**Spring事务**

1. 操作数据必须保证 ACID
   1. 原子性
   2. 一致性
   3. 持久性
   4. 隔离性 (针对并发场景，多线程下屏蔽只允许一个线程操作)
2. Spring 也封装对数据的库的事务操作并且 扩展了一些功能，当对数据进行多次操作时 也可以实现事务的控制
   1. .在数据库4种隔离级别的基础上 加入 事务传播概念

传播 ： 多个方法之间调用的时候 事务的传递性

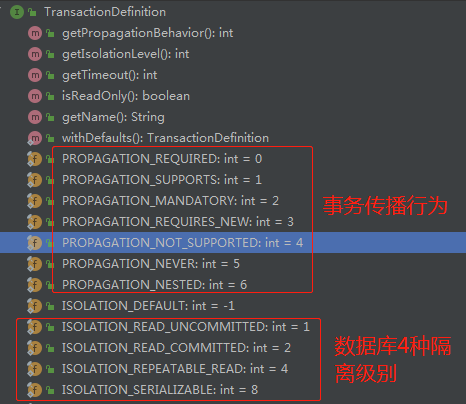
* 1. 提供多个事务合并 或者 隔离的功能
  2. 提供声明式事务

使用 aop 让业务代码与事务分离 ，只需通过配置即可实现事务

Aop实现是spring提供的事务拦截器 TranactionInterceptor

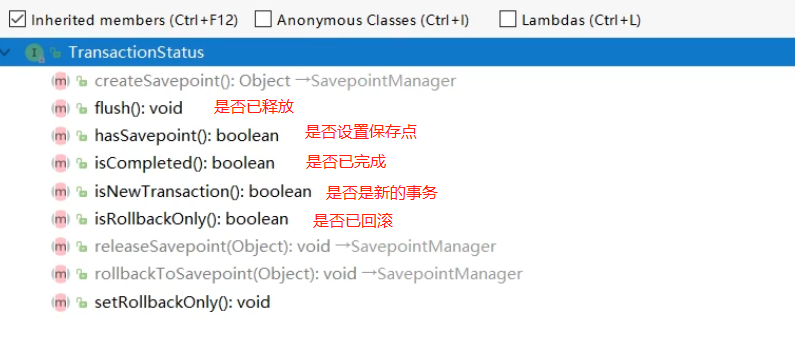
另外还有编程式事务 (使用代码实现事务开启、设置保存点、提交、回滚等操作)

1. Spring 事务相关的接口
   1. TransactionDefinition 接口 ： 定义事务的 4种隔离级别 + 7种传播行为



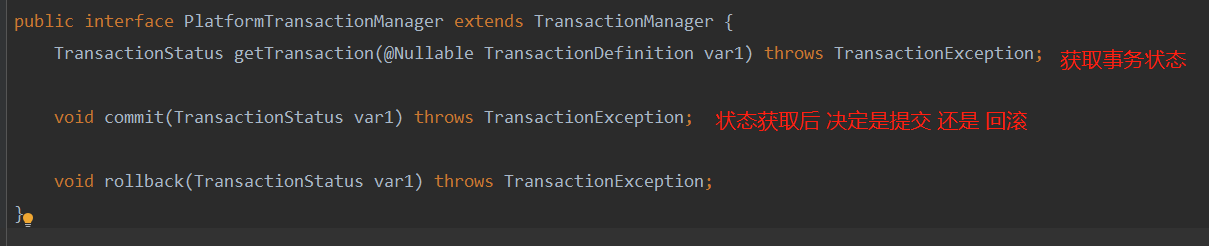
* 1. TransactionAttribute 接口 ： 扩展了TransactionDefinition 扩展了 事务回滚功能
  2. PlatformTranscationManager接口 ： 事务管理器包括

1. 获取事务执行状态 TransactionStatus

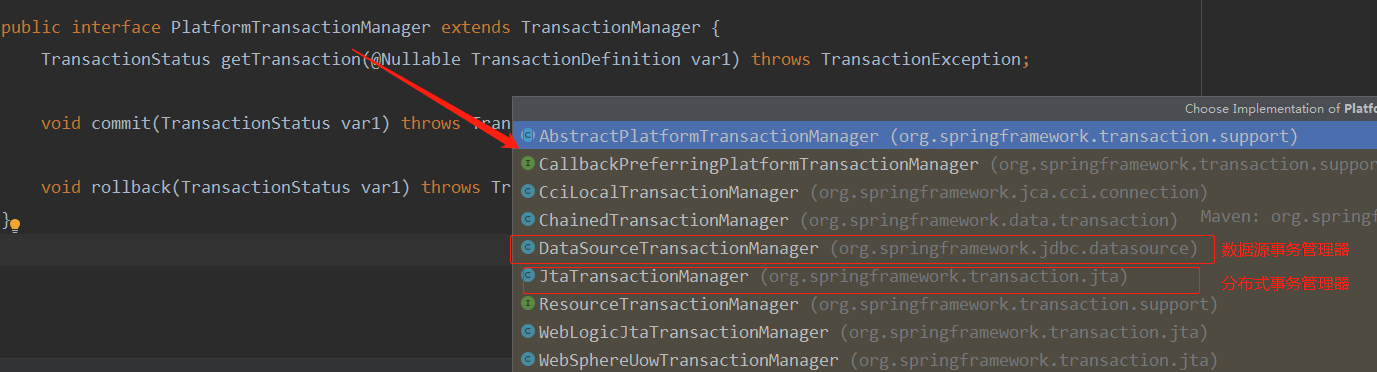


事务保存点的作用 ： 一个事务当对表进行多次操作时 ， 如果发生错误需要 回滚， 不是全部回滚 而是回滚到指定的保存点。 保存点之前的操作时持久 化的

1. 提交事务
2. 回滚事务



* 1. 事务管理器 实现类



DataSourceTransactionManager 事务管理器是如何保存连接的？

DataSourceTransactionManager 依赖了数据源 ，从数据源中获取连接然后保 存至ThreadLocal中 ， 也没有实现切换数据源的功能

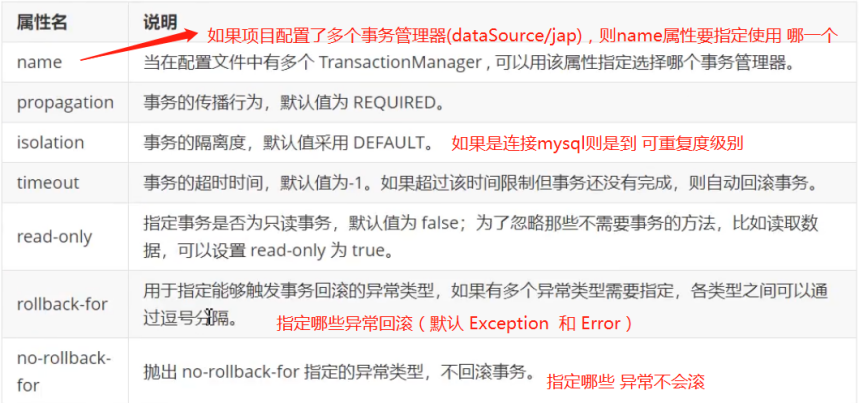
**因此 ：spring在一个事务中是不能切换数据源的 。**

**Spring提供你了 ImportBeanDefinitionRegister 实现切换数据源的功能 是多 个事务的**

**为什么spring 在夸库操作时 事务会不起作用 ？**

**因为 默认的事务管理器用的是** DataSourceTransactionManager

1. 声明式 事务的使用
   1. 使用 @Transaction注解



* 1. 7种传播行为

1. Required (默认)
2. Requires\_new

该方法只要被调用就会新创建一个新的事务， 且不影响其他的

1. Supports
2. Not\_support ： 声明此方法不需要事务， 用于读操作 。 如果被事务方法调 用。那么事务会先挂起

@Transactional(propagation=Propagation.NOT\_SUPPORTED)

