**Spring**

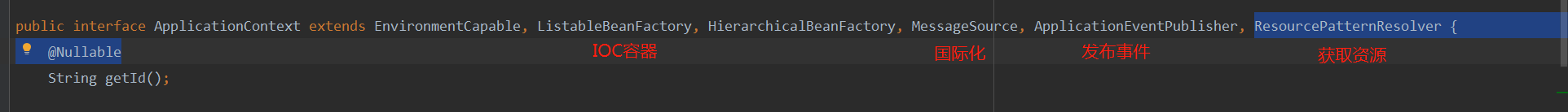
1. 什么是IOC ：控制反转？

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. 获取 beanFactory的方式有哪些
   1. Aware

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. Bean 工厂的后置处理器

BeanDefiniitonRegisterPostProcessor

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 默认单例
   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

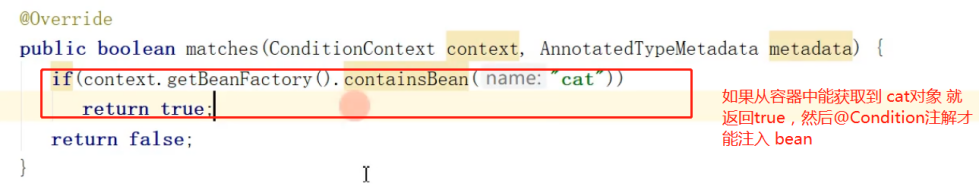
selectImport()方法会返回一个String数组，数据中是 全限定类名。 会将返回的这 些全限定类名加载至spring 容器中 。 springbo的自动配置的 ImportBeanDefenitionRegister 类就是通过这个接口实现的

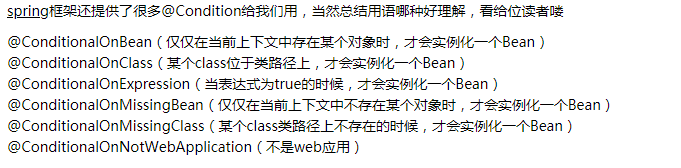
由于数据的特性是有序的 ，对象所以加入到 spring容器中也是有序的

* 1. 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. 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

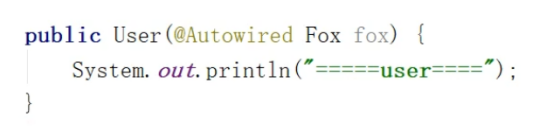
（6）webSocket

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 。



标注在 属性上

标注在 方法上

标注在 参数上

@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 （循环依赖）

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@Bean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

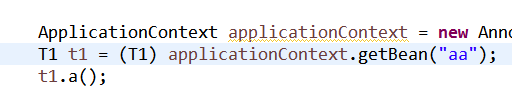
(1)BeanFactory： spring（ApplicationContext）容器的顶级接口 是产生bean 的工厂

(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**)

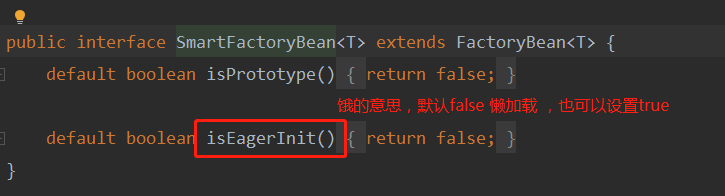
* + 1. 如果通过容器获取 实现FactoryBean 的bean时 ， 就通过 getBean(“&xxx”)



* + 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

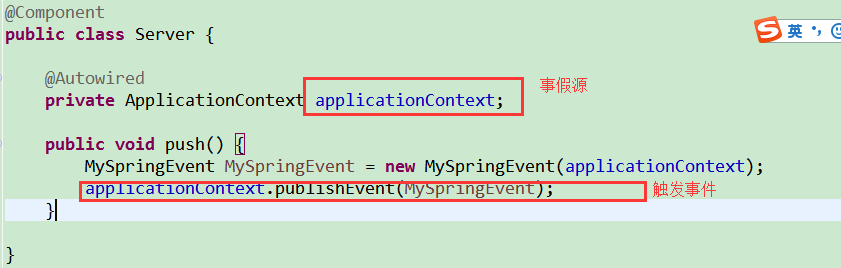


1. Spring中的事件 (观察者模式)
   1. 观察者模式包含3个部分
2. 事件 EventObject
3. 事件源 Source
4. 监听 @EventListener （当事件源发生变化时，触发监听）
   1. 在spring中如何定义事件 ： 实现ApplicationEvent ，ApplicationEvent 内部实现了 java 的EventObject 接口
   2. 在spring中如何定义监听 ：实现 ApplicationListener ，ApplicationListener内部实现了 java 的 EventListener

或者通过 事件监听注解@EventListener

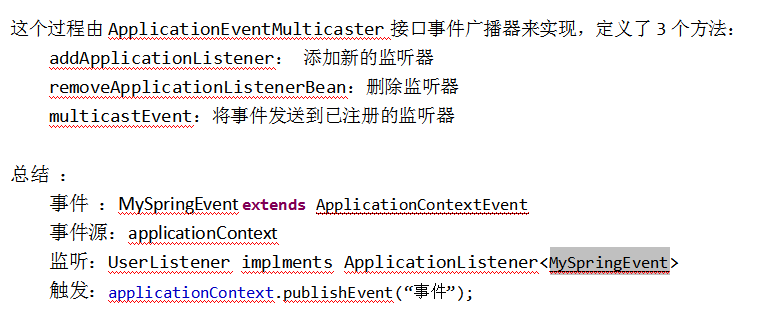
* 1. 如何发布事件来触发监听，实现ApplicationEvent的事件一pulish，就会触发监听。 在spring容器初始化时 会将 ApplicationContext对象注入容器

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 fastjson

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 ，实例化文件中 （handlerAdapter、视图解析器等） 然后扫描Controller 将@RequestMapping中配置的url 跟controller进行绑定保存至 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 MATE-INFO目录下的 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

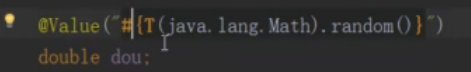
* 1. 读取yml中配置的字符串 可以直接映射成 一个类

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)

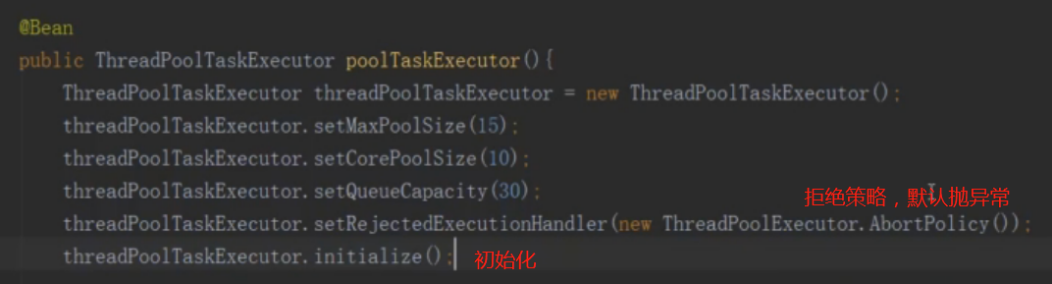
@ConditionOnBean作用 ： spring容器中须依赖 xxx.class 类，如果没有不能执行该配置文件。而自定义的注解实现的功能是注了的 xxx.class 。

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springMVC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



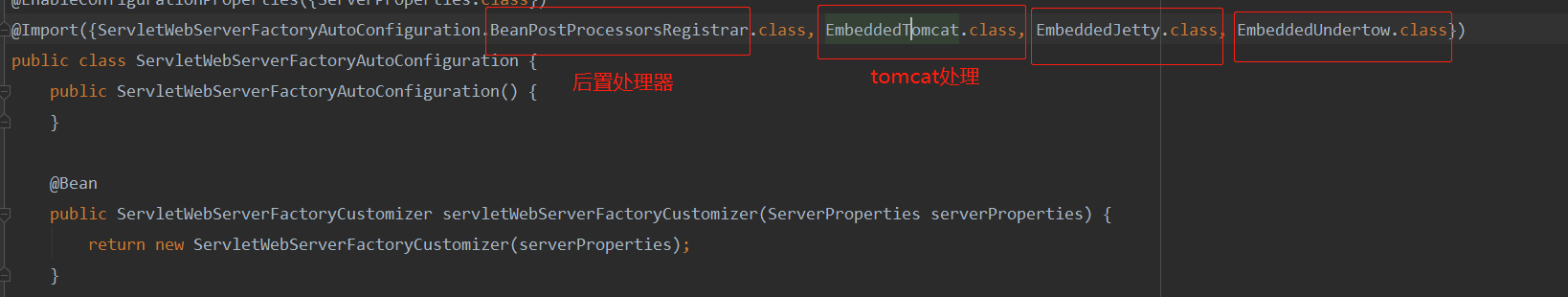
1. Springboot 默认的 web 容器 tomcat , 也可以自定义其他的 web容器
   1. Tomcat
   2. Netty
   3. Jetty 等
2. springBoot如何自动配置tomcat

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，

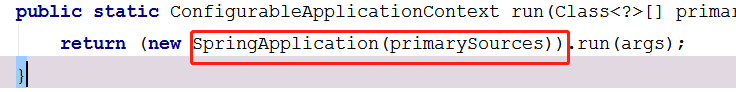


**注意 ：** @import一次性注入4个.class数组 ，那么在放入spring容器初始化时，他们也是有顺序的， 首先会加载 tomcat容器 ，**有tomcat容器就不再加载其他容器。 tomcat容器没有 再加载其他类型的容器** 。 因此想给springboot自定义 其他容器 必须删除tomcat内置的依赖，让springboot加载其他的依赖

1. 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
2. Springboot 热部署原理

通过自定义类加载器 动态的加载class文件

1. Springboot 启动流程 包括两部分
   1. @SpringbootApplication 中自动配置 负责将必要的类进行加载 交给spring管理
   2. Main方法的启动 .run() 启动
2. 内部 new SpringApplication() 对象 .run(args)



