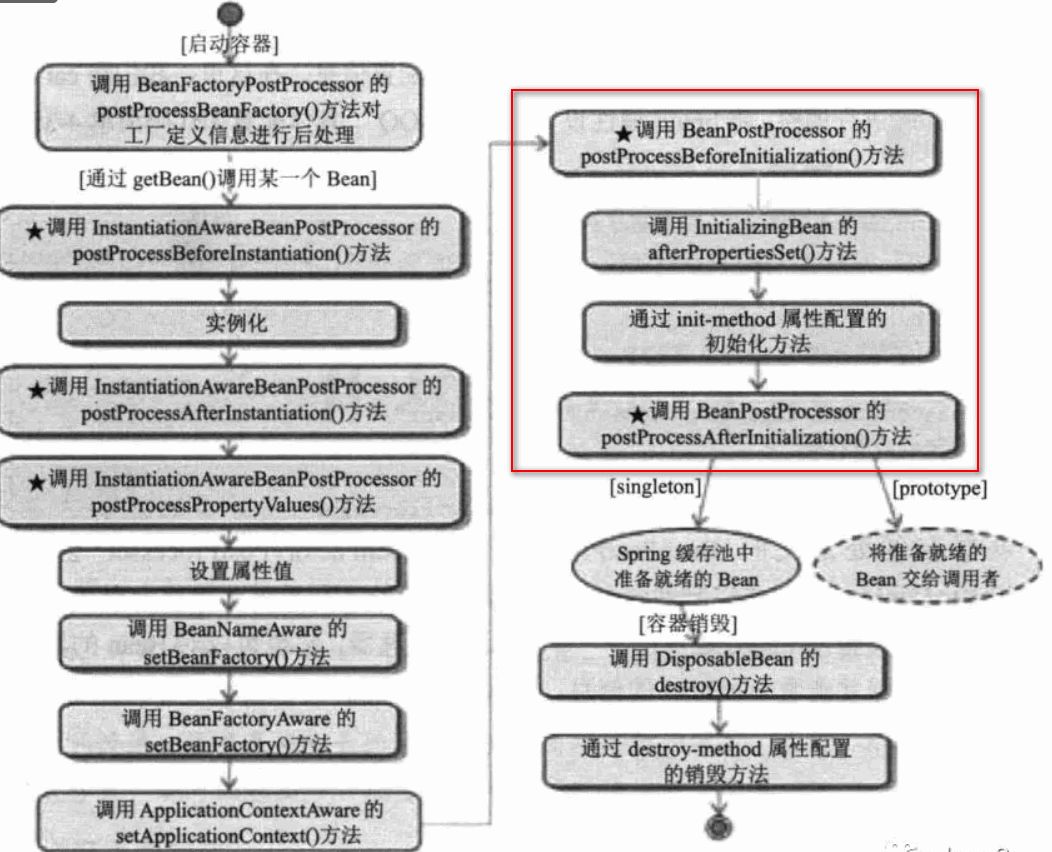
如果该Bean配置了BeanPostProcessor的before方法，则调用

