# 1、说说你对Spring的理解？

1、Spring是一个开源框架，主要是为简化企业级应用开发而生。可以实现EJB可以实现的功能，Spring是一个IOC和AOP容器框架。

① 控制反转(IOC)：Spring容器使用了工厂模式为我们创建了所需要的对象，我们使用时不需要自己去创建，直接调用Spring为我们提供的对象即可，这就是控制反转的思想。

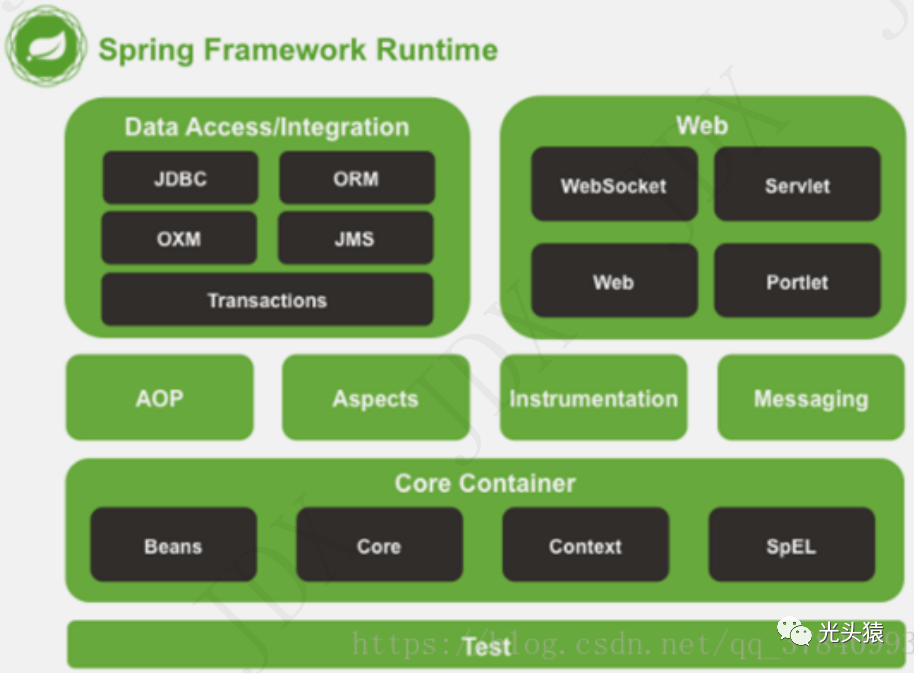
② 依赖注入(DI)：Spring使用Java Bean对象的Set方法或者带参数的构造方法为我们在创建所需对象时将其属性自动设置所需要的值的过程就是依赖注入的基本思想。

③ 面向切面编程(AOP)：在面向对象编程(OOP)思想中，我们将事物纵向抽象成一个个的对象。而在面向切面编程中，我们将一个个对象某些类似的方面横向抽象成一个切面，对这个切面进行一些如权限验证，事物管理，记录日志等公用操作处理的过程就是面向切面编程的思想。

2、在Spring中，所有管理的都是JavaBean对象，而BeanFactory和ApplicationContext就是Spring框架的那个IOC容器，现在一般使用ApplicationContext，其不但包括了BeanFactory的作用，同时还进行了更多的扩展。

# 2、Spring由哪些模块组成？

截止到目前Spring框架已集成了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) 。



Spring常见的模块说明:

1、Spring Core(核心容器): 核心容器提供Spring框架的基本功能。Spring以bean的方式组织和管理Java应用中的各个组件及其关系。Spring使用BeanFactory来产生和管理Bean，它是工厂模式的实现。BeanFactory使用控制反转(IoC)模式将应用的配置和依赖性规范与实际的应用程序代码分开

2、Spring Context(应用上下文): 应用上下文: 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，如JNDI、EJB、电子邮件、国际化、校验和调度功能

3、SpringAOP(面向切面编程)：是面向对象编程的有效补充和完善，Spring的AOP是基于动态代理实现的

4、SpringDao(JDBC和Dao模块)：JDBC、DAO的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所抛出的错误信息。异常层次结构简化了错误处理，并且极大的降低了需要编写的代码数量，比如打开和关闭链接