1. Mandatory [ˈmændətɔːri](强制的)
2. Never
3. Nested(嵌套)[ˈnestɪd]

注意：

1. 当事务方法调用 另一个事务方法时 ， 必须指定事务传播
2. 事务默认回滚的情况是 RuntimeException 或者 Error

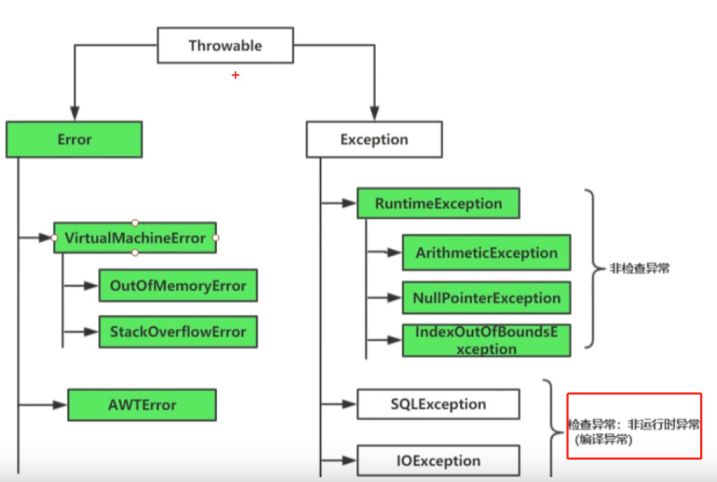
1. 事务失效的问题
   1. 使用事务的类 必须是aop 代理对象
   2. 当事务方法调用 另一个事务方法时 不指定传播行为 会失效

因为内部直接调用方法， 其实是this.xx(); this不是代理对象

* 1. 方法时 public修饰的
  2. 如果数据库使用myisam 也不支持事务，只有表锁
  3. 默认只支持 RuntimeException 和 Error异常 不支持检查异常，如果抛检查异常不能回滚 。

一般自定义异常时 使用 Exception ，而不是RuntiomeException 处理事务报错时 就不能回滚 ，如果需要Exception 则 重新定义rollbackFor属性

 @Transactional(rollbackFor=Exception.class)



**Spring**

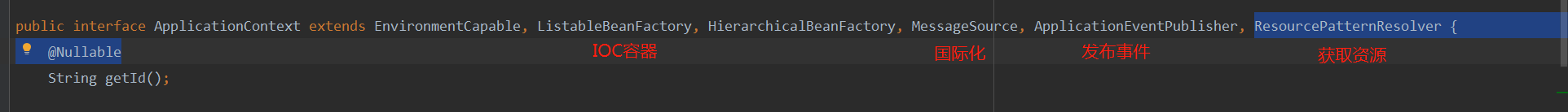
1. 什么是IOC ：控制反转？

IOC 就是将项目中写的对象 定义成bean对象 然后交给spring管理 ，在项目需要对象的时候可以通过 DI 进行注入

1. ApplicationContext是什么 ？

ApplicationContext首先是spring容器 继承了 BeanFactory， 他不仅是一个容器 spring还对他扩展了许多功能 ， 跟他继承的接口有关 如：

实现国际化、发布事件、获取资源 (读取classpath目录、以及运行时环境 虚拟机参数) 等



1. 在spring中创建一个bean 的方式有哪些
   1. 通过 xml 的 <bean> 标签
   2. @bean 返回是多例的
   3. @Controller、@Server、@Compent 等
   4. @Configuration
   5. 通过AnnotationConfigRegistry接口调用 register方法注册一个bean ，然后refresh至容器内
   6. @Import 直接注入类的 .class
   7. 接口ImportSelector

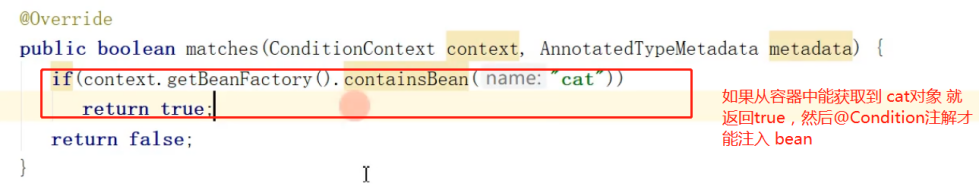
selectImport()方法会返回一个String数组，数据中是 全限定类名。 会将返回的这 些全限定类名加载至spring 容器中 。 springbo的自动配置的 ImportBeanDefenitionRegister 类就是通过这个接口实现的

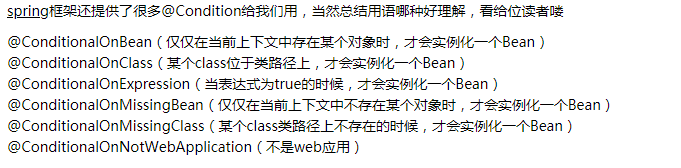
由于数据的特性是有序的 ，对象所以加入到 spring容器中也是有序的

* 1. 通过AutoConfiggurationSelector 类，加载一个类。Springboot启动时有使用到，实现了 ImportSelector 接口
  2. BD注册器： ImportBeanDefenitionRegister 注入一个bean至容器

1. @Condition 条件注解

使用@Bean注解注册bean时 ，可以使用 @Condition来进行条件判断或者业务处理 。 当满足@Condition的matches条件，才允许加载@Bean。 判断条件就是 实现 Spring 的Condition接口重写 matches方法





1. 获取在项目中获取容器

实现ApplicationContextAware 接口 ，这个接口在spring容器初始化后 ，会进行调用，然后将 ApplicationContext 传至当前接口中

1. 单例bean 引用多例bean ， 多例bean 默认是会变成 单例的，如果解决
   1. 每次使用多例bean 直接取容器中拿 ，不使用 依赖注入
   2. @Lookup ：注解作用也是每当需要新的对象 就到 容器中主动去取
2. springBean的作域 @Scope 6 种

（1）singleton 默认单例

（2）prototype 没次取都是新的

（3）request 单个请求中使用一个对象

（4）session 一次会话中

（5）application 真个项目的运行周期

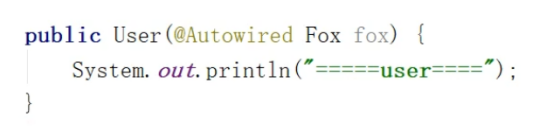
（6）webSocket

1. 依赖注入的实现有哪些
   1. 构造器注入
   2. 属性的set方法setXX(X x)
   3. 接口注入 （spring4之后已经不支持了）

@Atowarid 标注在属性上 ，默认就是使用反射 Fild.set() 方式注入 ，不是直接调用 默认生成的set方法

1. @Atowarid 和 @Resource 区别

@Atowarid 现根据类型找bean ，找不到再根据 name找 。 @Atowarid 可以以参数的 形式注入对象 。



@Resource是 jdk 提供的注解， 只能通过 name去查找对象 。 不能以参数的形式 注入对象

1. Spring bean相互引用的问题

有A 、B 两bean ， A、B作为各自的属性相互依赖 那么在spring容器实例话对象的时候就会 有一个问题 :

由于容器实例化对象 是有顺序的 假如：A被实例化时 B还不存 ，那么A也无法被实例化

但是默认是情况（@Scope(“单例”)）下是可以的 因为当spring 配置相互引用的bean时 会开辟缓存区 ，将A、B 放入然后再缓冲区创建对象（创建一次后）然后再将对象放入bean的工厂，然后销毁缓冲区

如果@Scope(“prototype”)是不可以的， 不会创建缓存区

1. @DependsOn

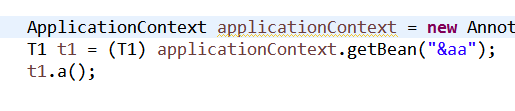
用于控制@Bea的初始化顺序当一bean 依赖另一bean时 可以使用 @DependsO注解