如果该Bean配置了init-method或者实现InstantiationBean，则调用对应的方法

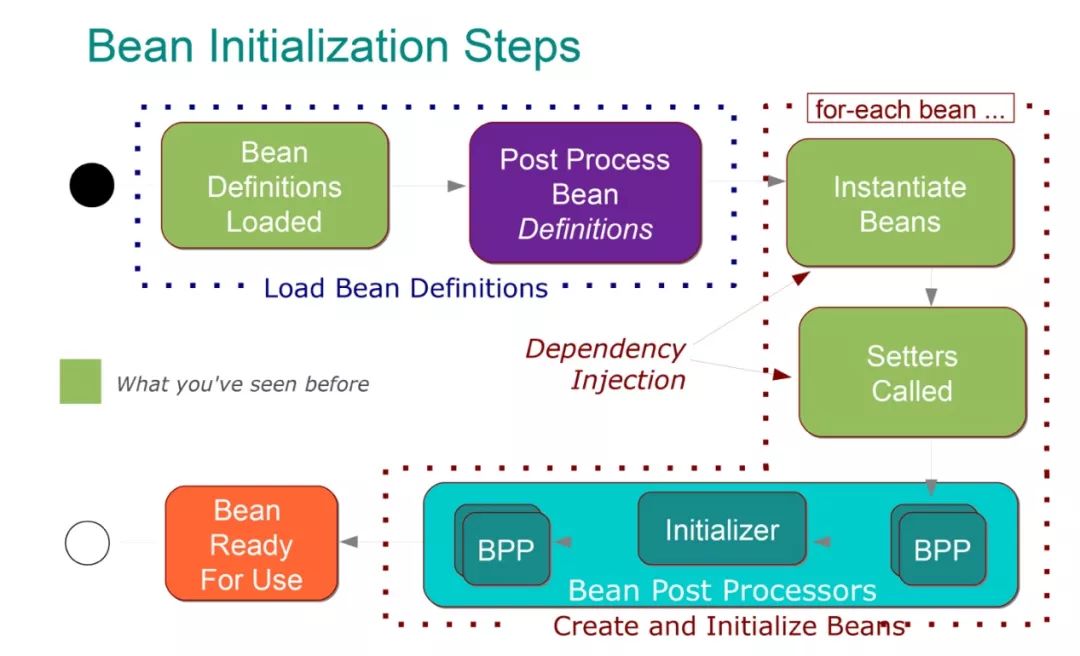
如果该Bean配置了BeanPostProcessor的after方法，则调用

将对象放入到HashMap中

最后如果配置了destroy或者DisposableBean的方法，则执行销毁操作



其中也有关于BPP图片：



## 为什么特意讲BPP？

Spring AOP编程底层通过的是动态代理技术，在调用的时候肯定用的是代理对象。那么Spring是怎么做的呢？

我只需要写一个BPP，在postProcessBeforeInitialization或者postProcessAfterInitialization方法中，对对象进行判断，看他需不需要织入切面逻辑，如果需要，那我就根据这个对象，生成一个代理对象，然后返回这个代理对象，那么最终注入容器的，自然就是代理对象了。

Spring提供了BeanPostProcessor，就是让我们可以对有需要的对象进行“加工处理”啊！

## 认识Spring事务几个重要的接口

在编程式事务中有以下几个重要的了接口：

TransactionDefinition：定义了Spring兼容的事务属性(比如事务隔离级别、事务传播、事务超时、是否只读状态)

TransactionStatus：代表了事务的具体运行状态(获取事务运行状态的信息，也可以通过该接口间接回滚事务等操作)

PlatformTransactionManager：事务管理器接口(定义了一组行为，具体实现交由不同的持久化框架来完成---类比JDBC)

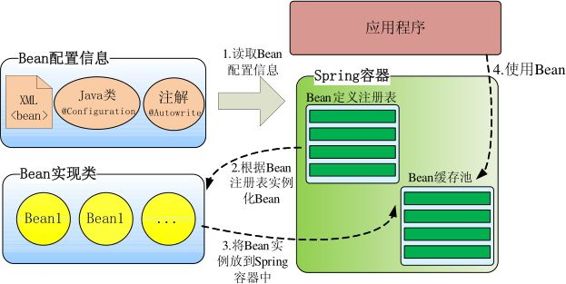
在声明式事务中，除了TransactionStatus和PlatformTransactionManager接口，还有几个重要的接口：

TransactionProxyFactoryBean：生成代理对象

TransactionInterceptor：实现对象的拦截

TransactionAttrubute：事务配置的数据

## Spring容器初始化过程



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容器确实堪称一部设计精密的机器，其内部拥有众多的组件和装置。Spring的高明之处在于，它使用众多接口描绘出了所有装置的蓝图，构建好Spring的骨架，继而通过继承体系层层推演，不断丰富，最终让Spring成为有血有肉的完整的框架。所以查看Spring框架的源码时，有两条清晰可见的脉络：

1）接口层描述了容器的重要组件及组件间的协作关系；

2）继承体系逐步实现组件的各项功能。

接口层清晰地勾勒出Spring框架的高层功能，框架脉络呼之欲出。有了接口层抽象的描述后，不但Spring自己可以提供具体的实现，任何第三方组织也可以提供不同实现， 可以说Spring完善的接口层使框架的扩展性得到了很好的保证。纵向继承体系的逐步扩展，分步骤地实现框架的功能，这种实现方案保证了框架功能不会堆积在某些类的身上，造成过重的代码逻辑负载，框架的复杂度被完美地分解开了。