5、Spring ORM(对象实体映射)：Spring框架插入了若干个ORM框架，从而提供了ORM对象的关系工具，其中包括了Hibernate、JDO和 IBatis SQL Map等，所有这些都遵从Spring的通用事物和DAO异常层次结构。

6、Spring Web(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Struts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

7、Spring Web MVC(MVC模块)：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、POI等，模型由JavaBean构成，存放于m当中，而视图是一个接口，负责实现模型，控制器表示逻辑代码，由c的事情。Spring框架的功能可以用在任何J2EE服务器当中，大多数功能也适用于不受管理的环境。Spring的核心要点就是支持不绑定到特定J2EE服务的可重用业务和数据的访问的对象，毫无疑问这样的对象可以在不同的J2EE环境，独立应用程序和测试环境之间重用。

# 3、在Spring中有几种配置Bean的方式？

配置方式：

1、基于XML的配置

2、基于注解的配置

3、基于Java的配置

# 4、BeanFactory和ApplicationContext有什么区别？

1、BeanFactory：BeanFactory在启动的时候不会去实例化Bean，当从容器中拿Bean的时候才会去实例化；

2、ApplicationContext：ApplicationContext在启动的时候就把所有的Bean全部实例化了。它还可以为Bean配置lazy-init=true来让Bean延迟实例化；

# 5、Spring框架中的单例bean是线程安全的吗？

肯定不是线程安全的，当多用户同时请求一个服务时，容器会给每一个请求分配一个线程，这是多个线程会并发执行该请求多对应的业务逻辑（成员方法），此时就要注意了，如果该处理逻辑中有对该单列状态的修改（体现为该单例的成员属性），则必须考虑线程同步问题。Spring框架并没有对单例bean进行任何多线程的封装处理。关于单例bean的线程安全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Service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中状态的bean的作用域由singleton变更为prototype。+--

# 6、Spring Bean有哪些作用域，它们之间有什么区别？

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 。

# 7、Spring框架中有哪些不同类型的事件？

Spring 提供了以下5种标准的事件：

1、上下文更新事件（ContextRefreshedEvent）：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。

# 8、请解释一下Spring框架有哪些自动装配模式，它们之间有何区别？

spring的自动装配功能的定义：无须在Spring配置文件中描述javaBean之间的依赖关系（如配置<property>、<constructor-arg>）。

自动装配模式：

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 ：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 。

比如如下注入：

可以改造为：

# 9、你用过哪些重要的Spring 注解？

1、@Controller - 用于 Spring MVC 项目中的控制器类。

2、@Service - 用于服务类。

3、@RequestMapping - 用于在控制器处理程序方法中配置 URI 映射。

4、@ResponseBody - 用于发送 Object 作为响应，通常用于发送 XML 或 JSON 数据作为响应。

5、@PathVariable - 用于将动态值从 URI 映射到处理程序方法参数。

6、@Autowired - 用于在 spring bean 中自动装配依赖项。通过类型来实现自动注入bean。和@Qualifier注解配合使用可以实现根据name注入bean。

7、@Qualifier - 和@Autowired一块使用，在同一类型的bean有多个的情况下可以实现根据name注入的需求。

8、@Scope - 用于配置 spring bean 的范围。

9、@Configuration，@ComponentScan 和 @Bean - 用于基于 java 的配置。

10、@Aspect，@Before，@After，@Around，@Pointcut - 用于切面编程（AOP）

# 10、解释一下Spring AOP里面的几个名词？

名词解释：

1、连接点（Join point）：指程序运行过程中所执行的方法。在Spring AOP中，一个连接点总代表一个方法的执行。

2、切入点（Pointcut）：被拦截用于做增强的那些连接点。

3、通知（Advice）：指要在切入点（Join Point）上执行的动作，即增强的逻辑，比如权限校验和、日志记录等。

4、切面（Aspect）：被抽取出来的公共模块，可以用来横切多个对象。Aspect切面可以看成 Pointcut切点和 Advice通知的结合，一个切面可以由多个切点和通知组成。

5、目标对象（Target）：包含连接点的对象，也称作被通知（Advice）的对象。由于Spring AOP是通过动态代理实现的，所以这个对象永远是一个代理对象。