1. 如何使用 spring 动态代理
   1. 引入 aspectJ 依赖
   2. 开启 使用cegelib
   3. 创建 切面 @Aspect
2. 声明切点 @Pointcut
3. 声明前置通知@before、后置通知@After、环绕通知等@Around
4. 在注解的参数上配置 execution
5. Jdk 动态代理 和 cegelib的区别
   1. JDK 基于接口实现代理
   2. Cglib 基于继承实现代理
   3. Cglib 性能相对JDK 较高 (高一点)
      1. 基于接口 ，就必须通过反射 才能拿到 被代理类中的方法
      2. 基于继承，代理了中直接就有了 被代理的中的方法
      3. 但是他们 底层都是 使用字节码动态拼接 来生成代理类
   4. 适用场景
      1. 主要看项目需要 看是有了 接口 ，还是有了 父类
6. BeanFactory 和 FactoryBean 的区别

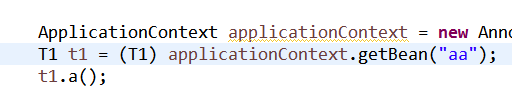
(1)BeanFactory： spring（ApplicationContext）容器的顶级接口 是产生bean 的工厂

(2)FactoryBean： 实现此接口的bean，不仅会被speing管理 。FactoryBean getObject方法中返回的bean 也会被 spring容器管理 (**默认是单例的，懒加载的**)

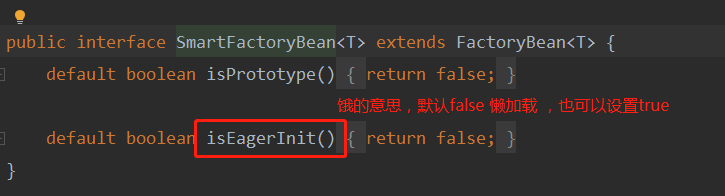
* + 1. 如果通过容器获取 实现FactoryBean 的bean时 ， 就通过 getBean(“&xxx”)



* + 1. 如果通过容器获取 getObject返回的 bean是 ， 就通过 getBean(“xxx”)



FactoryBean接口接口下有个实现类 SmartFactoryBean，可以实现恶汉

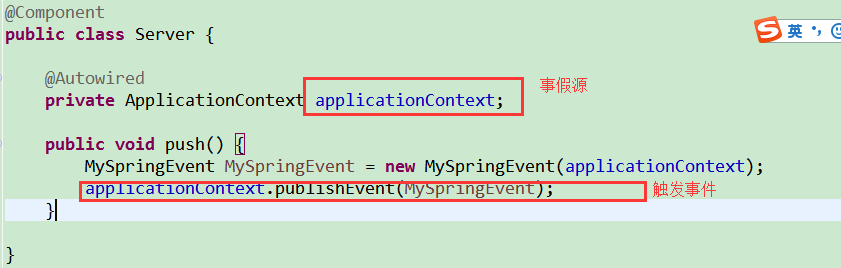


1. Spring中的事件 (观察者模式)
   1. 观察者模式包含3个部分
2. 事件 EventObject
3. 事件源 Source
4. 监听 @EventListener （当事件源发生变化时，触发监听）
   1. 在spring中如何定义事件 ： 实现ApplicationEvent ，ApplicationEvent 内部实现了 java 的EventObject 接口
   2. 在spring中如何定义监听 ：实现 ApplicationListener ，ApplicationListener内部实现了 java 的 EventListener

或者通过 事件监听注解@EventListener

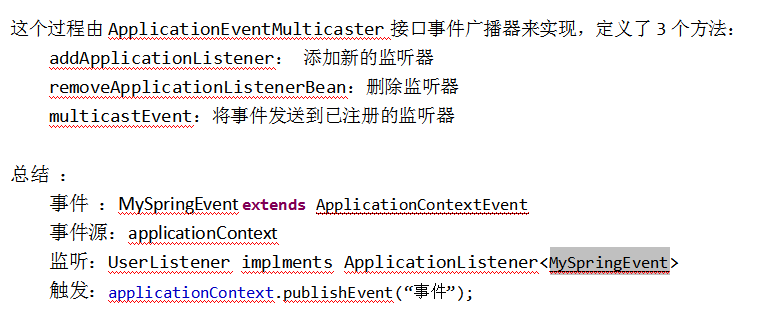
* 1. 如何发布事件来触发监听，实现ApplicationEvent的事件一pulish，就会触发监听

applicationContext.publishEvent(自定义事件);



* 1. spring是如何将事件发布给spring对应的的事件监听器的 ?

Spring 内部通过事件广播的机制 ，将注册的时间 广播给监听器



1. 自定义注解
   1. @Target 设置注解的定义的位置
   2. 运行时起作用 @Retention(RetentionPolicy.***RUNTIME***)
2. 消息转换器 MessageConverter

MessageConverter 作用 ： HTTP请求接收数据和发送数据都是以文本的形式 。

而springMVC 返回数据 和 接收数据时对象那， MessageConverter就是 负责量对象和文本相互转换的

Spring默认的转换器是 jackjon 也可以自定义 fasejson

1. SpringMVC 是如何启动的

WebApplicationContext 是springMVC 的上下文接口 ，当sring 调用 WebApplicationContext 时表示 ，启动SpringMVC 。启动springMVC有两种方式

1. ContextLoaderListener （常用）
2. ContextLoaderServlet
3. SpringMVC 在启动时是初始化 AnnotationConfigWebApplicationContext 容器，然后将把这个容器 交给 DispatcherServlet ，再去进行servlet的初始化
4. SpringMVC的Dispatcher的初始化总结：

DispatcherServlet的初始化 ，其实就是servlet的初始化 ， 加载DispatcherServlet.properties配置文件 ， 扫描Controller 将@RequestMapping中配置的url 跟controller进行保定保存至 map中 ，方便后续ReauestMapping 查找根据 requst请求找到对应的controller 返回ExecuttionChain

1. SpringMVC请求处理的过程

请求先进入servlet 的service方法饭后DispatcherServlet继承了servlet重写了service 方法会走 DispatcherServlet方法的Service()方法 ，然后进入doDiscapter()方法 ，

通过HttpServletReques 根据handlerMappin 获取 HandlerExecutionChain ， 里面封装 了controller ，然后 HandlerExecutionChain.getHandler获取到controller 。在通过 controller找到对应的 handAdapter 其实就是获取 SimpleControllerHandlerAdapter类 ， 然后调用他的 hand(controller) 方法 ，去执行业务返回 ModelAndView 。 然后在 通 过 ViewResolver去解析 ModelAndView 返回视图

**SpringBoot**

1. 启动注解 @SpringBootApplication 包含以下注解
   1. @ComponentScan 默认扫描当前目下写的所有子目录文件
   2. @SpringBootConfiguration
   3. @EnableAutoConfiguration Springboot自动配置注解
2. 依赖 @Impot(AutoConfigurationImportSelector)这个自动配置类 ， 在springboot启动时 会调用 AutoConfigurationImportSelector的selectImports方法加载 spring.factores 配置文件中配置的权限定类名的集合数组。 然后返回交给spring 去初始化这些类放入spring容器。

spring.factores：是springboot的自动配置文件 ， 里面配置了在springBoot启 动需要加载的模块

1. Springboot的 yml 配置文件
   1. 项目中的时间转换
2. 通过 spring.jackson.date-formate: yyyy-MM-dd HH:mm:ss

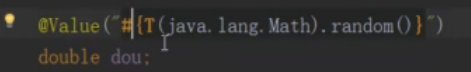
* 1. 读取yml中配置的字符串 可以直接映射成 一个类

通过 @ConfigurationProperties(prefix = “xxx”)