1. 判断当前项目是什么项目

主要是 判断当前项目中 是否引入必要的依赖 如 webflex、springmvc 、java

如何判断 ： 看项目中是否有必要的 类 ，如判断 是否是springmvc项目，就查找 DispathcerServlet 类是否存在

1. 创建 springboot的 计时器 帮助类 ，用于记录方法执行的时间 。在springboot启动时控制台时间输出就是这个对象输出的

StopWatch stopWatch = new StopWatch ();

1. 获取 环境配置，其中包括 application.yml ，如果没有则自己创建
2. 初始化 banner logo
3. 创建 springApplication 对象 根据 当前是什么项目来初始化改对象

如果是springmvc 就初始化 ：AnnotationConfigServletWebServerApplicaitonContext

如果是java环境 就初始化 ：AnnotationConfigApplicaitonContext

1. 接着初始化 spring 的容器 并进行bean 的初始化 ， 在spring容器初始化进 行refresh时 ， 重写onRefresh方法 判断当前项目的 web容器 是否是内嵌 的
2. Spring中的 核心类概念
3. 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 ，

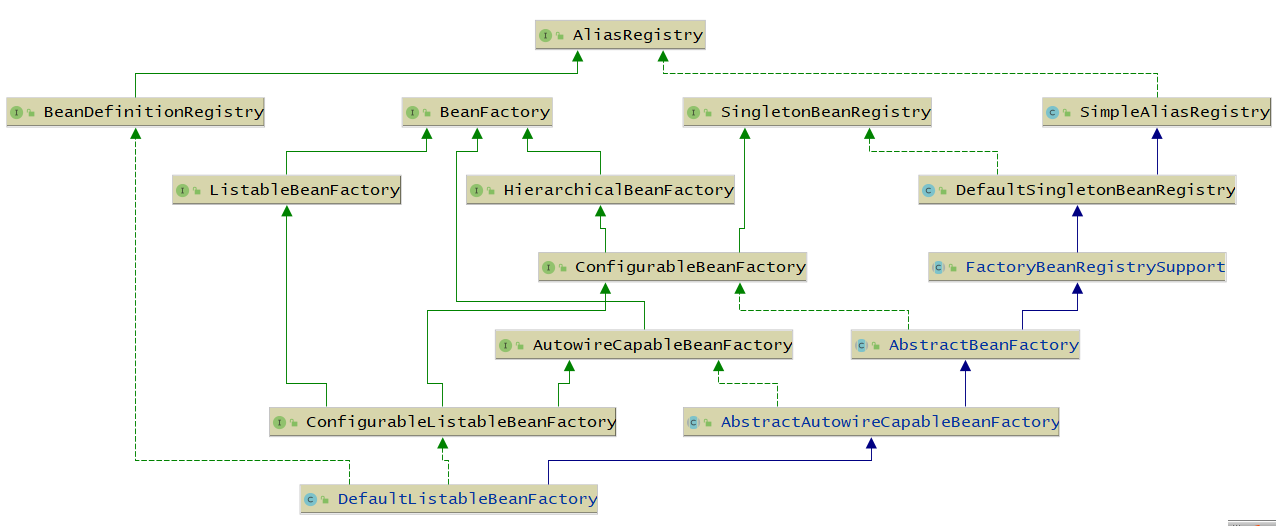
判断calss上的注解是否需要当前class交给spring管理，如果需要将交给reader对象 ，将class对象实例化成 bd对象

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**

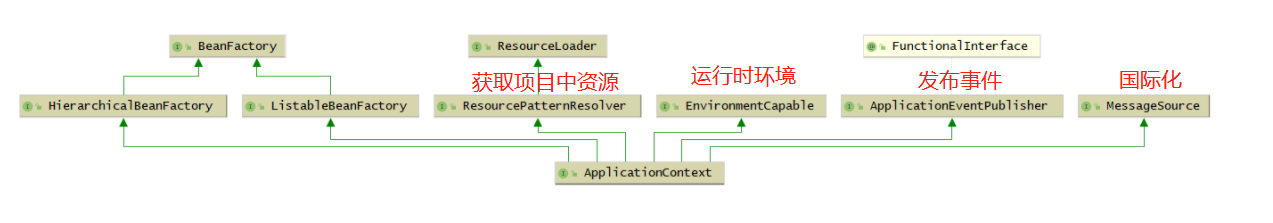


**总结：** **DefaultListableBeanFactory具有的能力**

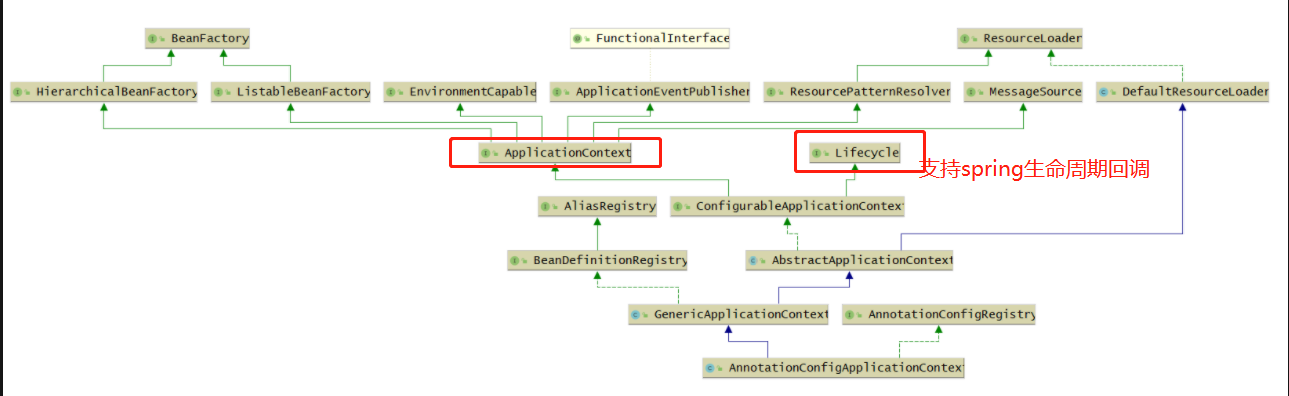
1. **操作BeanDefenition (注册、获取、移除)**
2. **操作bean的别名**
3. **操作bean** 
   1. **具有beanFactory 获取bean ， 获取父容器bean、销毁bean**
   2. **操作单例bean (注册、获取、移除)**
   3. **Autowried对bean自动装配**
   4. **支持bean后置处理器**
4. **register 将对象 交给spring容器管理 @Import（xxx.class）**

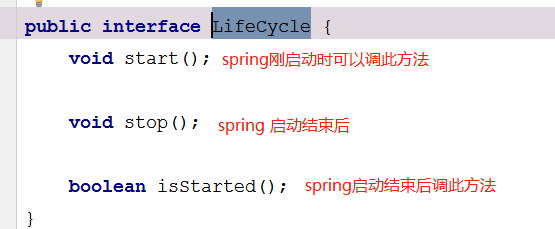
**将对象实例化成bean 或者 bd**

1. ApplicationContext

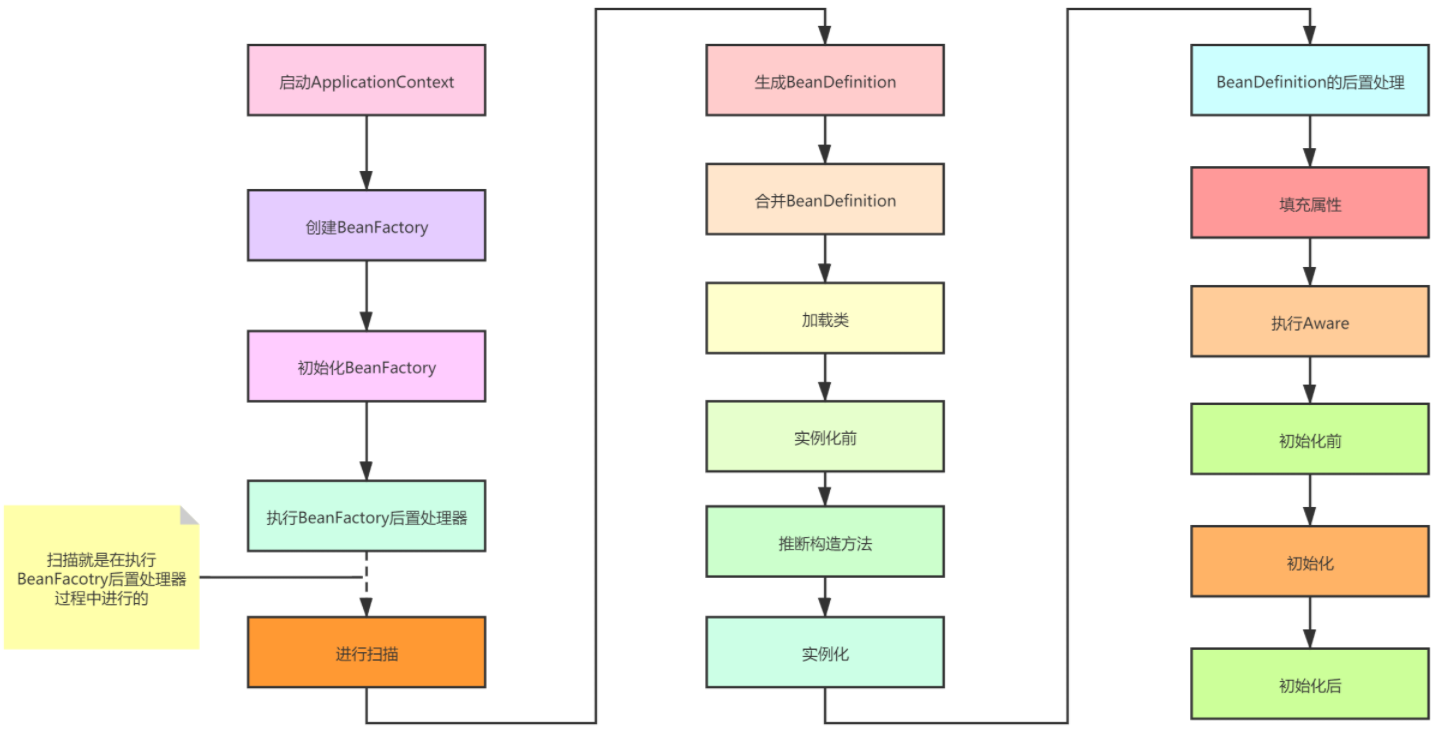


1. HierarchicalBeanFactory 用于 BeanFactory 的功能
2. 拥有获取 BeanNames的功能
3. AnnotationConfigApplicationContext





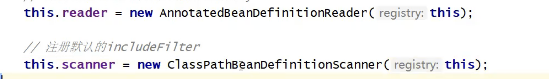
1. 真正创建出来的单例bean ，是放在单例池中 通过BeanFactory.getBean() 也是从单 例池获取中 获取bean
2. **Spring生命周期**