6、织入（Weaving）：通过动态代理，在目标对象（Target）的方法（即切入点）中执行增强逻辑（Advice）的过程。

7、引入（Introduction）：添加额外的方法或者字段到被通知的类。Spring允许引入新的接口（以及对应的实现）到任何被代理的对象。

如下图所示：

# 11、Spring中AOP的底层是怎么实现的？

Spring中AOP底层的实现是基于**动态代理**进行实现的。

常见的动态代理技术有两种：JDK的动态代理和CGLIB。两者的区别如下所示：

1、JDK动态代理只能对实现了接口的类生成代理，而不能针对类

2、Cglib是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法进行增强，但是因为采用的是继承，所以该类或方法最好不要声明为final，对于final类或方法，是无法继承的。

Spring如何选择是用JDK还是cglib？

1、当bean实现接口时，会用JDK代理模式

2、当bean没有实现接口，会用cglib实现

3、可以强制使用cglib，如下所示

# 12、Spring事务的实现方式以及原理？

Spring支持编程式事务管理和声明式事务管理两种方式！

编程式事务控制：需要使用TransactionTemplate来进行实现，这种方式实现对业务代码有侵入性，因此在项目中很少被使用到。

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

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

# 13、什么是事务的传播行为？在Spring框架中都有哪些事务的传播行为？

Spring的事务传播行为：指的就是当一个事务方法被另一个事务方法调用时，这个事务方法对事务的态度。举例：methodA事务方法调用methodB事务方法时，methodB是继续在调用者methodA的事务中运行呢，还是为自己开启一个新事务运行，这就是由methodB的事务传播行为决定的。

在Spring中提供了7种事务的传播行为：

1、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、REQUIRES\_NEW：创建新事务，无论当前存不存在事务，都创建新事务。

3、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。

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、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。

6、NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

7、NESTED：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则按REQUIRED属性执行。

# 14、Spring如何管理事务的？

Spring事务管理主要包括3个接口，Spring事务主要由以下三个共同完成的：

1、PlatformTransactionManager：事务管理器，主要用于平台相关事务的管理。

主要包括三个方法：

① commit：事务提交。

② rollback：事务回滚。

③ getTransaction：获取事务状态。

2、TransacitonDefinition：事务定义信息，用来定义事务相关属性给事务管理器PlatformTransactionManager使用

主要包含的方法：

① getIsolationLevel：获取隔离级别。

② getPropagationBehavior：获取传播行为。

③ getTimeout获取超时时间。

④ isReadOnly：是否只读（保存、更新、删除时属性变为false--可读写，查询时为true--只读）

3、TransationStatus：事务具体运行状态，事务管理过程中，每个时间点事务的状态信息。

主要包含的方法：

① hasSavepoint()：返回这个事务内部是否包含一个保存点。

② isCompleted()：返回该事务是否已完成，也就是说，是否已经提交或回滚。

③ isNewTransaction()：判断当前事务是否是一个新事务。

# 15、Spring事务什么情况下会失效？

事务失效的常见场景：

1、数据库引擎不支持事务：这里以 MySQL为例，其MyISAM引擎是不支持事务操作的，InnoDB才是支持事务的引擎，一般要支持事务都会使用 InnoDB。

2、bean没有被Spring 管理

*// @Service*public class OrderServiceImpl implements OrderService {    @Transactional public void updateOrder(Order order) { *// update order* }}

如果此时把 @Service注解注释掉，这个类就不会被加载成一个 Bean，那这个类就不会被Spring 管理了，事务自然就失效了。

3、方法不是public的：@Transactional只能用于public的方法上，否则事务不会失效。

4、自身调用问题

public void save() { this.show();}  
@Transactionalpublic void show() {  
 Account account = new Account() ; account.setName("itcast"); account.setMoney(300d); accountDao.save(account);  
 *// 抛出异常* int i = 1/0; *// 事务是否回滚？*}

5、数据源没有配置事务管理器

6、异常在方法内部通过try...catch处理掉了

public void save() { this.show();}  
@Transactionalpublic void show() {    Account account = new Account() ; account.setName("itcast"); account.setMoney(300d); accountDao.save(account);    try {        *// 抛出异常* int i = 1/0; *// 事务是否回滚？*    }catch (Exception e) { e.printStackTrace(); }}

7、异常类型错误：

事务默认回滚的

是：RuntimeException

public void save() throws Exception { this.show();}  
@Transactionalpublic void show() throws Exception{    Account account = new Account() ; account.setName("itcast"); account.setMoney(300d); accountDao.save(account);    try {        *// 抛出异常* int i = 1/0; *// 事务是否回滚？*    }catch (Exception e) { throw new Exception("保存订单数据失败") ; }}