1. @Value
   1. # 执行 spel 表达式 。 spel甚至可功能更强大支持逻辑运算
2. @Value(“#{${1} + ${1}}”)

Private String str;

B、



* 1. $ 从配置文件中获取值

1. Springboot是如何将一个自定义的项目 ，整合至 springboot项目中(**自定义starter组件，实现自动装配**)

自定的项目，肯定是要依赖spring的 。 但是自定义的项目最终是被打成 jar包引入 springboot的 pom文件中 。 但是有一个问题，由于是当成 jar引入的，当前的项目 是扫描不到自定义的jar项目 。

解决 ：

1. @SpringBootApplication 主机内部有 @EnableAutoConfiguration注解 ， 这个注解

[依赖@Import(AutoConfigGurationIMportSeletor.class)](mailto:依赖@Import(AutoConfigGurationIMportSeletor.class))

这个类内部去加载 springFactorie文件里面定义的是springboot需要装载的类 ， 然后返回数组的形式返回。 将返回的权限定类名去实例化放入spring容器中

1. 可以通过spi 机制，在自定义的jar项目中 resouce目录下的 mate-info目录 下 配置springFactories文件 自定义项目的主要的一些类 。 在springBoot启动时就 会去加载这个文件 。 在文件中配置 自定义项目的主配置类。 来启动jar
2. . 再自定义一个注解 ， 这个注解的作用 是标注的 springBoot项目的配置类上， 让其具有使用 jar项目的功能。 实现动态插拔
3. 自定义jar如何实现动态 插拔？
4. 自定义注解，注解中上标记 @Import(xxxx.class)实现xxx类注入至spring容器 中的功能
5. Jar项目的主配置类 需要使用 @ConditionOnBean(xxx.class)

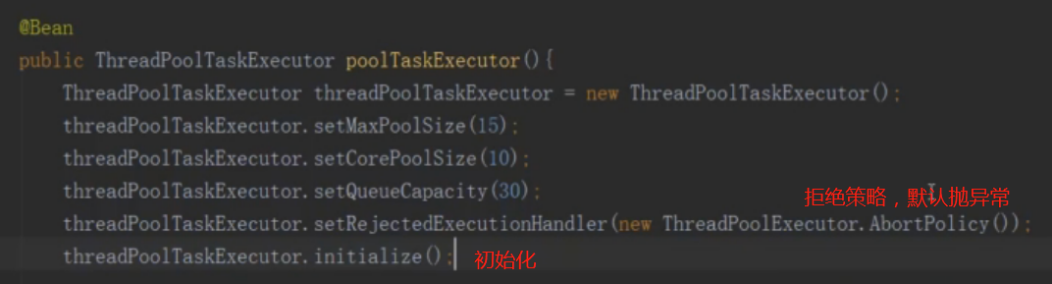
@ConditionOnBean作用 ： spring容器中须依赖 xxx.class 类，如果没有不能执行该配置文件。而自定义的注解实现的功能是注了的 xxx.class 。

所以当前springboot项目 ：只有加了自定义注解，往spring容器中加了xxx.class 才能 使用自定义项目

1. Springboot 在启动时如何 加载 springMVC的
   1. springboot首在启动时，会读取 springFactories配置文件，将文件中配置的类以数组的形式返回 ，然后加载至spring其中 DispatcherServletAutoConfiguration 这个类 就是springMVC的配置类 ，他主要做了两件事情
2. . 创建一个 DispatcherServlet对象然后将其放入 spring容器中
3. . 然后获取springMVCd的项目配置文件，将配置文件 和 DispatcherServlet注册至 DispatcherServletRegistrationBean中，并设置servlet在启动tomcat时的loadOnStartup(-1) 。 当servlet接收第一个请求时才初始化
4. 配置webAutoConfiguration 配置 httpMessageConverter ，默认使用Jackson

也可以是使用 fastJson 通过@Bean来指定

1. 配置 ViewResolver 视图解析器
2. 配置 Converter 参数转换
3. 配置 MessageConverter 默认使用 Jackson
4. Spring也 提供了线程池的创建类



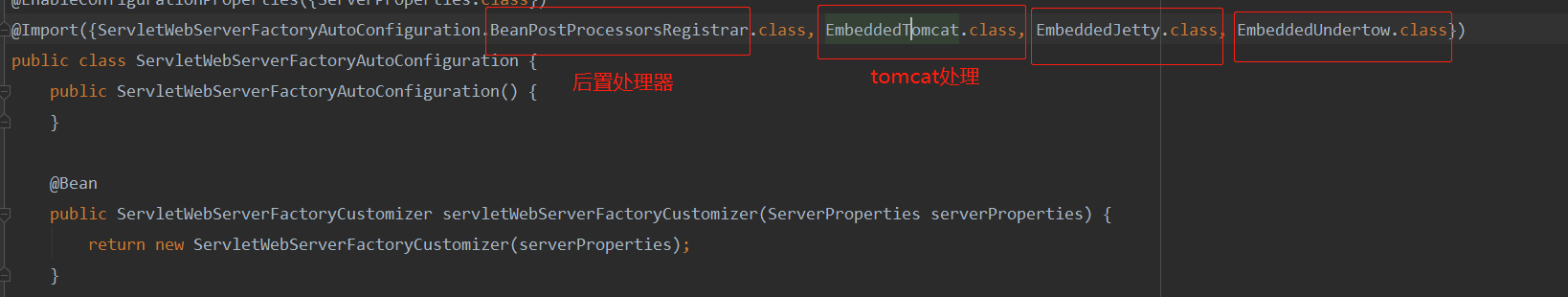
1. Springboot 默认的 web 容器 tomcat , 也可以自定义其他的 web容器
   1. Tomcat
   2. Netty
   3. Jetty 等
2. springBoot如何自动配置tomcat

启动时加载springFactories 配置文件初始化 ServletWebServerFactoryConfiguration

主要做的事情 ：

1. .读取配置文件类ServerProperties (这个类会读物 yml文件 ，然后将 server 的配置信息封装)
2. 通过@Import注解依赖 注入 4个对象 ，通过配置 tomcat是通过 EnableTomcat.class来实现的 ，nableTomcat.class 会创建TomcatServletWebServerFactory

对象，



**注意 ：** @import一次性注入4个.class数组 ，那么在放入spring容器初始化时，他们也是有顺序的， 首先会加载 tomcat容器 ，有tomcat容器就不再加载其他容器。 tomcat容器没有 再加载其他类型的容器 。 因此想给springboot自定义 其他容器 必须删除tomcat内置的依赖，让springboot加载其他的依赖

1. 后置处理器的作用
2. springboot如何使用其他web容器
   1. 使用 <exclusion> 标签 提出tomcat依赖
   2. 使用其他的容器的实现
      1. 引入jar包
      2. 在配置文件中使用 @bean 注入实现类
3. Springboot 热部署原理