1. **启动AnnotationConfigApplicationContext 加载@Configguratioin 配置文件**

**1)进行初始化容器 ， 在初始化容器时会初始化AnnotationConfigApplicationContxt 的父类 ，在父类中创建DefaultlistableBeanFactory对象(即bean工厂对象)**

**2) AnnotationConfigApplicationContxt再调用自己的构造方法**

**对 reder 和 scanner对象进行一次初始化** 

Reader初始化时 ：

1. 初始化spring 的比较器AnnotationAwareOrderComarator 比较器，用于为集合排序作准备
2. 初始化ContextAnnotationCandidateResolver依赖注入解析器
3. 将 初始化最开始的5 BD

**接着初始化最开始的5个BD 存放至 BDMap ，其中包含2个 BeanPostProcessor 后置处理器 + 3 个bean 工厂的后置处理器**

* 1. **.初始化 ConfigurationClassPostProcessor配置类后执处理器 用于解析配 置文件以及扫描calss类ConfigurationClassPostProcessor然后被实例化成BD， 放到 BDmap中**
  2. **初始化 AutowiedAnnotationBeanPostProcessor 属性填充后置处理器被 实例化成BD ，放到 BDmap中**
  3. **初始化CommonAnnotationBeanPostProcessor 属性填充后置处理器被实 例化成BD ，放到 BDmap中**
  4. **初始化EventListnerMethodProcessor 用于处理事件的的后置处理器 ， 被实例成BD，放到 BDmap中**
  5. **初始化DefaultEventListnerProcessor 用于处理事件的的后置处理器 ，被 实例成BD，放到 BDmap中 。 和上一个后置处理器搭配使用的**

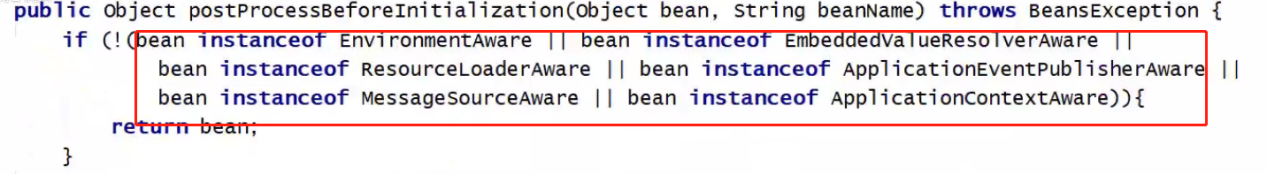
Scanner初始化时：

1. 将项目中 配置类 实例化成 BD
2. 初始化AnnotationTypeFilter 对象 ， 这个对象用于后面 扫描 calss类上的注解 ，来判断当前被扫描的类需不需要注入

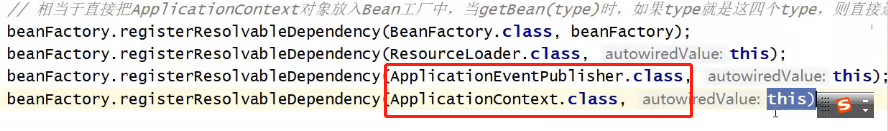
1. **接着调用 refresh() 方法**
2. **prepareRefresh()**
3. 记录 当前 refresh时间
4. 设置 spring 开启标识
5. 以及对环境变量的验证
6. **获取 开始创建的 DefaultListableBeanFactory 容器 包含之前始初始化 的一些资源 6 个bd(5个默认的 + 项目配置的的bd)、比较器、依赖注 入解析器(此时 单例池为 空 )**



1. **准备 beanFactory** 
2. 初始化 calssloader 类加载器
3. 设置 spring spel 表达式的解析器 #value (“$”) 使用
4. **添加 bean 的后置处理器 ApplicationCoontextAwareProcessor，到 spring容器 执行 各种awre 的后置处理器**



1. 添加 spring 的事件发布对ApplicationContextPublisher 到spring容器
2. 添加 spring 的资源管理对象 到spring容器
3. 添加spring 的ApplicationContext对象 到spring容器



1. 往单例池中 添加最开始的 3 个单例 bean

environment、systemproperties、systemEnvironment

1. **执行将之前添加的 bean工厂的后置处理器扫描 项目中定义的bean**



**包括 ：**

a.拿到 BDmap 6个中的 3 个bean工厂的后置处理器

ConfigurationClassPostProcessor扫描项目中的calss文件，实例化成bd

**扫描过程 ：**

1. **.扫描加了 @Compoent 注解的类 和 @Bnea 注解的方法**
2. **获取 项目中 自定义bean的后置处理器(项目中的扩展点) + BDmap中开 始创建的3个bean的后置处理器**

**循环这些后置处理器 去进行分类 并排序 ，**

**分类：因为后置处理器有很多种 （分类是通过接口进行分类）**

**排序：因为后置处理器有很多 ， 执行的时机也很多 。 同一个执 行时机不同的后置处理器也有先后顺序 使用@Order注解**

**（排序是通过spring 的**AnnotationAwareCompatator排序 **）**

**分类、排完序后 ，将这些后置处理器添加至 beanFactory的集合中**



@order 的值越小 在list中的瞬息越靠前 ，执行时机也越靠前

@Vaule 注解 的 @Order值默认是最大值

@Autowried 注解 的 @Order值默认是最大值 - 1 ，虽然 @Autowried 的order值比@Vaule的小 ，但是 @Autowried 是优先级更低的bean 后置处理器分类。 @Vaule 后置处理器的优先级分类比@Autowried 后置处理器优先级分类要高

1. **初始化项目中 @Bena 注入的 国际化对象**

**MessageSource对象， 将值赋值给 AppplicationContext的 messageSource属性**

1. **初始化项目中的 事件监听器Listener**
2. **实例化 非 懒加载的单例bean ， 把bean创建出来 存放至单例池**
3. **执行 spring 容器的生命周期 Lifecycle 接口**

**H、发布CntextRefreshEvent事件 执行事业务逻辑**

1. **扫描Bean 为创建bean做准备** 
   * 1. 扫描 实例化 BD

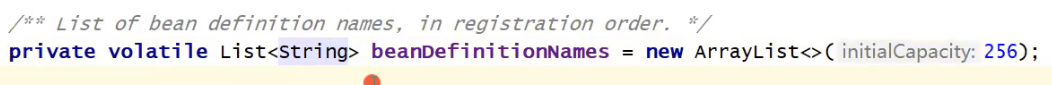
Spring 通过asm 字节码扫描技术 ，将标注 @Compotent相关注解的类加载 至内存 生成BeanDfinition对象（包括普通bean 和 抽象bean）。

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



F**or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** 。

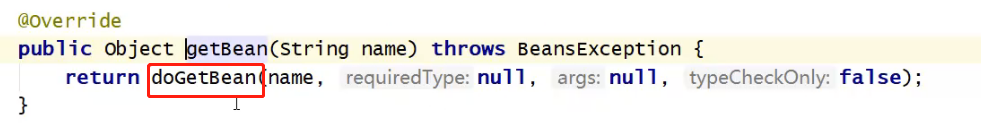
* + 1. 判断 bd是否是 smartFactoryBean(因为smartFactoryBean有实现恶汉的功能)

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)**

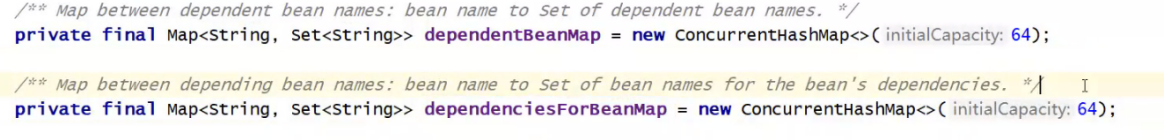
1. 先通过 beanName 到单例池缓冲池中找是否存在
   1. 存在 返回
   2. 不存在
      1. 单例池中没有 ，通过beanName 查找 父BeanFactory容器中是否存在

(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



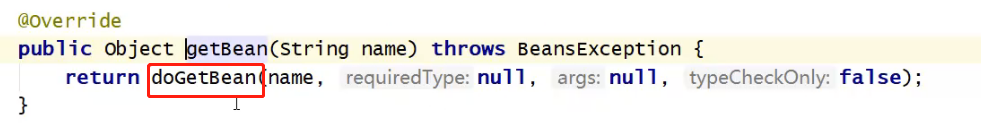
1. 如果存在 @DependsOn(“xxx”) 注解

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

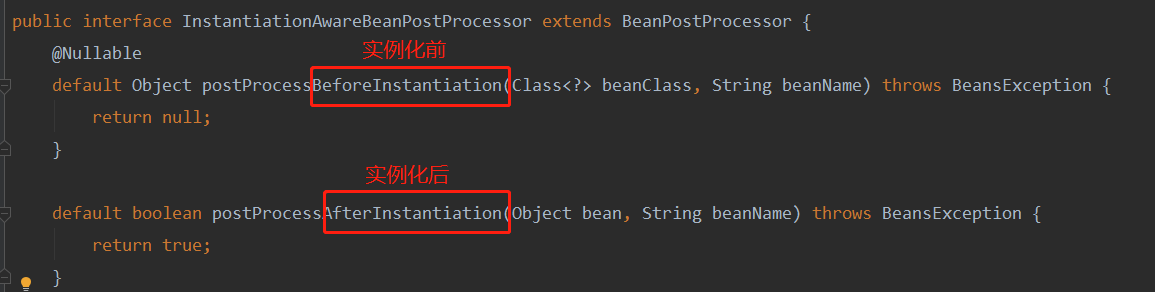
参数3： args

**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)



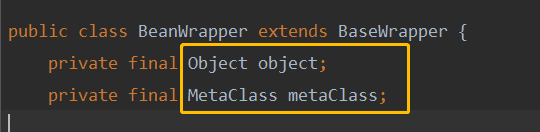
**注意** ： 实例化前默认的方法是 为null 的，如果从写此方法 返回实例