这样事务也是不生效的，因为默认回滚的是：RuntimeException，如果你想触发其他异常的回滚，需要在注解上配置一下，如：

@Transactional(rollbackFor = Exception.class)

# 16、请解释一下Spring Bean的生命周期？

Spring Bean的生命周期如下图所示：

1、实例化Bean：

对于BeanFactory容器，当客户向容器请求一个尚未初始化的bean时，或初始化bean的时候需要注入另一个尚未初始化的依赖时，容器就会调用createBean进行实例化。对于ApplicationContext容器，当容器启动结束后，通过获取BeanDefinition对象中的信息，实例化所有的bean。(BeanDefinition是Spring 中极其重要的一个概念，它存储了bean对象的所有特征信息，如是否单例，是否懒加载，factoryBeanName 等。BeanDefinition对象的创建时通过各种bean的配置信息进行构建 )

2、设置对象属性（依赖注入）:

实例化后的对象被封装在BeanWrapper对象中，紧接着Spring根据BeanDefinition中的信息以及通过BeanWrapper提供的设置属性的接口完成依赖注入。

3、处理Aware接口：

Spring会检测该对象是否实现了xxxAware接口，并将相关的xxxAware实例注入给Bean：如果这个Bean已经实现了BeanNameAware接口，会调用它实现的setBeanName(String beanId)方法，此处传递的就是Spring配置文件中Bean的id值。如果这个Bean已经实现了BeanFactoryAware接口，会调用它实现的setBeanFactory()方法，传递的是Spring工厂自身。

如果这个Bean已经实现了ApplicationContextAware接会调用setApplicationContext(ApplicationContext)方法，传入Spring上下文。

4、Bean的后置处理器（BeanPostProcessor）：

如果想对Bean进行一些自定义的处理，那么可以让Bean实现了BeanPostProcessor接口，那将会调用postProcessBeforeInitialization方法。由于这个方法是在Bean初始化结束时调用的，所以可以被应用于内存或缓存技术；

5、InitializingBean 与 init-method：

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

6、Bean的后置处理器（BeanPostProcessor）：

如果这个Bean实现了BeanPostProcessor接口，将会调用postProcessAfterInitialization(Object obj, String s)方法；以上几个步骤完成后，Bean就已经被正确创建了，之后就可以使用这个Bean了。

7、DisposableBean：

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，会调用其实现的destroy()方法；

8、destroy-method：

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。

示例代码如下所示：

User类：

@Component@Slf4jpublic class User implements BeanNameAware , BeanFactoryAware , ApplicationContextAware {    public User() { System.out.println("a的构造方法执行了........."); }     private String name ;        @Value("张三") public void setName(String name) { System.out.println("setName方法执行了........."); }     @Override public void setBeanName(String name) { System.out.println("setBeanName方法执行了........."); }     @Override public void setBeanFactory(BeanFactory beanFactory) throws BeansException { System.out.println("setBeanFactory方法执行了.........");    }        @PostConstruct public void init() { System.out.println("init方法执行了................."); }     @PreDestroy public void destory() { System.out.println("destory方法执行了..............."); }  
 @Override public void setApplicationContext(ApplicationContext applicationContext) throws BeansException { System.out.println("setApplicationContext方法执行了........"); }}

Bean的后置处理器(BeanPostProcessor)：

@Componentpublic class MyBeanPostProcessor implements BeanPostProcessor {  
 @Override public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException { System.out.println("postProcessBeforeInitialization方法执行了........."); return bean; }  
    @Override public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException { System.out.println("postProcessAfterInitialization方法执行了........."); return bean; }  
}

测试类：

ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext("applicationContext.xml");User user = applicationContext.getBean(User.class);System.out.println(user);applicationContext.close();

控制台输出结果：

Generic bean: class [com.itheima.user.domain.User]; scope=singleton; abstract=false; lazyInit=false; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null; defined in file [E:\idea-edu-workplace\spring\target\classes\com\itheima\user\domain\User.class]a的构造方法执行了.........setName方法执行了.........setBeanName方法执行了.........setBeanFactory方法执行了.........setApplicationContext方法执行了........postProcessBeforeInitialization方法执行了.........init方法执行了.................postProcessAfterInitialization方法执行了.........com.itheima.user.domain.User@4386f16destory方法执行了...............