Spring组件按其所承担的角色可以划分为两类：

1）物料组件：Resource、BeanDefinition、PropertyEditor以及最终的Bean等，它们是加工流程中被加工、被消费的组件，就像流水线上被加工的物料；

2）加工设备组件：ResourceLoader、BeanDefinitionReader、BeanFactoryPostProcessor、InstantiationStrategy以及BeanWrapper等组件像是流水线上不同环节的加工设备，对物料组件进行加工处理。

## spring有两种代理方式

**spring在运行期创建代理，不需要特殊的编译器。spring有两种代理方式**：

a.若目标对象实现了若干接口，spring就会使用JDK动态代理。JDK动态代理主要涉及到java.lang.reflect包中的两个类：Proxy和InvocationHandler

b.若目标对象没有实现任何接口，spring就使用CGLIB库生成目标对象的子类。CGLib全称为Code Generation Library，是一个强大的高性能，高质量的代码生成类库，可以在运行期扩展Java类与实现Java接口，CGLib封装了asm，可以再运行期动态生成新的class。

和JDK动态代理相比较：JDK创建代理有一个限制，就是只能为接口创建代理实例，而对于没有通过接口定义业务方法的类，则可以通过CGLib创建动态代理。

CGLib 创建代理的速度比较慢，但创建代理后运行的速度却非常快，而 JDK 动态代理正好相反。如果在运行的时候不断地用 CGLib 去创建代理，系统的性能会大打折扣。

**Spring是使用AspectJ工具实现的AOP**。AspectJ是一个面向切面的框架，他定义了AOP语法，所以它有一个专门的编译器用来生成遵守Java字节编码规范的Class文件

**Spring把事务分为两种一种是：编程式事务，一种是声明式事务。**

编程式事务使用Spring 提供的事务模版类：org.springframework.transaction.support.TransactionTemplate

声明式事务是非侵入式的事务。声明式事务只需在配置文件中配置，而不需要去操作逻辑代码。

**spring事务的7大传播属性**

1.PROPAGATION\_MANDATORY :要求调用该方法的线程必须处于事务环境中，否则抛出异常。propagation\_mandatory

2.PROPAGATION\_NESTED :如果执行该方法的线程已处于事务环境下，依然启动新的事务，方法在嵌套的事务里执行。nested如果执行该方法的线程并未处于事务中，也启动新的事务，然后执行该方法，此时与PROPAGATION\_REQUIRED相同。

3.PROPAGATION\_NEVER :不允许调用该方法的线程处于事务环境下，如果调用该方法的线程处于事务环境下，则抛出异常。

4.PREOPAGATION\_NOT\_SUPPORTED :如果调用该方法的线程处于在事务中，则先暂停当前事务，然后执行该方法。

5.PROPAGATION\_REQUIRED ：要求在事务环境中执行该方法，如果当前执行线程已处于事务中，则直接调用，如果当前执行线程不已处于事务中，则启动新的事务后执行该方法。

6.PROPAGATION\_REQUIRES\_NEW :该方法要求有一个在新的事务环境中执行，如果当前执行线程已处于事务中，先暂停当前事务,启动新的事务后执行该方法；如果当前执行线程不已处于事务中，则启动新的事务后执行该方法。

7.PROPAGATION\_SUPPORTS :如果当前执行线程已处于事务中，则使用当前事务，否则不使用事务。

**spring是如何管理实务的，事务管理机制**

Spring并不直接管理事务，而是提供了多种事务管理器，他们将事务管理的职责委托给Hibernate或者JTA等持久化机制所提供的相关平台框架的事务来实现。

## ?Spring事务失效情况

Spring事务失效的5种大的原因:

1. 如使用mysql且引擎是MyISAM，则事务会不起作用，原因是MyISAM不支持事务，可以改成InnoDB

2. 如果使用了spring+mvc，则context:component-scan重复扫描问题可能会引起事务失败。

3. @Transactional 注解开启配置，必须放到listener里加载，如果放到DispatcherServlet的配置里，事务也是不起作用的。

4. @Transactional 注解只能应用到 public 可见度的方法上。 如果你在 protected、private 或者 package-visible 的方法上使用 @Transactional 注解，它也不会报错，事务也会失效。

5. Spring团队建议在具体的类（或类的方法）上使用 @Transactional 注解，而不要使用在类所要实现的任何接口上。在接口上使用 @Transactional 注解，只能当你设置了基于接口的代理时它才生效。因为注解是 不能继承的，这就意味着如果正在使用基于类的代理时，那么事务的设置将不能被基于类的代理所识别，而且对象也将不会被事务代理所包装。

需要注意的事项:

在业务代码中，有如下两种情况，比如：

throw new RuntimeException(“xxxxxxxxxxxx”); 事务回滚

throw new Exception(“xxxxxxxxxxxx”); 事务没有回滚

spring内部catch的就是 RuntimeException, service抛出RuntimeException可以回滚。如果抛出Exception，就不回滚….