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 。



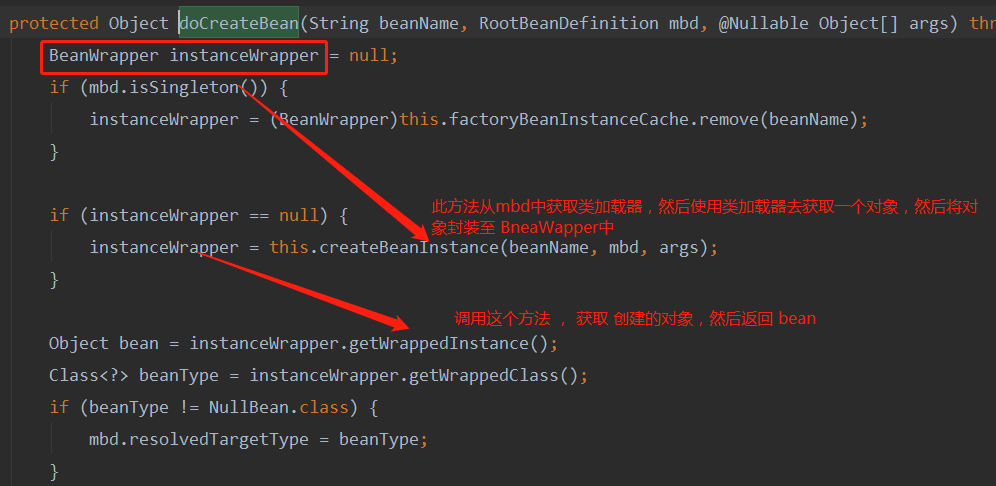
**C、**实例化

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

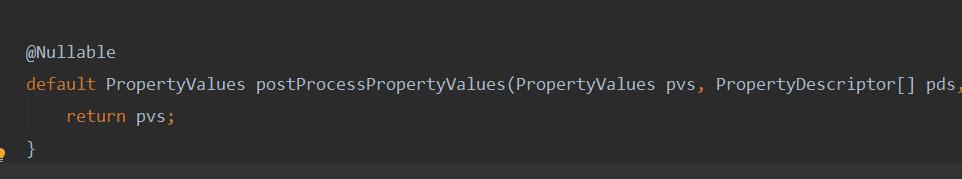
**E、**执行 BeanDefinition的后置处理器 **BeanDefinitionRegisterPostProcessor** (因为对象只是被实例化后面还要对bean属性进行填装、以及对象的初始化 ，在这个阶段是可以对 bd进行扩展的。 如：修改bean 的scope l、azy等 如 mybaties注册新的BD )

**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

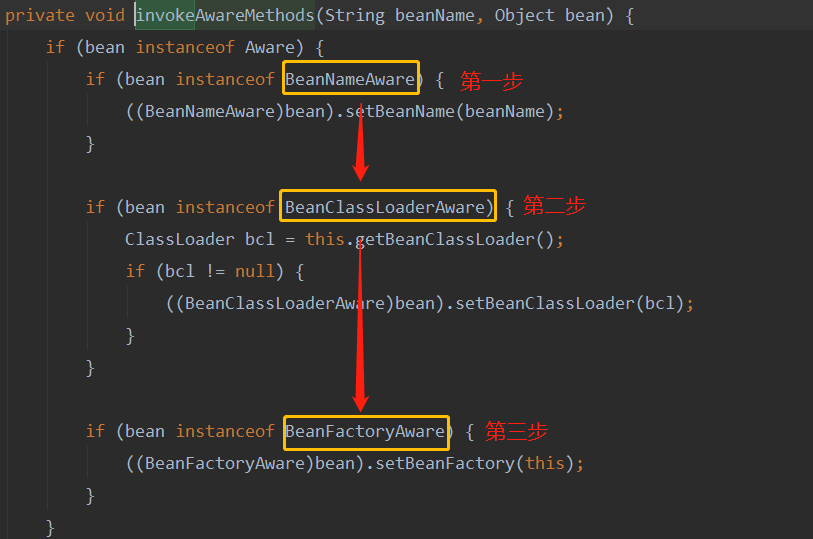
注意 ： spring 是不会为代理对象进行属性注入的 ，只会对原始对象进行属 性注入

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



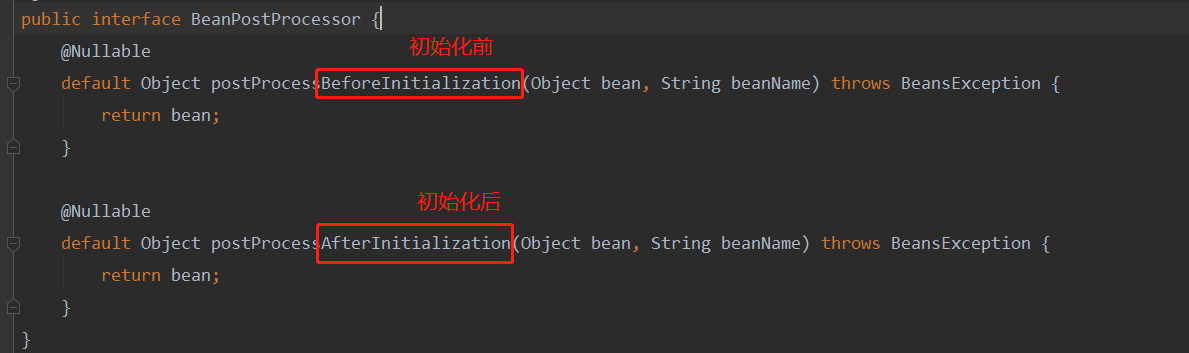
**H、**执行bean 级别的aware接口 并且有顺序

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 )

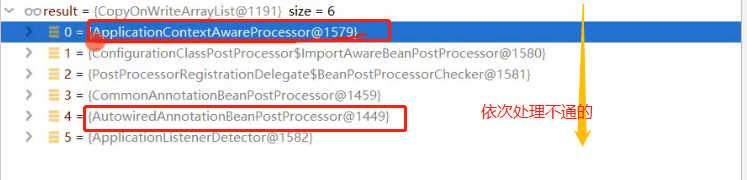


另外还有 ApplicationContextAware 也可以获取spring 的context容器来执行更多的功能。但是ApplicationContextAware 执行的时机不再此处 。他 以及还有其他的aware是在工厂初始化时执行的 ， bean工厂级别的aware会在工厂初始化时先执行

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



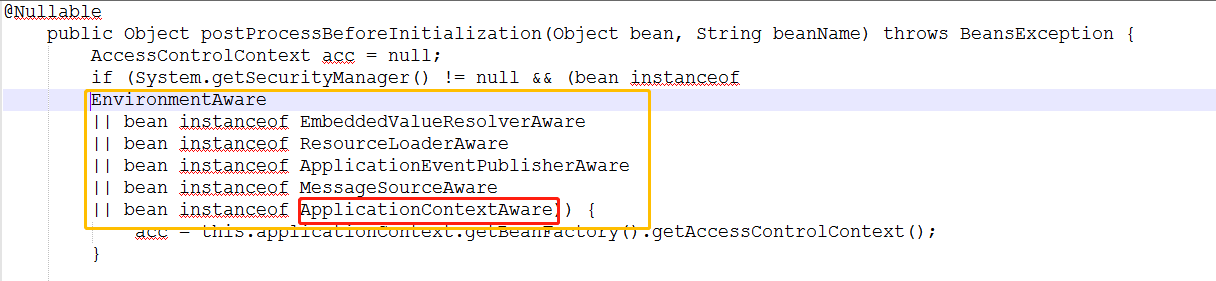
这个接口会被很多的类实现 ，然后会依次执行



**@PostConstract ：** CommonAnnotationBeanPostProcessor

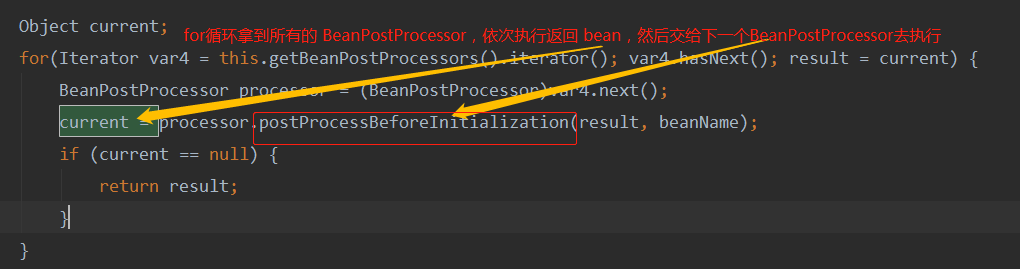
**@Autowired ：** AutowiredAnnotationBeanPostProcessor

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;

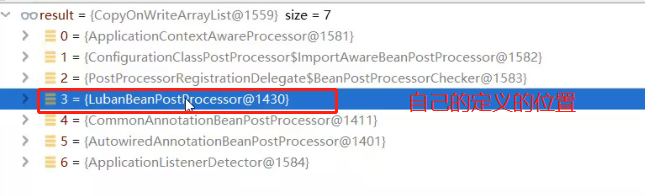
那么将直接return 当前实例化的bean ， 下面的步骤不再执行

如果项目中定义了自己的 BeanPostProcessor 。那顺序是在

CommonAnnotationBeanPostProcessor

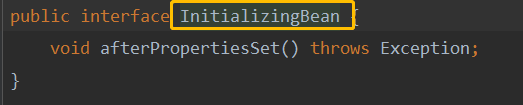
AutowiredAnnotationBeanPostProcessor 之前 ，如果自己定义的返回null则