InitializingBean接口为bean提供了初始化方法的方式，它只包括afterPropertiesSet方法，凡是继承该接口的类，在初始化bean的时候都会执行该方法。

# 17、Spring中的AOP是在bean生命周期的哪一步实现的？

Spring的AOP是通过

AspectJAwareAdvisorAutoProxyCreator来实现的， 该类的类图如下：

# 18、什么是Spring的循环依赖？

简单的来说就是A依赖B的同时，B依赖A。在创建A对象的同时需要使用的B对象，在创建B对象的同时需要使用到A对象。如下代码所示：

@Componentpublic class A { public A(){ System.out.println("A的构造方法执行了..."); }    private B b;    @Autowired public void setB(B b) { this.b = b; System.out.println("给A注入B"); }}@Componentpublic class B {    public B(){ System.out.println("B的构造方法执行了..."); }    private A a;    @Autowired public void setA(A a) { this.a = a; System.out.println("给B注入了A"); }}

# 19、出现循环依赖以后会有什么问题？

对象的创建过程会产生死循环，如下所示：

在spring中通过某些机制(三级缓存)帮开发者解决了部分循环依赖的问题。

# 20、Spring如何解决循环依赖的？

生命周期回顾：

1、Spring扫描class得到BeanDefinition

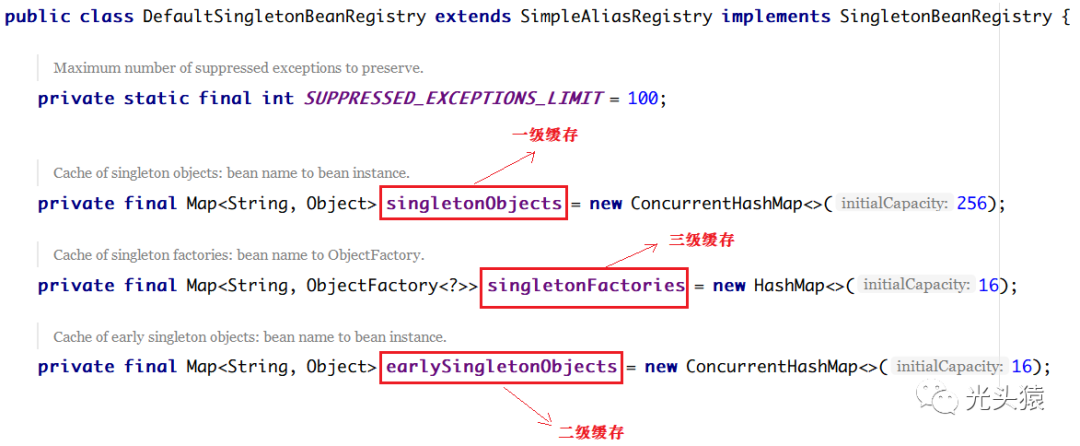
2、根据得到的BeanDefinition，根据class推断构造方法， 通过反射得到一个对象（原始对象）

3、为原始对象填充属性（依赖注入）

4、如果原始对象中的某一个方法配置的有AOP，则需要针对于该原始对象生成一个代理对象

5、把最终的生成的代理对象放入单例池(singletonObjects)中, 下次getBean时直接从单例池拿即可当然bean的整个生命周期很复杂, 还有很多的步骤, 这里就不一一列举了。

Spring解决循环依赖是通过三级缓存，对应的三级缓存如下所示：



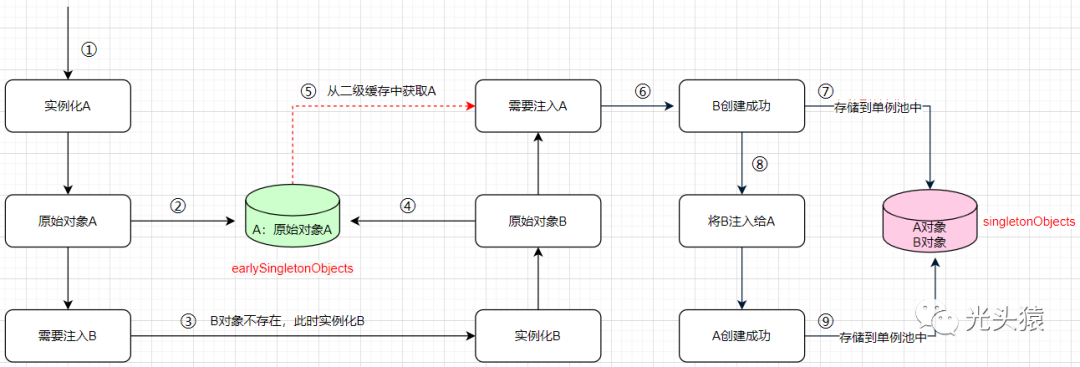
一级缓存(singletonObjects): 单例池; 缓存已经经历了完整声明周期, 已经初始化完成的bean对象