通过自定义类加载器 动态的加载class文件

1. Spring中的 核心类概念
2. BeanDefinition 相关

A、 BeanDefinition 用于定义一个 bean ， 封装了bean的属性

B、ClassPathBeanDefinitionScanner 通过路径扫描class文件 ，

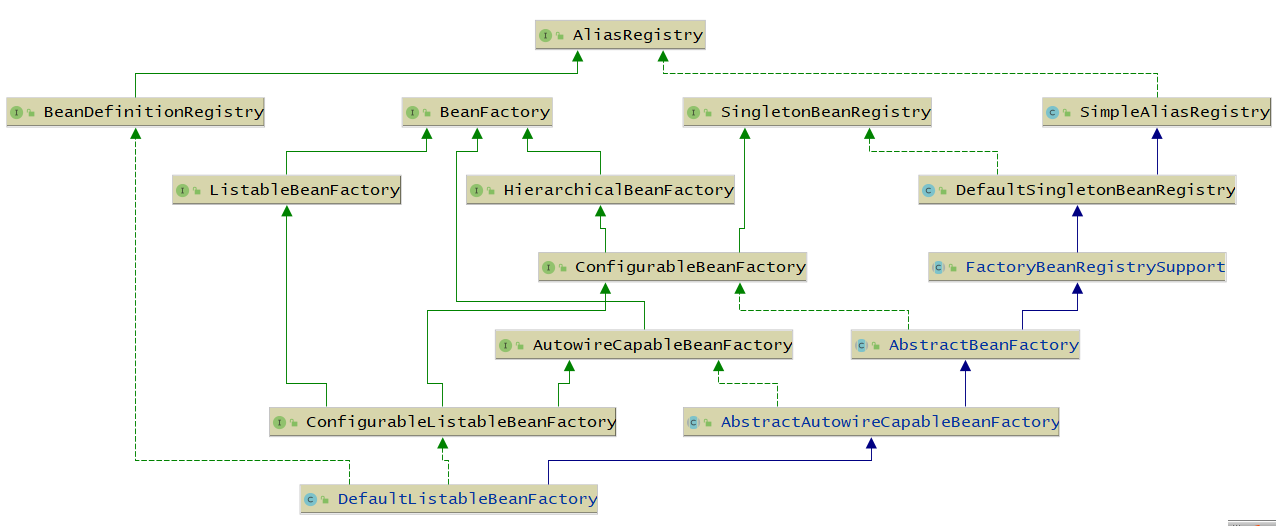
并将class文件然后解析类上的注解，如果标注了@Compontent、@Bean等 注解可以将bean交给spring容器管理的注解，就将这些class 加载成 BeanDefinition 对象

C、AnnotatedBeanDefinitionReader解析类上非@Compontent 注解包括 @Conditional，@Scope、@Lazy、@Primary、@DependsOn、@Role、 @Description等 ， 然后将类加载成BeanDefinition 对象

1. SingletonBean 单例bean

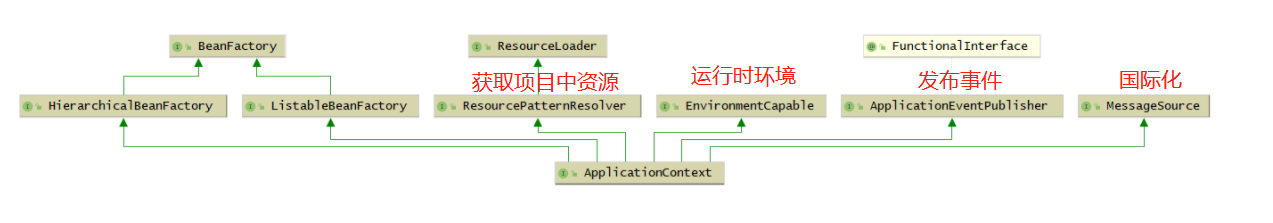
用于存储单例bean ，初始化的时机 refresh之后

1. BeanFactory 中的 **DefaultListableBeanFactory**

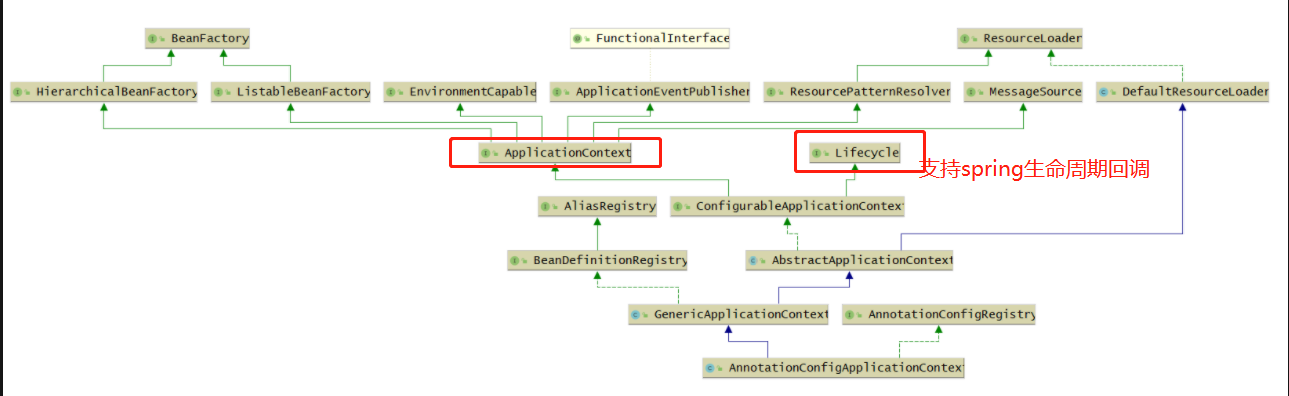


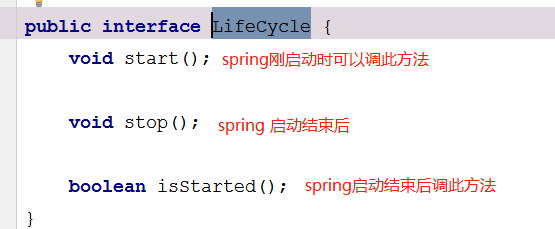
**总结：** **DefaultListableBeanFactory具有的能力**

1. **操作BeanDefenition (注册、获取、移除)**
2. **操作bean的别名**
3. **操作bean** 
   1. **具有beanFactory 获取bean ， 获取父容器bean、销毁bean**
   2. **操作单例bean (注册、获取、移除)**
   3. **Autowried对bean自动装配**
   4. **支持bean后置处理器**
4. ApplicationContext