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**

**使用后置处理器完成依赖注入 ：** AutowiredAnnotationBeanPostProcessor

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
2. spring在通过后置处理器 ConfiguratioinClassPostprocessor 扫描class文件时 就是去扫 描 一系列的注解
3. 在一个 普通的类 这个类上没有@Configuration 、@Compent ， 这个类 中定义一个 @Bean ，这个@Bean 是可以被加载到spring容器中的
4. 如何对list 进行排序
   1. 可以使用 stream
   2. 可以使用 Comptor接口
   3. 可以使 spring 中的 AnnotationAwareCompatator 类 （比较方便）

AnnotationAwareCompatator 这个类在sping中专门用于对集合进行排序的 ，

集合中的对象标注 @Order注解 ，然后AnnotationAwareCompatator 根据@order对集合重排序

1. Aware 的理解

Aware 可以分为 工厂级别的aware 、 bean 级别的aware

工厂级别的aware ： 会在beanFactory 初始化是执行

Bean级别aware： 会在 bean 实例化时 执行

1. BeanFactoryPostProcessor 工厂后置处理器 有2个接口
   1. BeanFactoryPostProcessor
   2. BeanDefinitionRegisterPostProcessor 带注册功能的 工厂后置处理器
2. BeanPostProcessor bean后置处理器 有4个接口
3. 注册 bd 的方式总结
   1. ImportBeanDefinitionRegister 接口 搭配@Import注解
   2. BeanDefinitionRegisterPostProcessor 后置处理器
4. 项目中重写 bean的后置处理器进行扩展时时 ， 方法不要返回null 。 因为bean后置处理器时链式调用 如果前一个后置处理器return null ， 那么后面的后置处理器将不再执行。

如自定义的后置处理在 AutowiredAnnotaionBeanPostProcessor、 ComoonAnnotaionBeanPostProcessor之后执行， 如果自定的返回null 就不会进行属 性填充

**Spring循环依赖**

1. .什么是循环依赖

对象 AB 相互依赖 ，在创建A对象时 依赖B， B对象创建时又依赖A

1. Spring 如何解决的？ 通过spring的三级缓存  
    spring中的缓存
   1. 一级缓存 CurrenthashMap ： 单例池
   2. 二级缓存 HashMap： 存放 是循环依赖的bean的**代理对象 和 没被代理的实例化对象**

**普通bean 的代理对象的创建和循环依赖bean的代理对象的创建的时机是不同的**

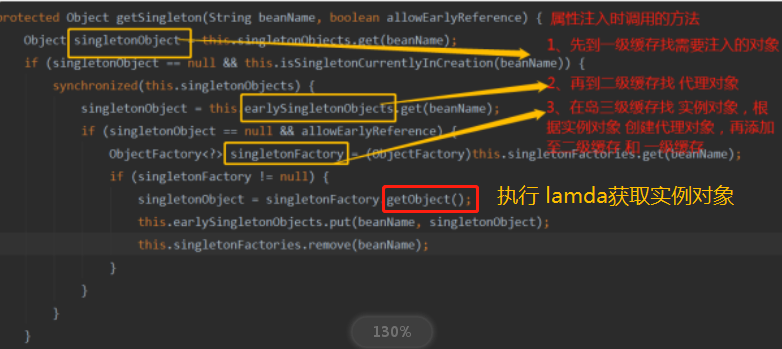
1. . 普通bean ： 代理对象的创建时机 在属性填充后的初始化对象时
2. . 循环依赖bean ： 代理对象的创建在 当前A Bean 属性填充 循环依赖另 外一个B bean ，然后B bean 又需要填充A Bean**时 (创建代理对象的时机提前了)**
3. 先从 一级缓冲中找Abean ， 由于循环依赖 Abean 正在创建中
4. 然后到二级缓冲中找 被代理的对象 ，此时还未创建 所以还没有。
5. 然后到三级缓存中获取 Abean 的**实例化**对象 ，在判断 A Bean 是否需要被代理

c1、发现A Bean需要代理 ，然后创建A bean的代理对象， 并将A对象的代 理对象赋值给B Bean 的属性 。接着将代理对象存放 至 二级缓存 。

c2、发现A Bean不需要代理 ，直接 将A对象的代理对象赋 值给B Bean 的属性 。接着将实例对象（实例对象时是调用构造器new）存放至二级缓存

D、假如后面A bean 跟C Bean又存在循环依赖的关系，在A bean 填充C bean 时 ，回去创 建C Bean ，C Bean 再去属性填充， 判断A bean 需要代理 ，

然后先回去一级缓存拿没有 ，再去二级缓存拿 ，二级缓存有 然后再返回 。



**当A bea 填充完属性 ，初始化结束之后 就从二级缓存中取出代理对象或者实例 对象 存放至 单例池中(一级缓存)**

填充属性时，创建另外一个对象 A bean的代理对象就已经被创建了  
 如果A Bean不是循环依赖 ， 则创建的时机是在 初始化对象之后

* 1. 三级缓存 HashMap ： 存放 bean刚实例化时的对象 也就是刚通过构造器new出来的对象 ，属性并未赋值。 三级缓存 真正存的是 lamda表达式 key是beanName ， 这个lamda表达式 的功能是 用于创建一个实例化的bean 。

循环依赖从三级缓存找 实例对象， 其实就是执行这个lamda 然后返回一个实 例对象 。 当从 三级缓存中获取对象 ，创建完 代理对象并将对象存放至二级缓 存后 ， 会将三级缓存中的 lamda表达式进行移除 和 二级缓存中的代理对象移 除. （为什么移除 ？ 因为是 HashMap）

1. **为什么循环依赖时 创建代理对象的时机提前了 ?**

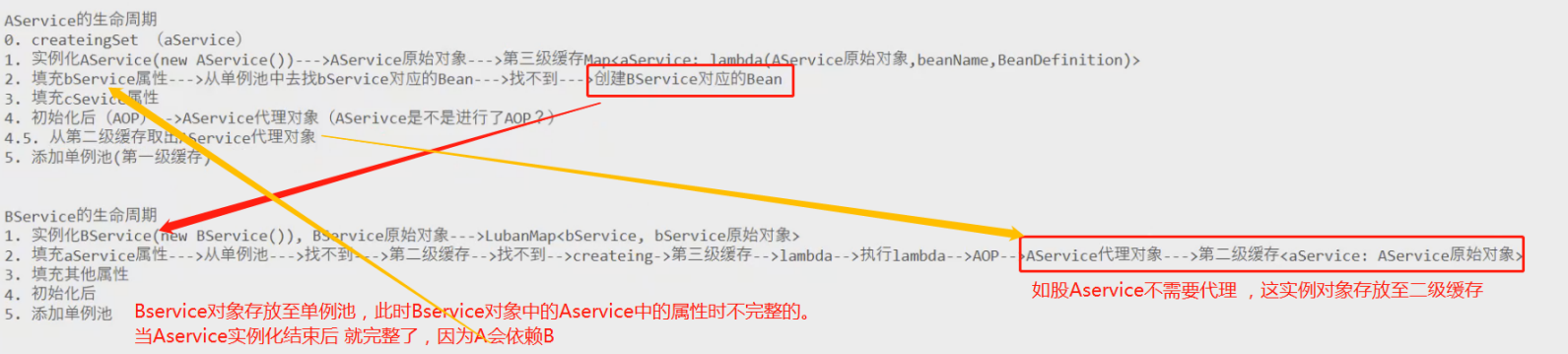
如果跟原来的时机相同 ，那么 其他对象循环依赖属性填充时 填充的对象是 刚实例 化的对象，当前对象还没额外的附件功能

1. 二级缓存的作用 ：用于解决循环依赖的问题时才被使用

二级缓存用于存放 循环依赖bean 的代理对象 和 实例对象，当一个bean 与多个bean都有循环依 赖的关系 ， 只需创建一次bean的代理对象 ，存放至map中 ，下次需要用到不需要 重新创建 ，直接到map中取

1. 如果 一个 对象被代理 ，那么代理对象最终会被存放至单例池
2. 循环依赖 bean 实例化过程总结

如果 是循环依赖 且 需要aop ， 那么在第4步初始化后就 不需要aop 了



1. 循环依赖都是 原型的 @Scope(“prototype”)是不可以的

依赖时每次创建都是新创建对象， 逻辑上都走不通 ， 再加上 每次新创建 从缓存中 也拿不到被实例的对象

**Spring AOP**

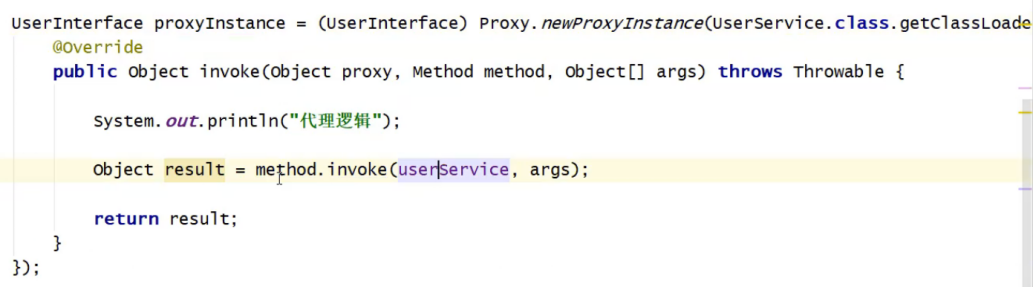
1. 动态代理的 3个核心
   1. 目标对象
   2. 代理对象
   3. 代理逻辑
2. JDK 实现动态代理 ：

(1)基于接口 ， 实现 InvocationHandler 接口 ， 实现Invoke方法

Invoke方法 是代理逻辑

(2)Proxy.newProxyInstance创建代理对象,依赖目标对象。当代理对象执行方法时 ，就会走代理逻辑

**注意 ：** 由于 jdk动态代理是基于接口的 ， 代理对象 必须使用接口接收



1. Spring 使用 aop的方式有3种
   1. 手动编程的方式 ， 使用 ProxyFactory 类进行 aop 。在spring初始化后会进行aop
   2. 使用AbstractAutoProxyCreator 进行 aop

就是在 xml 中或配置类中注入 AbstractAutoProxyCreator 的实现类，然后再设 置类的pointCut 。

这个类在实现了 spring 的bean的后置处理器 BeanPostProcessor 在spring 初始 化后 ， 根据配置的 pointCut**找到advisor Bean (PointcutAdvisor)** 根据不同的 advice和pointcut 去进行aop