二级缓存(earlySingletonObjects): 缓存早期的bean对象(生命周期还没有走完)

三级缓存(singletonFactories):

缓存的是ObjectFactory, 表示对象工厂, 用来创建某个对象的

二级缓存的作用：如果要想打破上述的循环 , 就需要一个中间人的参与, 这个中间人就是缓存。



步骤如下所示：

1、实例化A得到A的原始对象

2、将A的原始对象存储到二级缓存中

3、需要注入B，B对象在一级缓存中不存在，此时实例化B，得到原始对象B

4、将B的原始对象存储到二级缓存中

5、需要注入A，从二级缓存中获取A的原始对象

6、B对象创建成功

7、将B对象加入到一级缓存中

8、将B注入给A，A创建成功

9、将A对象添加到一级缓存中

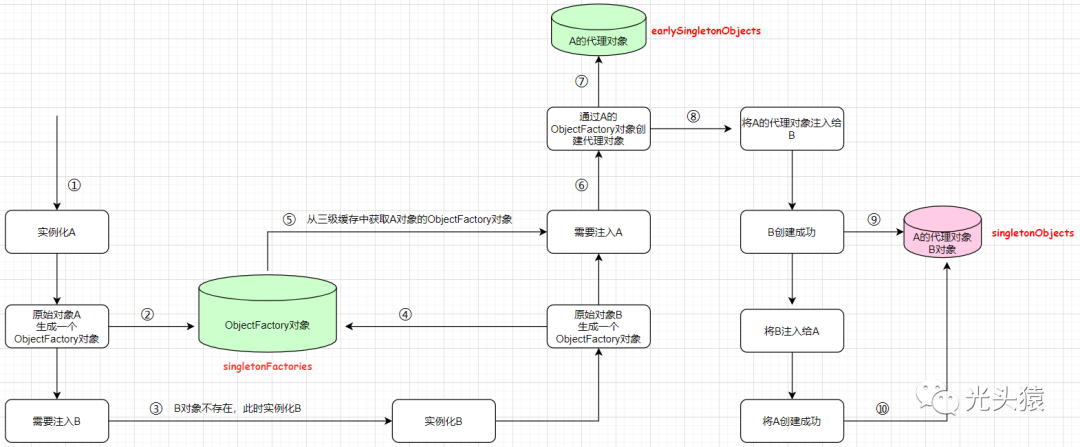
三级缓存的作用：

从上面这个分析过程中可以得出，只需要一个缓存就能解决循环依赖了，那么为什么Spring中还需要三级缓存 ？

基于上面的场景想一个问题：如果A的原始对象注入给B的属性之后，A的原始对象进行了AOP产生了一个代理对象，此时就会出现，对于A而言，它的Bean对象其实应该是AOP之后的代理对象，而B的a属性对应的并不是AOP之后的代理对象，这就产生了冲突。也就是说， 最终单例池中存放的A对象（代理对象）和B依赖的A对象不是同一个。所以在该场景下， 上述提到的二级缓存就解决不了了。那这个时候Spring就利用了第三级缓存singletonFactories来解决这个问题。

singletonFactories中存的是某个beanName对应的ObjectFactory，在bean的生命周期中，生成完原始对象之后，就会构造一个ObjectFactory存入三级缓存中，后期其他的Bean可以通过调用该ObjectFactory对象的getObject方法获取对应的Bean。