1).Spring的AOP即声明式事务管理默认是针对unchecked exception回滚。也就是默认对RuntimeException()异常或是其子类进行事务回滚；checked异常,即Exception可try{}捕获的不会回滚，如果使用try-catch捕获抛出的unchecked异常后没有在catch块中采用页面硬编码的方式使用spring api对事务做显式的回滚，则事务不会回滚， “将异常捕获,并且在catch块中不对事务做显式提交=生吞掉异常” ，要想捕获非运行时异常则需要如下配置：

解决办法： 1.在针对事务的类中抛出RuntimeException异常，而不是抛出Exception。 2.在txAdive中增加rollback-for，里面写自己的exception，例如自己写的exception,或者定义不会滚的异常

2).spring的事务边界是在调用业务方法之前开始的，业务方法执行完毕之后来执行commit or rollback(Spring默认取决于是否抛出runtime异常). 如果抛出runtime exception 并在你的业务方法中没有catch到的话，事务会回滚。 一般不需要在业务方法中catch异常，如果非要catch，在做完你想做的工作后（比如关闭文件等）一定要抛出runtime exception，否则spring会将你的操作commit,这样就会产生脏数据.所以你的catch代码是画蛇添足。

3).基于注解的事务：Transactional的异常控制，默认是Check Exception 不回滚，unCheck Exception回滚。如果配置了rollbackFor 和 noRollbackFor 且两个都是用同样的异常，那么遇到该异常，还是回滚。rollbackFor 和noRollbackFor 配置也许不会含盖所有异常，对于遗漏的按照Check Exception 不回滚，unCheck Exception回滚

**Spring中 @Transactional 注解的限制**

1. 同一个类中, 一个nan-transactional的方法去调用transactional的方法, 事务会失效

2. 在private方法上标注transactional, 事务无效

结论

@Transactional 加于private方法, 无效

@Transactional 加于未加入接口的public方法, 再通过普通接口方法调用, 无效

@Transactional 加于接口方法, 无论下面调用的是private或public方法, 都有效

@Transactional 加于接口方法后, 被本类普通接口方法直接调用, 无效

@Transactional 加于接口方法后, 被本类普通接口方法通过接口调用, 有效

@Transactional 加于接口方法后, 被它类的接口方法调用, 有效

@Transactional 加于接口方法后, 被它类的私有方法调用后, 有效

总结: Transactional是否生效, 仅取决于是否加载于接口方法, 并且是否通过接口方法调用(而不是本类调用)

**spring父子上下文**

父上下文：使用listener监听器来加载配置文件<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>Spring 会创建一个WebApplicationContext上下文，称为父上下文（父容器），保存在 ServletContext中.

子上下文：使用Spring MVC 来处理拦截相关的请求时，会配置DispatchServlet，每个DispatchServlet会有一个自己的上下文，称为子上下文，它也保存在 ServletContext，父子上下文的key不一样。

父上下文（父容器）和子上下文（子容器）的访问权限：

子上下文可以访问父上下文中的bean，但是父上下文不可以访问子上下文中的bean。

## SpringMVC工作流程描述

1. 用户向服务器发送请求，请求被Spring 前端控制Servelt DispatcherServlet捕获；

2. DispatcherServlet对请求URL进行解析，得到请求资源标识符（URI）。然后根据该URI， 调用HandlerMapping获得该Handler配置的所有相关的对象（包括Handler对象以及Handler对象对应的拦截器），最后以 HandlerExecutionChain对象的形式返回；