* 1. 使用注解的方式 @EnablAspectJAutoProxy进行aop

@EnablAspectJAutoProxy 内部通过 @Import注入一个 类 ，这个类也是 AbstractAutoProxyCreator 同理也是在初始化后进行aop代理

在初始化时不仅 找**找到advisor Bean** 还会找到 **标注 @AspectJ 的类 。** 然后解 析@AspectJ 类中的注解， 并将注解 转成 advisor 。 然后根据advisor 中不同的 advice去 进行代理

1. Spring的 AOP 使用 **ProxyFactory** 抽象了 jdk 动态代理 和 cglib 动态代理

引入了 切面、切点、连接点、通知的概念

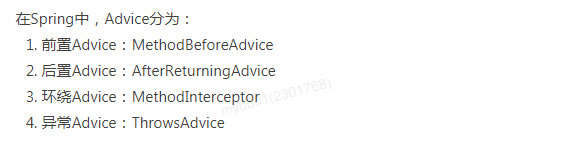
**切面 aspect ：** 就是AOP 面向切面

**切点 pointCut：** 表示通知可以作用在 哪些方法、注解上或者包含关键的方法上， spring中使用表达式实现连接点

**通知 advice** ： 代理逻辑 ,由于这个逻辑可以作用在目标方法的 前后

因此出现了 前置、后置、环绕通知 、

Davice接口不同的通知分别对应不同的实现类。 并且也有不同 的注解实现类。



@Before对应的是AspectJMethodBeforeAdvice，

@After对应的是AspectJAfterAdvice

@Around对应的是AspectJAroundAdvice

@AfterThrowing对应的是AspectJAfterThrowingAdvice

@AfterReturning对应的是AspectJAfterReturningAdvice

**连接点 Advisor:** 切面中的代理方法 （连接点 + 通知）

1. **ProxyFactory**
2. 添加目标对象 ProxyFactory.setTarget(初始化后的bean)

如果 初始化后的bean实现了接口，那么spring aop会使用 jdk动态代理

初始化后的bean没有实现接口或者继承了类 那么spring aop会使用 cglib动态 代理

其实接口也可以用cglib 需要

1. 通过注解 @EnableAspectJAutoProxy(ProxyTargetClass = true)
2. ProxyFactory.setProxyTargetClass(true)

来关闭jdk动态代理 ，但是如果使用 jdk动态代理 目标类必须实现接口

1. 生成代理对象 ProxyFactory .getProxy

在 getProxy() 获取代理对象时会进行代理 ，在代理之前会 判断是用 jdk动态 代理还是cglib动态代理 。 这个主要是跟根据目标对象来判断的

1. 添加 代理逻辑ProxyFactory .addAdvice()
2. Spring 初始化Aop的时机 ，在 对象初始化之后 如何执行的 ？
3. 创建 ProxyFactory 对象
4. 将初始化后的对象作为 目标对象 设置到 ProxyFactory 中
5. 判断选择哪种方式来实现 aop

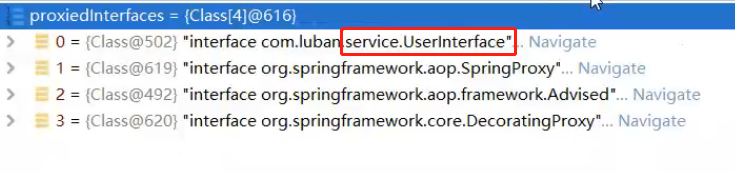
默认 jdk 动态代理不关闭的情况下， 如果bean实现了接口 就使用 jdk动态代理

如果 bean 没有实现接口 或者 继承类 则使用 cglib动态代理

* + 1. Jdk 动态代理 JDKDymicAopProxy implents InvocationHandler {}

1. 先获取bean 的接口封装类，包括自己实现的接口 + aop自带的接口 。

这样的话 代理对象 可以被其中的 任一一个接口 接收



1. 然后直接 走 Proxy.newProxyInstance(接口) 代理逻辑,调用 invoke方法
   1. 获取 目标对象
   2. 获取通知 (代理逻辑) 判断通知的实现类是 前置通知、后置通知、 还是环绕通知
   3. 执行目标对象的方法 和 通知
      1. Cglib 动态代理
2. Spring 注解的方式使用 AOP :

通过 @ EnableAspectJAutoProxy 注解 ， 直接内部定了 register类 ，这个类实现了 ImportBeanDefinitionRegister 的bd注册器

**两个属性 ：**

1. . proxTargetClass 设置 是否关闭 jdk 动态代理 默认是不关闭的
2. . exposeProxy 设置代理逻辑中如果有当前类的方法调用， 想让当前类的 方法被调用时 也使用代理对象 ，可以设置此属性为true
3. Cglib 和 jdk 代理的比较
   1. . 一个基于接口 一个 基于类
   2. . cglib功能较强大，可以设置 exposeProxy 参数

如：被代理的 A 方法 ， 调用B 方法。 默认B方法在被调用时 是实例对象调 用的。但是在调用B方法时 有时期望是被 代理类调用 可以通过 Cglib来实现。

如事物的传播跟这个属性就有关系 。 一个事物方法中 调用另一个事物方法如果 不指定传播行为 ， 事物就会失效

**实现原理 ：**Cglib在 创建完代理对象后 ，会将当前对象放入 threadLoacl中 ， 在 执行代理逻辑时 ，会取出代理对象 ，执行方法 。

**此种效果也可以使用 另一种方式实现** ： 直接在当前 Bean中 注入当前Bean

因为@Auwtried注入的bean 是从单例池拿的。 单例池中存放的是代理对象

1. Spring 为一个类的方法进行了代理 如权限代理 ， 但是这个方法上又加了@Tranction 事务注解， 那么最终生成的代理对象存放至 单例池中只有一个 ，而且是同一个
2. **总结 ：**
3. .3种 aop 的执行其实都是 由 后置处理器来调用的 ，都在初始化之后。 代 理对象会存放至单例池
4. . 代理对象中的 属性是没有值的 null ， 主要代理的是方法 。 如果使用代理对 象的属性调用方法 会报 NullPointException

**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 事务操作 数据的流程

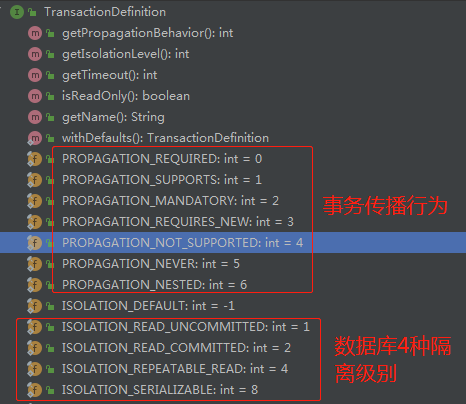
数据库在执行 sql 语句的流程是 ，开始事务**--->**执行**--->** 执行sql浴巾**---->** commit;提交

只不过 开启事务 和 提交是默认开启的。 但是在 spring在操作事务时

先拿到连接 **---->**事务设置为 手动提交**--->** 执行sql 语句**--->**提交

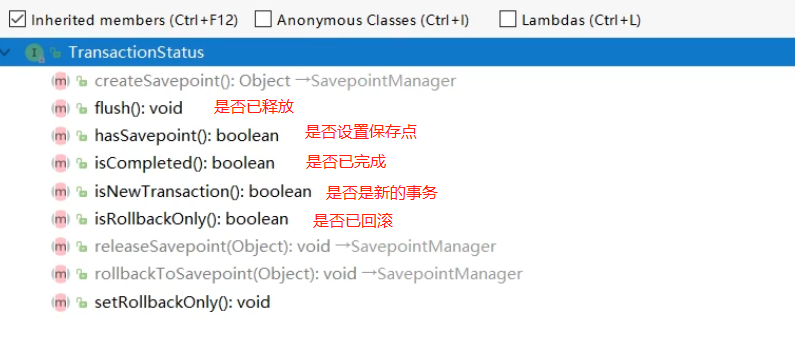
或者 如果 出现异常就会滚

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



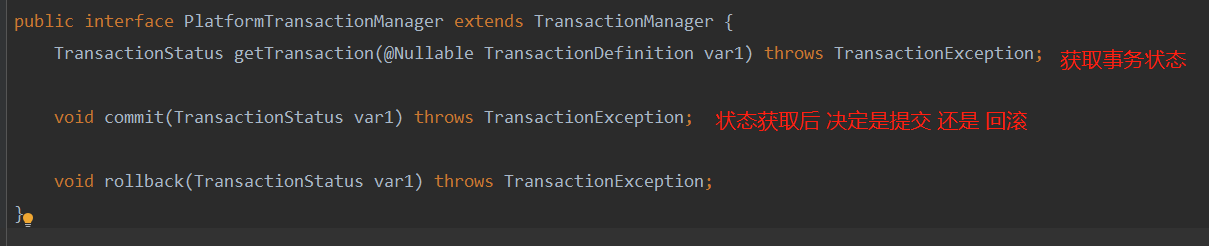
* 1. TransactionAttribute 接口 ： 扩展了TransactionDefinition 扩展了 事务回滚功能
  2. PlatformTranscationManager接口 ： 事务管理器包括