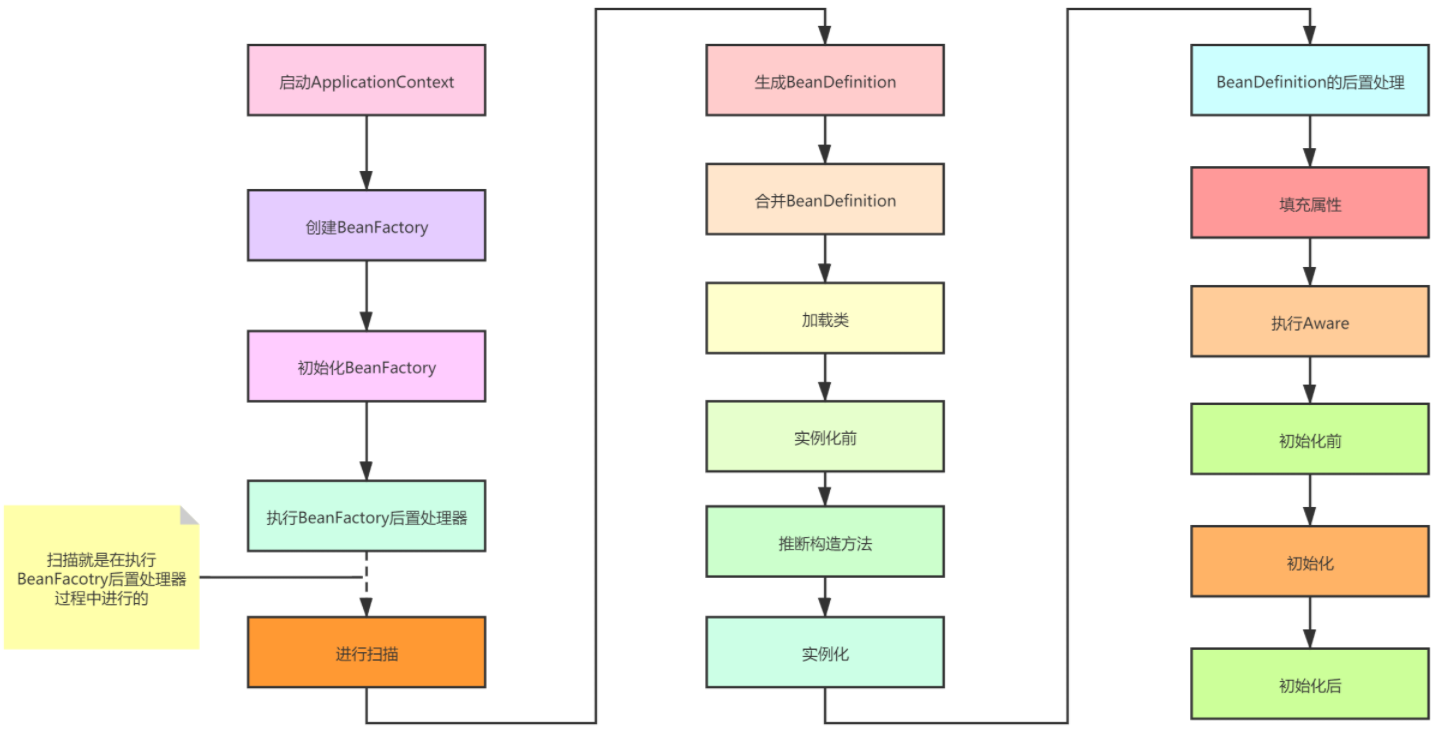


1. HierarchicalBeanFactory 用于 BeanFactory 的功能
2. 拥有获取 BeanNames的功能
3. AnnotationConfigApplicationContext





1. 真正创建出来的单例bean ，是放在单例池中 通过BeanFactory.getBean() 也是从单 例池获取中 获取bean
2. **Spring生命周期**



1. **初始化BenaFactory**
2. **扫描Bean 为创建bean做准备** 
   * 1. 扫描 实例化 BD

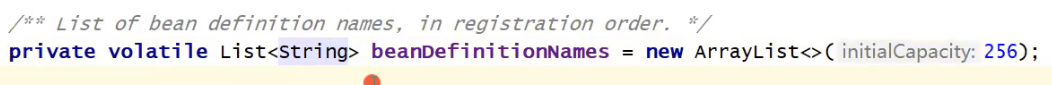
Spring 通过asm 字节码扫描技术 ，将标注 @Compotent相关注解的类加载 至内存 生成BeanDfinition对象（包括普通bean 和 抽象bean）。

如果有抽象类也会被实例化成BD (abstract = ture). 如果一个普通类实现了 抽象类那么 最终会生成两个BD 。 一个是abstract类型的一个普通类型 的 。对于实现了抽象类的bean他们的BD会进行合并.然后合并后的bd在存 至新的 mergeBeanDfinitionMap中

为什么进行BD合并

因为生成的子BD都不是完整的 ， 子类中还缺少父类中的属性以及父类上被标注的注解 。

* + 1. 接着获取项目中所有的 BeanBefinitionNames 的名字的集合



For循环遍历所有的 bd 然后根据 拿到 不是抽象的、是单例 且 非懒加载的 bd ，然后进行bean合并 并返回合并后的bd 。

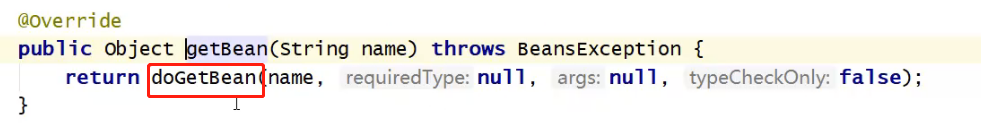
* + 1. 判断 bd是否是 smartFactoryBean(因为smartFactoryBean有实现恶汉的功能)

拿到合并的bd后，再通过 & + beanNmae. 获取 bd，然后判断当前bd 是否是 SmartFactoryBean(因为这个接口 可以设置bean是懒加载或者饿加载)

A、**不是** 调用getBean(beanName) 方法创建bean

B、**是** SmartFactoryBean就 判断是否是懒加载，如果不是懒加载就创 建bean就调用getBean(beanName) 方法创建bean

* + 1. getBean(beanName)---------------> doGetBean(beanName)



**doGetBean(beanName)**

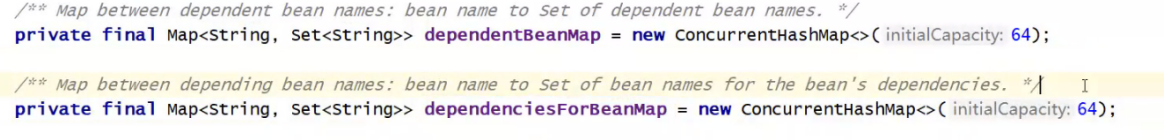
1. 先通过 beanName 到单例池缓冲池中找是否存在
   1. 存在 返回
   2. 不存在
      1. 单例池中没有 ，通过beanName 查找 父BeanFactory容器中是否存在

(spring + springMVC项目中 springmvc就是子容器)

* + 1. 父容器不存在 ，接着判断 @DependsOn(“xxx”)注解

1. 通过 beanName拿到合并后的bd 然后判断当前 bd 是否加了 @DependsOn(“xxx”) 注解 (表示当前bean必须依赖xxxbean才能创建)

在扫描bean时 ， 会将标注@DependsOn(“xxx”) bean的beanName 和 存放 xxx beanName 分别存储至两个 map



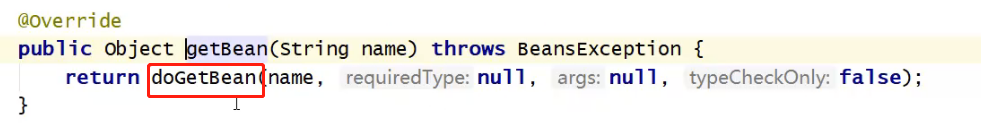
1. 如果存在 @DependsOn(“xxx”) 注解

拿到 DependsOn bean的beanName ， 从循环依赖的map中去检查 是否存在循环依赖

如果 bean通过 @DependsOn 进行互相依赖 ， 就抛异常不创建

1. 如果不存在 @DependsOn(“xxx”) 注解

再去调用 getBean(“xxx”) ,去创建 xxx bean， 应为当前xx Bean依 赖xxx ，xxx必须先创建再去 递归 getBean(xxx)方法



接在会再 doGetBean()方法 重复之前的操作 查找单例池等....

* + 1. 最后如果不是循环依赖 ，就会再次判断 合并后的 bd是单例 还是 原型的 还是其他作用域的 。如论是哪一种方法 最终都会调用 createBean方法

1. 如果是 单例的

1. 再一次判断 当前对象是否在单例池中
2. 先在threadLocal中保存当前bean开始被创建的开始标识(循环依赖)
3. 创建调用createBean(mergBD,beanName)方法
4. 将threadLocal中保存当前bean的标识删除

d、将创建的bean 保存至单例池 ，然后返回创建的bean

1. 如果是原型的

a.不判断，直接创建

b.先在threadLocal中保存当前bean开始被创建的开始标识(循环依赖)

c.创创建调用createBean(mergBD,beanName)方法

d.将threadLocal中保存当前bean的标识删除

e.然后直接返回

1. 其他作用域是一起判断的(Request 、Session )

在类生成BD时，会解析类中@scope注解 ，然后将 当前beanName 与 其对应的 scope 的关系保存在 map中

1. 先判断具体是哪一种作用域 ，通过单签beanName到map中获取自 己的作用域
2. 然后到自己的作用对应 map中去取 看是否存在 ，如果不存在则创建
3. 创建时根据 scope接口 有两个实现类 RequestScope、SessionScope 然后根据自己的作用域 去调用不同的实现类创建对象 ,在创建对象 时内部还是调用createBean(mergBD,beanName)，创建完对象后 然后 通过RequestContextHolder 获取 RequestAttributes 对象 。

然后把对象set至 RequestAttributes .setAttribute(object , 作用域 )