3. DispatcherServlet 根据获得的Handler，选择一个合适的HandlerAdapter。（附注：如果成功获得HandlerAdapter后，此时将开始执行拦截器的preHandler(...)方法）

4. 提取Request中的模型数据，填充Handler入参，开始执行Handler（Controller)。 在填充Handler的入参过程中，根据你的配置，

Spring将帮你做一些额外的工作：

HttpMessageConveter： 将请求消息（如Json、xml等数据）转换成一个对象，将对象转换为指定的响应信息

数据转换：对请求消息进行数据转换。如String转换成Integer、Double等

数据根式化：对请求消息进行数据格式化。 如将字符串转换成格式化数字或格式化日期等

数据验证： 验证数据的有效性（长度、格式等），验证结果存储到BindingResult或Error中

5. Handler执行完成后，向DispatcherServlet 返回一个ModelAndView对象；

6. 根据返回的ModelAndView，选择一个适合的ViewResolver（必须是已经注册到Spring容器中的 ViewResolver)返回给DispatcherServlet ；

7. ViewResolver 结合Model和View，来渲染视图

8. 将渲染结果返回给客户端。

SpringMVC工作流程的简洁描述

DispatcherServlet根据请求URI，调用HandlerMapping获得该Handler，然后DispatcherServlet会根据获得的Handler，选择一个合适的HandlerAdapter。提取Request中的模型数据，填充Handler入参，开始执行Handler（Controller)。Handler执行完成后，向DispatcherServlet 返回一个ModelAndView对象。根据返回的ModelAndView，选择一个适合的ViewResolver。ViewResolver 结合Model和View，来渲染视图

**beanfactory和facatoryBean**

实现 BeanFactory 接口的类表明此类事一个工厂，作用就是配置、新建、管理 各种Bean而 实现 FactoryBean 的类表明此类也是一个Bean，类型为工厂Bean（Spring中共有两种bean，一种为普通bean，另一种则为工厂bean）。顾名思义，它也是用来管理Bean的，而它本身由spring管理。

有一点需要注意，如果将一个实现了FactoryBean的类成功配置到了spring上下文中，那么通过该类对象的名称（比如appleFactoryBean）从spring的applicationContext或者beanFactory获取bean时，获取到的是appleFactoryBean创建的apple实例，而不是appleFactoryBean自己，如果想通过spring拿到appleFactoryBean，需要在名称前加 & 符号

如何实现spring aop

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；而动态代理则以Spring AOP为代表。

静态代理：

举个实例的例子来说。首先我们有一个普通的Hello类

public class Hello {

public void sayHello() {

System.out.println("hello");

}

public static void main(String[] args) {

Hello h = new Hello();

h.sayHello();

}

}

使用AspectJ编写一个Aspect

public aspect TxAspect {

void around():call(void Hello.sayHello()){

System.out.println("开始事务 ...");

proceed();

System.out.println("事务结束 ...");

}

}

查看一下编译后的Hello.class

public class Hello {

public Hello() {

}

public void sayHello() {

System.out.println("hello");

}

public static void main(String[] args) {

Hello h = new Hello();

sayHello\_aroundBody1$advice(h, TxAspect.aspectOf(), (AroundClosure)null);

}

}

可以看到，这个类比原来的Hello.java多了一些代码，这就是AspectJ的静态代理，它会在编译阶段将Aspect织入Java字节码中， 运行的时候就是经过增强之后的AOP对象。

public void ajc$around$com\_listenzhangbin\_aop\_TxAspect$1$f54fe983(AroundClosure ajc$aroundClosure) {

System.out.println("开始事务 ...");

ajc$around$com\_listenzhangbin\_aop\_TxAspect$1$f54fe983proceed(ajc$aroundClosure);

System.out.println("事务结束 ...");

}

**动态代理**：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。JDK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。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某个类的子类，注意，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。