1. 获取事务执行状态 TransactionStatus

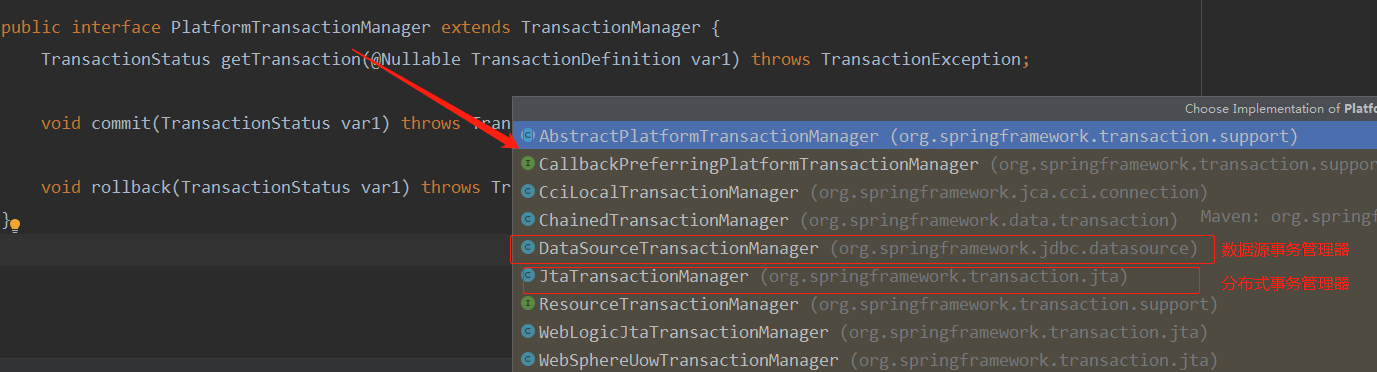


事务保存点的作用 ： 一个事务当对表进行多次操作时 ， 如果发生错误需要 回滚， 不是全部回滚 而是回滚到指定的保存点。 保存点之前的操作时持久 化的

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 ？**

**因为已经不是同一个事物了**

**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注解本质就是创建代理对象

先连接--> 再开始手动提交 --> 再去执行代理逻辑(将spring创建的连接交给

mybaties去执行)

**再开启事物时Mybaties 是如何拿到spring创建的连接的？**

通过SqlSessionFactoryBean 中的事务工厂属性 ， 调用事物时 会将spring创建的 事物管理器 交给 这个事物工厂 去获取连接datasource

**注意 ：**代理对象主要代理的是方法逻辑。 方法被代理时去调实例对象的方法执行 业务。

假如被代理的实例对象中有 mapper属性 ， 这个属性在

运行时执行mappe.insert()是被目标对象调用的 。 代理对象中的 mapper 属性是 null；

假如直接通过 代理对象调mappe.insert() 那么会报 nullPointException



RuntimeException 和 Error

[ˌprɒpə'ɡeɪʃ(ə)n] 、 [ˌaɪsəˈleɪʃn]

* 1. 7种传播行为 分为两大类

当前线程必须已经存在事务 和 可以不存在事务 2类

1. **当前线程可以不存在事务**

A 、Required (默认)

如果当前事务，是在前一个事务方法中被调用的，会先判断当前线程中的连接是否已经开启了事务 ，如果开启了事务 那么被调用的方法就不用开启事务

(**将事务合并，直接用前一个事务**)

B、Requires\_new

该方法只要被调用就会新创建一个新的事务， 且不影响其他的事务

如果当前事务，是在前一个事务方法中被调用的，会先判断当前线程中的连接是否已经开启了事务 ，如果开启了事务 ，则将前一个事务挂起（将当前连接保存至其他地方，再创建一个新的连接去执行事物 ）， 开启一个新的事务 去执行，将事务提交后再恢复上一个事务被挂起的事务再去执行提交。 (**前事务挂起再拿一个新连接执行新事务**)

C、Mandatory [ˈmændətɔːri](强制的)  
 处理当前事物之前， 判断当前是否已经存在事务 ，如果存在强制加入已存在的 事务 中如果不存在 则直接抛异常 未发现事务 (**强制加入已存在的事务中**)

D、Nested(嵌套)[ˈnestɪd]

如果当前事务，是在前一个事务方法中被调用的 ， 在调用当前事务时会自动 设置保存点 。 如果当前事务回滚 则直接 回滚到保存点。

注意： 如果子事物 没有把异常 吃了 ，还是会全部回滚

同样的效果 ，也有其他实现 默认传播行为的子事务 如果回滚直接 catch捕获异常， 不往上面抛 ，也不会回滚 前面的事务。

1. **只有在当前线程必须存在事务时才被使用**
2. Not\_support ：

如果已经存在事物， 就将事物挂起 ， 当前的事物以非事务方式运行直 接处理业务 ， 处理结束后 再恢复被 挂起的事物

**注意 ：** 由于正常执行事物是 spring将连接交给mybaties执行sql

如果当前以非事物运行， 此时肯定不能共用一个连接，而是新 创建一个连接

1. Supports

当前线程中必须存在事务 ，如果存在事务则当前方法 在这个事务中运行

C、Never

已经存在了 事物 直接抛异常

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

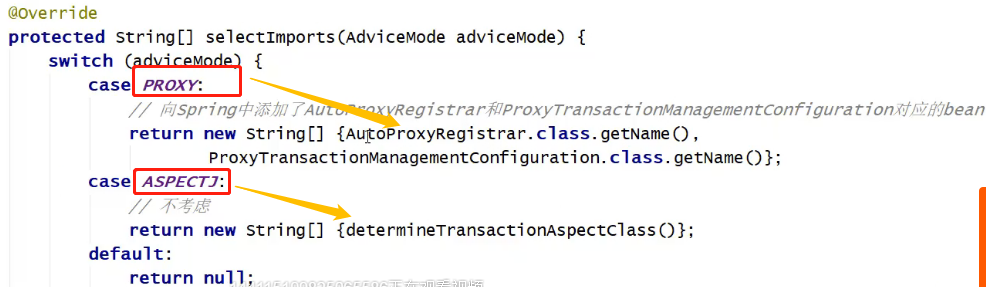
1. Spring 开启事务注解 @EnableTranctionManagent 的实现

Spring 事务是 基于动态代理来实现的 ， 在项目启动时根据 获取加了@Trancttion注 解的类， 根据设置的 不同的事务传播行为 创建不同的代理对象存放至 单例池中。

当项目执行业务操作时 去 进行事务操作。

跟其他的@EnableXXX注解类似，都是依赖@Import注解 注入一个类，然后这个类实现 ImportSelector接口 重写SelectImports接口 。 根据项目中使用不同的动态代理

向 spring 容器中 注入不同的类



1. . AutoProxyRegister

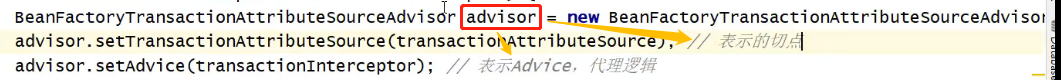
作用 ： 内部会注入 aop代理创建器AutoProxyCreator 使用这个类去进行AOP

获取 Advisor ，而advisor 就是通过下面的

ProxyTrancactionManagerConfiguration 类配置的

1. . ProxyTrancactionManagerConfiguration 事务配置类 ， 先创建advisor

再往advisor中设置 advice + point



1. 设置切点TranctionAttributeSource类其实就对应@Tranction 注解(内部定义 了point) 。 很好理解 ：

AOP 在创建 代理对象时要代理我们的业务方法 ，而**加了@Tranction 注 解 并且是 方法是 public 修饰的** 才会进行事务代理 。

a.事务隔离级别

b.事务传播行为

c.事务回滚

d.事务处理时间限制

e.事务只读

B 、设置代理逻辑 advice TranctionIntercepter

提供了 invoke代理方法，执行代理逻辑

1. 获取事物管理器 DataSourceTransactionManager

创建数据库连接

设置隔离级别

设置timeOut

设置autoCommit = false

将数据库连接 保存到 DataSourceTransactionManager 中的 threadLocal中

1. **创建一个事务** (根据传播机制，去创建事物对象)
   1. 判断传播机制 ，分为当前线程中是否存在事务 两类
      1. **不存在事务**：

Mandatory 、required 、required\_new 、nested

* + 1. **存在事物**

Supports、Not\_support、

1. Try块 执行业务方法

执行业务方法时，是根据事物来决定的 ， 一个线程只能同时处理一 个事务

1. 提交事务
2. catcher 执行回滚逻辑
3. Finally 清除threadLocal 中的连接信息
4. 在spring aop进行代我们的 业务方法时 ， 加@Tranction 表示是定义切点

具体的代理逻辑，是根据 传播行为来决定的。

代理对象的功能大致就是 ：

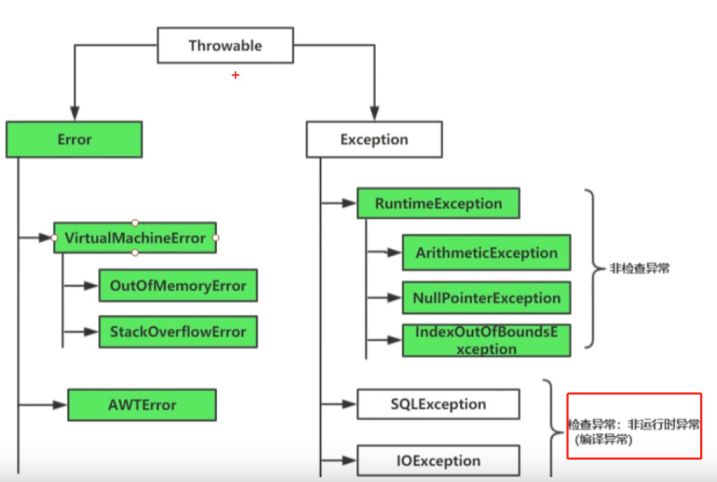
1. 获取连接
2. 设置事务为手动提交
3. 执行业务方法操作数据库
4. 提交 或者 回滚
5. 事务失效的问题
   1. 使用事务的类 必须是aop 代理对象，如果@Tracntion方法中是用 this调用方法， 则不是代理对象， 就不能实现事物
   2. 当事务方法调用 另一个事务方法时 不指定传播行为 会失效

因为内部直接调用方法， 其实是this.xx(); this不是代理对象

* 1. 方法不是 public修饰的 ， 使用其他修饰的关键字 ，等于没加@Tracntion注解
  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)



1. 事物执行 流程总结 ：
   1. 创建事物管理器
      1. 创建连接 设置隔离级别、设置手动提交、设置执行过期时间
      2. 将连接保存至 事物管理器的 threadLocal中
   2. 创建事物 判断传播行为
   3. 实行业务逻辑
   4. 提交事务
      1. 回滚事务
      2. Finall 清空事务管理器中的 连接信息

**Spring整合mybaties**

1. Spring 整合mybaties的意义在于 将mapper的代理对象交给spring容器进行管理

然后 service对象才能依赖 mapper

1. 如何将 mapper代理对象交个 spring 容器 ? (通过**ImportBeanDefinitionRegister bd 注册接口 将需要注入的对象 以 beanDefinition的形式交给 spring管理 + FactoyBean接口 将代理对象创建成一个 bd 交给spring容器**)