内部还是通过

request.setAttribute() ; //往request 的map中赋值

request.getAttribute().get(“”) // 从request中取值

session.setAttribute() ; //往session的map中赋值

session.getAttribute().get(“”) // 从session中取值

1. 然后返回对象

**8)createBean(mergBD,beanName) 方法创建对象**

参数1 ： beanName

参数2： 合并后的bd

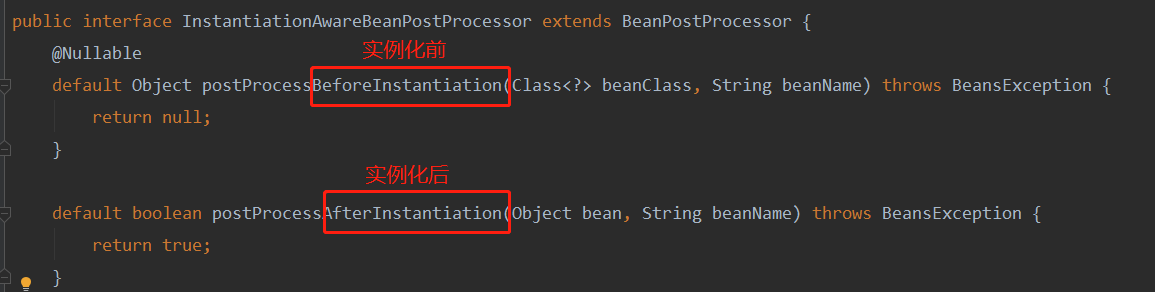
参数3： args

**A、**先获取类加载器

根据当前 mergeBD获取类的对应的类加载器 ，将对应的类加载器 设置到当前的 mbd 的属性中 ，当前 mbd就可以通过反射去实例化bean

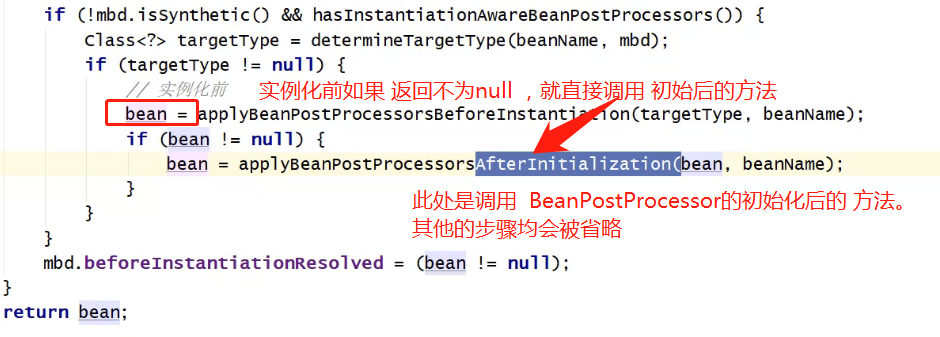
**B、** 实例化前 调用后置处理器BeanPostProcessor接口干涉 bean 的实例化过程

(实际上BeanPostprocess的接口没有提供实例化前、后的扩展点 而是**InstantitionAwareBeanprocessor** 接口 继承了 BeanPostProcessor 接口 ，并额外提供了方法 供对象实例化前后扩展使用)



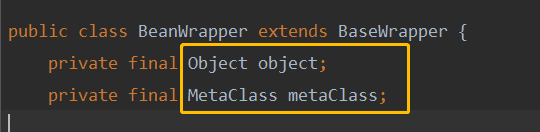
**注意** ： 实例化前默认的方法是 为null 的，如果从写此方法 返回实例

表示已经通过扩展创建了对象，不用在通过spring来创建对象了 然后逻辑会直接跳到BeanPostPorcessor出初始化之后的方法 。



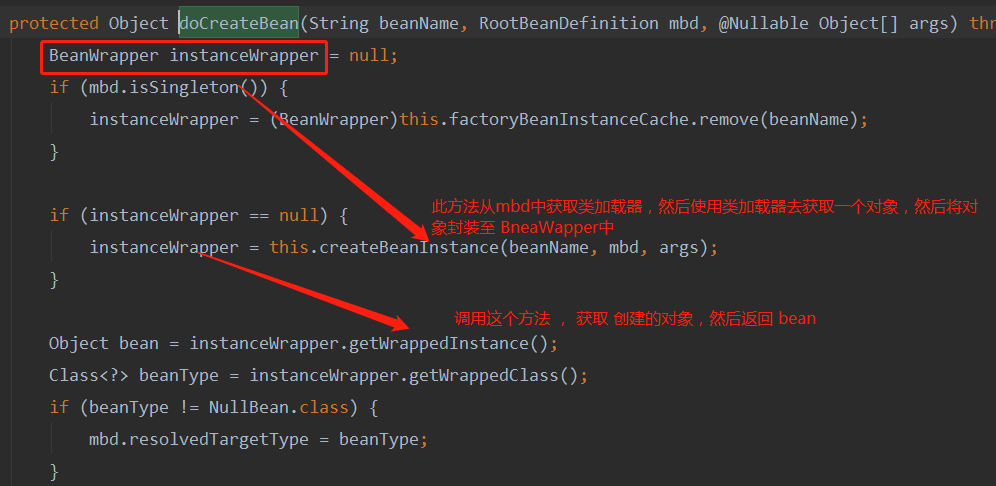
**C、**实例化

根据mbd中保存的classLoader类加载器 获取 BeanWapper 类 （beanWapper 是 spring中 bean的封装类 包 括了 实例化对象 和 对象的元信息）



获取对象：





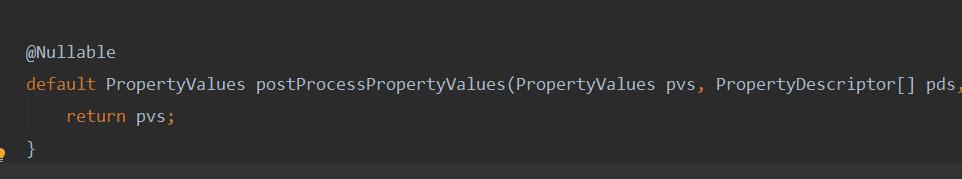
**D、**实例化后 调用后置处理器BeanPostProcessor接口干涉 创建后的bean

**E、**执行 BeanDefinition的后置处理器 (因为对象只是被实例化后面还要对bean属性进行填装、以及对象的初始化 ，在这个阶段是可以对 bd进行扩展的。 如：修改bean 的scope l、azy等)

**F、**填充属性( 将当前创建的对象 完成属性赋值 )

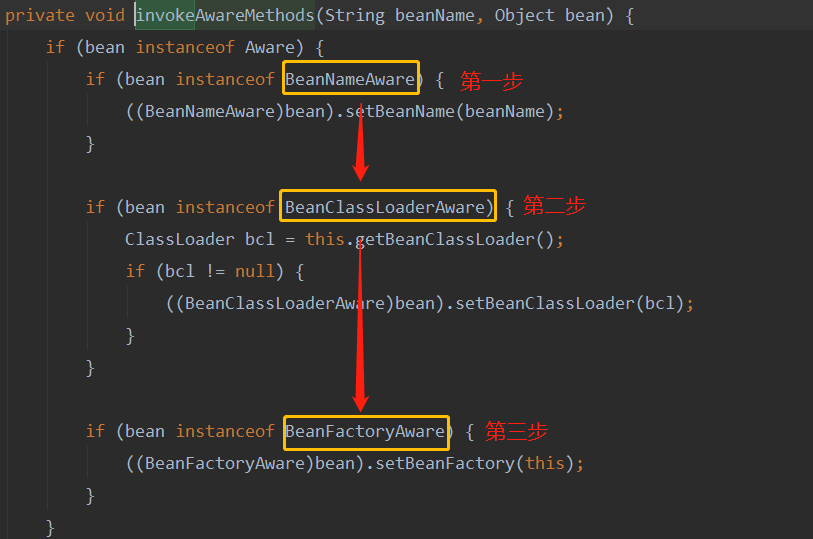
主要做的事情就是 解析 @Autowried @Resource @Value 注解