# Spring中的设计者模式

**Spring 框架中用到了哪些设计模式：**

**工厂设计模式** : Spring使用工厂模式通过 BeanFactory、ApplicationContext 创建 bean 对象。

**代理设计模式** : Spring AOP 功能的实现。

**单例设计模式** : Spring 中的 Bean 默认都是单例的。

**模板方法模式** : Spring 中 jdbcTemplate、hibernateTemplate 等以 Template 结尾的对数据库操作的类，它们就使用到了模板模式。

**包装器设计模式** : 我们的项目需要连接多个数据库，而且不同的客户在每次访问中根据需要会去访问不同的数据库。这种模式让我们可以根据客户的需求能够动态切换不同的数据源。

**观察者模式**: Spring 事件驱动模型就是观察者模式很经典的一个应用。

**适配器模式** :Spring AOP 的增强或通知(Advice)使用到了适配器模式、spring MVC 中也是用到了适配器模式适配Controller。

## 工厂模式

Spring使用工厂模式可以通过 BeanFactory 或 ApplicationContext 创建 bean 对象。

两者对比：

BeanFactory ：延迟注入(使用到某个 bean 的时候才会注入),相比于BeanFactory来说会占用更少的内存，程序启动速度更快。

ApplicationContext ：容器启动的时候，不管你用没用到，一次性创建所有 bean 。BeanFactory 仅提供了最基本的依赖注入支持，ApplicationContext 扩展了 BeanFactory ,除了有BeanFactory的功能还有额外更多功能，所以一般开发人员使用ApplicationContext会更多。

## 单例设计模式

Spring 中 bean 的默认作用域就是 singleton(单例)的。

## 代理设计模式(AOP)

分JDK Proxy代理与CGLIB代理

AspectJ 应该算的上是 Java 生态系统中最完整的 AOP 框架了。

Spring AOP 和 AspectJ AOP 有什么区别?

Spring AOP 属于运行时增强，而 AspectJ 是编译时增强。 Spring AOP 基于代理(Proxying)，而 AspectJ 基于字节码操作(Bytecode Manipulation)

模板方法

Spring 中 jdbcTemplate、hibernateTemplate 等以 Template 结尾的对数据库操作的类，它们就使用到了模板模式。一般情况下，我们都是使用继承的方式来实现模板模式，但是 Spring 并没有使用这种方式，而是使用Callback 模式与模板方法模式配合，既达到了代码复用的效果，同时增加了灵活性。

public abstract class Template {

//这是我们的模板方法

public final void TemplateMethod(){

PrimitiveOperation1();

PrimitiveOperation2();

}

protected void PrimitiveOperation1(){

//当前类实现

}

//被子类实现的方法

protected abstract void PrimitiveOperation2();

}

public class TemplateImpl extends Template {

@Override

public void PrimitiveOperation2() {

//当前类实现

}

}

## 观察者模式

Spring 的事件流程总结:

定义一个事件: 实现一个继承自 ApplicationEvent，并且写相应的构造函数；

定义一个事件监听者：实现 ApplicationListener 接口，重写 onApplicationEvent() 方法；

使用事件发布者发布消息: 可以通过 ApplicationEventPublisher 的 publishEvent() 方法发布消息。

## 适配器模式

适配器模式(Adapter Pattern) 将一个接口转换成客户希望的另一个接口，适配器模式使接口不兼容的那些类可以一起工作，其别名为包装器(Wrapper)。

spring AOP中的适配器模式:我们知道 Spring AOP 的实现是基于代理模式，但是 Spring AOP 的增强或通知(Advice)使用到了适配器模式，与之相关的接口是AdvisorAdapter 。Advice 常用的类型有：BeforeAdvice（目标方法调用前,前置通知）、AfterAdvice（目标方法调用后,后置通知）、AfterReturningAdvice(目标方法执行结束后，return之前)等等。每个类型Advice（通知）都有对应的拦截器:MethodBeforeAdviceInterceptor、AfterReturningAdviceAdapter、AfterReturningAdviceInterceptor。Spring预定义的通知要通过对应的适配器，适配成 MethodInterceptor接口(方法拦截器)类型的对象（如：MethodBeforeAdviceInterceptor 负责适配 MethodBeforeAdvice）。

spring MVC中的适配器模式:在Spring MVC中，DispatcherServlet 根据请求信息调用 HandlerMapping，解析请求对应的 Handler。解析到对应的 Handler（也就是我们平常说的 Controller 控制器）后，开始由HandlerAdapter 适配器处理。HandlerAdapter 作为期望接口，具体的适配器实现类用于对目标类进行适配，Controller 作为需要适配的类。

## 装饰者模式

Spring 中配置 DataSource 的时候，DataSource 可能是不同的数据库和数据源。我们能否根据客户的需求在少修改原有类的代码下动态切换不同的数据源？这个时候就要用到装饰者模式(这一点我自己还没太理解具体原理)。Spring 中用到的包装器模式在类名上含有 Wrapper或者 Decorator。这些类基本上都是动态地给一个对象添加一些额外的职责