整体的解决循环依赖问题的思路如下所示：



步骤如下所示：

1、实例化A，得到原始对象A，并且同时生成一个原始对象A对应的ObjectFactory对象

2、将ObjectFactory对象存储到三级缓存中

3、需要注入B，发现B对象在一级缓存和二级缓存都不存在，并且三级缓存中也不存在B对象所对应的ObjectFactory对象

4、实例化B，得到原始对象B，并且同时生成一个原始对象B对应的ObjectFactory对象，然后将该ObjectFactory对象也存储到三级缓存中

5、需要注入A，发现A对象在一级缓存和二级缓存都不存在，但是三级缓存中存在A对象所对应的ObjectFactory对象

6、通过A对象所对应的ObjectFactory对象创建A对象的代理对象

7、将A对象的代理对象存储到二级缓存中

8、将A对象的代理对象注入给B，B对象执行后面的生命周期阶段，最终B对象创建成功

9、将B对象存储到一级缓存中

10、将B对象注入给A，A对象执行后面的生命周期阶段，最终A对象创建成功，将二级缓存的A的代理对象存储到一级缓存中

注意：

1、后面的生命周期阶段会按照本身的逻辑进行AOP, 在进行AOP之前会判断是否已经进行了AOP，如果已经进行了AOP就不会进行AOP操作了。

2、三级缓存: 缓存的是一个ObjectFactory，主要用来去生成原始对象进行了AOP之后得到的代理对象，在每个Bean的生成过程中，都会提前暴露一个工厂，这个工厂可能用到，

也可能用不到，如果没有出现循环依赖依赖本bean，那么这个工厂无用，本bean按照自己的生命周期执行，执行完后直接把本bean放入singletonObjects中即可，如果出现了循环依赖依赖了本bean，则另外那个bean执行ObjectFactory提交得到一个AOP之后的代理对象(如果有AOP的话，如果无需AOP，则直接得到一个原始对象)。

# 21、只有一级缓存和三级缓存是否可行？

不行，每次从三级缓存中拿到ObjectFactory对象，执行getObject()方法又会产生新的代理对象，因为A是单例的，所有这里我们要借助二级缓存来解决这个问题，将执行了

objectFactory.getObject()产生的对象放到二级缓存中去，后面去二级缓存中拿，没必要再执行一遍objectFactory.getObject()方法再产生一个新的代理对象，保证始终只有一个代理对象。

总结：所以如果没有AOP的话确实可以两级缓存就可以解决循环依赖的问题，如果加上AOP，两级缓存是无法解决的，不可能每次执行objectFactory.getObject()方法都给我产生一个新的代理对象，所以还要借助另外一个缓存来保存产生的代理对象。

# 22、构造方法出现了循环依赖怎么解决？

Spring中大部分的循环依赖已经帮助我们解决掉了，但是有一些循环依赖还需要我们程序员自己进行解决。如下所示：

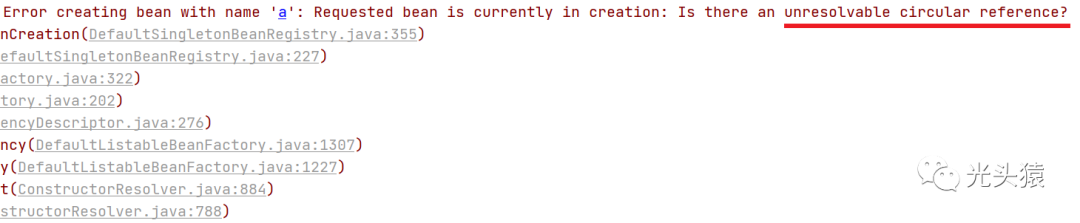
@Componentpublic class A {    *// B成员变量* private B b;    public A(B b){ System.out.println("A的构造方法执行了..."); this.b = b ; }}

@Componentpublic class B {    *// A成员变量* private A a;    public B(A a){ System.out.println("B的构造方法执行了..."); this.a = a ; }}

main方法程序：

@RunWith(SpringJUnit4ClassRunner.class)@ContextConfiguration(classes = SpringConfig.class)public class AccountServiceTest {    @Autowired private A a ;    @Test public void testTransfer() throws Exception { System.out.println(a); }}

控制台输出：



解决方案：使用@Lazy注解

@Componentpublic class A {    *// B成员变量* private B b;    public A(@Lazy B b){ System.out.println("A的构造方法执行了..."); this.b = b ; }}

在构造参数前面加了@Lazy注解之后, 就不会真正的注入真实对象, 该注入对象会被延迟加载 ， 此时注入的是一个代理对象 。