**G、**填充属性后 **InstantitionAwareBeanprocessor**



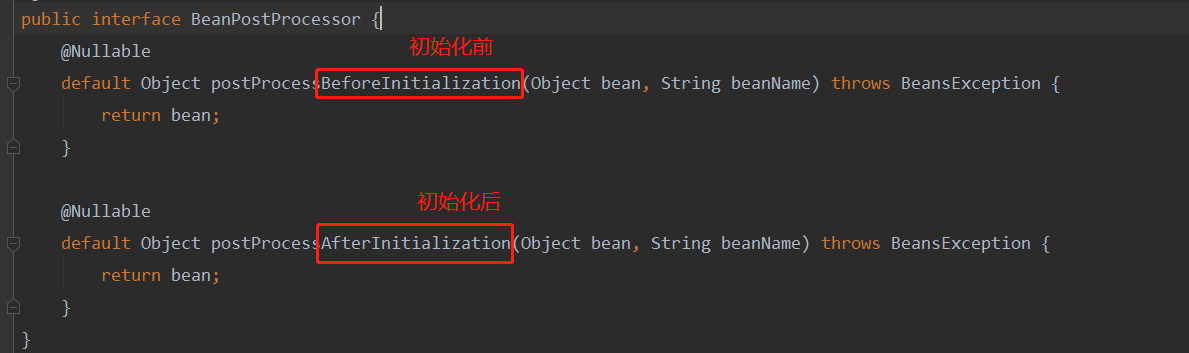
**H、**执行aware接口 并且有顺序

1. . BeanNameAware ( 对别名进行操作)
2. .BeanClassLoaderAware ( 可以获取classLoader并进行操作)
3. .BeanFactoryAware ( 可以获取beanFactory 并进行操作 )

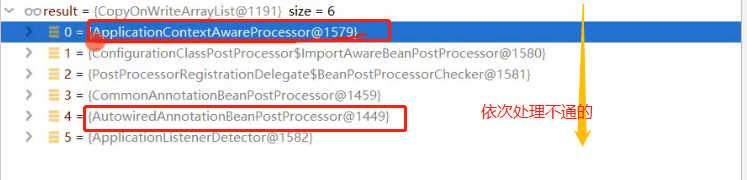


另外还有 ApplicationContextAware 也可以获取spring 的context容器来执行更多的功能。但是ApplicationContextAware 执行的时机不再此处 。他 以及还有其他的aware是在对象初始化前的 BeanPostProcessor中执行的 。他的优先级低于beanFactory Aware

**I、**初始化前 拿到实现了 BeanPostProcessor的类然后遍历去执行BeforeInit方法



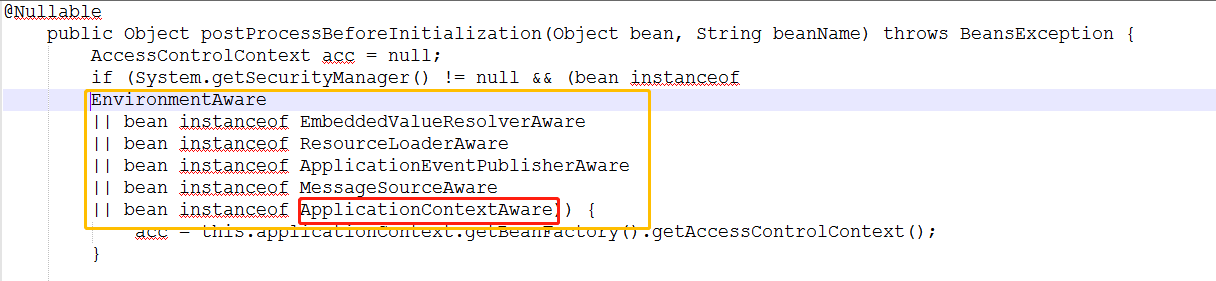
这个接口会被很多的类实现 ，然后会依次执行



**@PostConstract ：** CommonAnnotationBeanPostProcessor

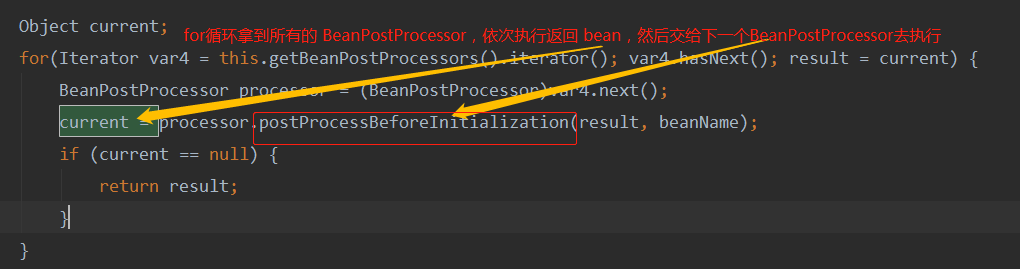
**@Autowired ：** AutowiredAnnotationBeanPostProcessor

其中也有其他的aware接口会在ApplicationContextAwareProcessor 中被调用然后去 其他的 aware 接口实现类



注意 ：

项目中如果有多个类实现BeanPostProcessor 接口 ，那么会依次执行所有的实 现类 ，将上一个 实现类返回的 bean 交给下一个BeanPostProcessor 实现类 去执行 ，形成链式调用 。 而 **@Autowired、@PostConstract 、@PreDestroy** 功能就是由不同的后置处理器去实现 。



**注意 ：**

如果有其中一个 BeanPostProcessor 接口的实现类 重写的方法 return null;

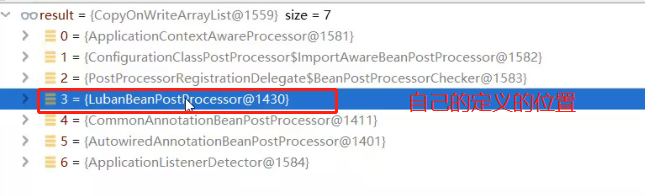
那么将直接return 当前实例化的bean ， 下面的步骤不再执行

如果项目中定义了自己的 BeanPostProcessor 。那顺序是在

CommonAnnotationBeanPostProcessor

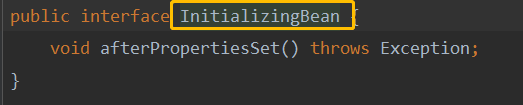
AutowiredAnnotationBeanPostProcessor 之前 ，如果自己定义的返回null则

后面的BeanPostProcessor 则不会执行



**J、**初始化

初始化的过程也可以干涉 实现 Initiazilingbean 接口 ，同理拿到所有实现了 InitzilitionBean所有的实现类 ，然后执行 afterPropertiesSet()方法



**K、**初始后 拿到实现了 BeanPostProcessor的类然后遍历去执行afterInit方法

**(9)BeanDefinition后置处理器**

**(10)填充属性**

1. Spring初始化时是从第一步开始扫描开始

当项目启动后 getBean(beanName) 是从第5步开始的 通过 beanName从单例池中获取响应的对象

1. @Primary 、 @Qualifiler 、 @Inject
   1. @Primary 当一个接口被多个类实现依赖注入时 ，spring在初始化时并不知道该注入哪一个实现类因此会报错， 但是实现类可以使用@Primary 注解表示可以被优先注入
   2. @Qualifiler 当一个接口被多个类实现依赖注入时 ，spring在初始化时并不知道该注入哪一个实现类因此会报错，但是实现类可以使用@Qualifiler 手动指明使用哪个类
   3. @Inject 是java提供的依赖注入的注解 ， 如果出现了 上述一个接口有多个实现类，需要用到某个 可以配合 @Named

.