需求注入的对象 是一个个mapper 接口的代理对象 ， 这些接口是 通过@MapperScanner 注解扫描来获取

**旧版本如何实现 ：**

* 1. MapperScannerRegister实现ImportBeanDefinitionRegister 注册bd的接口
  2. 在接口中 获取 mybaties配置类
  3. 定义扫描器 这个扫描器继承了 spring ClassPathBeanDefinitionScanner ，

然后扫描器根据@MapperScanner注解路径的值 获取项目中该路径下定义

所有的calss文件 存放至 set 集合中

* 1. 然后判断 calss 类还是 接口， 如果是接口才获取

**注意** ： 在spring中的scaner再扫描类时 是只扫描类，不扫描接口 mybaties重写的 扫描的过滤方法

* 1. 定义MapperFactoryBean类 实现 FactoryBean接口 和继承SqlSessionDaoSupport

1)继承SqlSessionDaoSupport 用于获取SqlSessionTemplete对象，进而获取

SqlSession ，进而通过 sqlSession.getMapper() 获取 mapper代理对象

2)FactoryBean 接口的getObject方法用于返回对象mapper 代理对象

* 1. 然后循环这些接口

1. . 创建 空的 bd
2. 将当前mapper 作为参数传至 MapperFactoryBean ，然后将 MapperFactoryBean 添加至空的 bd中 ， 再把新的bd 注册给spring
3. . 后面spring在进 bean 的实例化时， 就会对bd进行处理创建对象存放至 spring的单例池中

**新版本实现**

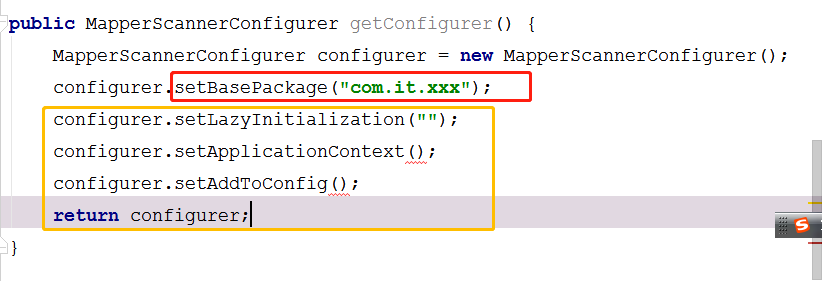
1. MapperScannerRegister实现ImportBeanDefinitionRegister 注册bd的接口
2. 然后不进行扫描., 直接注册一个 MapperScannerConguration

MapperScannerConguration 实现了 BeanDefinitionRegisterPostProcessor ，

在bean的实例化执行spring bd的后置处理器时会调这个接口

注册新的bd。 注册新的 bd时 再去 取扫描，扫描时可以不用 @MapperScanner() ,可以用MapperSannerConfigure类

**为什么中间多加了一步 ？ 目的为了不仅可以支持扫描 还可以自由扩展扫 描规则 以及 设置 懒加载、懒加载、SqlSessionTemplete等**



1. Mapper代理对象是如何生成的

通过getMapper怎么获取 mapper的 ？

通过 Configuration 对象 ，getMapper方法获取代理对象

1. 通过SqlSeesion对象获取 代理对象

UserMapper mapper = SqlSession.getMapper(UserMapperInterface);

1. SqlSeesion 如何获取 ？ 通过 SqlSessionTemplete ，

SqlSessionTemplete 其实就是 sqlSession ，实现了 sqlSession接口

1. SqlSessionTemplete（sqlSession） 如何生成的 ？

依赖SqlSessionFactory (依赖dataSource) 从而SqlSessionTemplete具有连接操作数 据库的功能有 增删改查数据库的方法

1. 关于 mybaties 中的 @mapper 注解

在配置类上 定义统一扫描 @MapperScannser 或者 在 mapper上定义 @Mapper

**Hystrix**

1. Hystrix ： 当某个服务故障 通过 降级、熔断、限流 的方式来保护整个项目的健壮性

主要功能 ： 降级 、熔断、超时 、限流

1. 问题 ： 服务雪崩 如何应对 ？ 缓存击穿、服务响应超时 项目中是如何应对的

使用 hystrix

1. .
2. 降级 、 熔断 、 限流

**降级**  ： 发送请求时 ，当服务出现故障 直接执行本地方法

A、触发 ： 服务报错

B、 请求超时 ， hystrix默认请求超时时间是1s

**熔断** ： 当大量请求都触发降级 ，就直接**打开**熔断器 直接执行本地降级方法

如何设置 ： requestVolumeThreshold  ：20

表示10秒的时间内出现降级20次 触发熔断

熔断状态 ： 关闭 、 半开 、 全开

全开 ： 降级20次 打开熔断 。 再过10秒中后置为半开状态

半开 ： 一旦出现降级 就打开 当继续访问如果服务正常就就会把熔断关闭

**限流** ： hystrix通过线程池的来维护 服务的请求的调用 默认10个线程

如果请求很多，10个线程处理不了 那么就限流

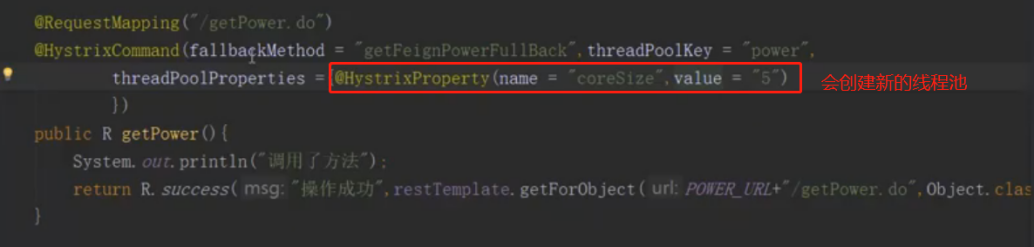
1. 限流 @HystrixConmmand 如果触发降级就调 forbackMethod配置的属性

如果不配置 则是默认全局的default配置，会有一个线程池 ， 里面有10个线程 如果线程都在使用 来了请求则会触发降级

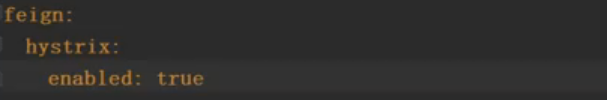
如果**配置 等待队列** ， 10个线程都在工作 又来了请求 那么请求会入队 ，如果对列满了 ，则再触发降级

注意 ： 如果不配置线程池 或者 配置了全局的default 的情况下 再在接口配置单独

线程池重新执行个数 ，则会创建一个新的线程池（不建议配置多个）

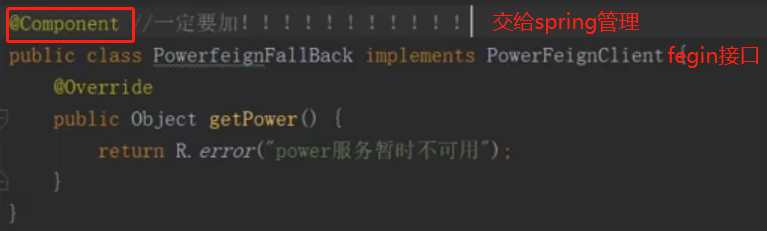


1. 通过fegin 来定义统一的降级方法
   1. Fegin整合 hystrix时 需要启用 hystrix



* 1. 定义统一的 降级方法

1. 先定义一个类实现 fegin 接口
2. 然后重写fegin 接口 里的方法 ， 那么当fegin调远程的接口出问题时 ，则进行降级触发调用 本地方法



**Gteway网关**

2. .
3. .

**Zuul网关**

1. .主要功能 ： 过滤 、路由请求 。 在启动时将自己作为一个服务 注册给 Eureka ，通过从Eureka server中拉取服务列表 。 后面当请求访问项目时 Eureka会想将请求分发给

Zuul ，然后再去请求相应的服务

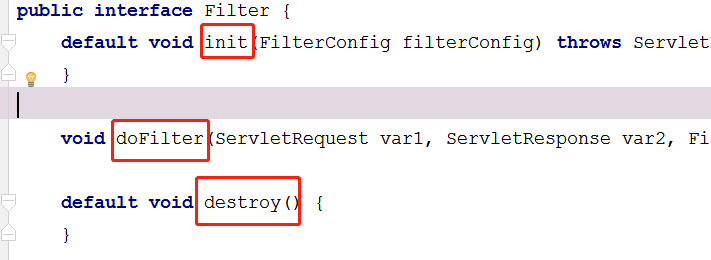
1. 路由 ： 定义 不同的服务模块的请求 前缀



1. 过滤 ： 4种类型过滤器

过滤器都实现并扩展了 java 的Filter接口

Java 过滤器 ： 定义了生命周期 ， 初始化 ----> 执行----->销毁 。



这4种过滤器在执行的各个阶段会被调用 ， 如果项目中没有自定义过滤器 zuul会在各个阶段执行自己默认的过滤器 ， 如果自己定义了过滤器 也可以对过滤器进行排序

* 1. . pre\_filter 请求被路由之前 执行 (**认证授权、获取请求信息做相关操作**)
  2. .routing\_filter 在请求被路由之后到达服务之前 执行 (**负载均衡)**
  3. .post filter 服务处理完请求后 ，执行 **(对返回的数据 进行业务上的二次处理，如 规范返回结果的 code 、msg、data)**
  4. .error filter 请求出错 执行

如何使用 ：

1. 继承 ZuulFilter
2. 定义 过滤器类型 通过filterType() 方法 renturn 常量。
3. 定义 相同类型过滤器 执行的优先级，通过 order方法 通过 常量 加1 或减 1 ，自定义的过滤器 最好在默认的之后
   1. zuul 整合 hystrix ：实现 FallbackProvider 接口

zuul 整合 hystrix是**监听整个服务如果调用出现异常** ，进行降级 (因为这里是服务入口)

如果在服务内 fengin调接口接口时 出现服务降级 ，对于网关来说 是正常请求

1. zuul 集群 ： 一个zull 挂了 会影响整个微服务 ， 因此需要集群

**HystrixDashbord 微服务监控（Dashbord 仪表盘） + actuator**

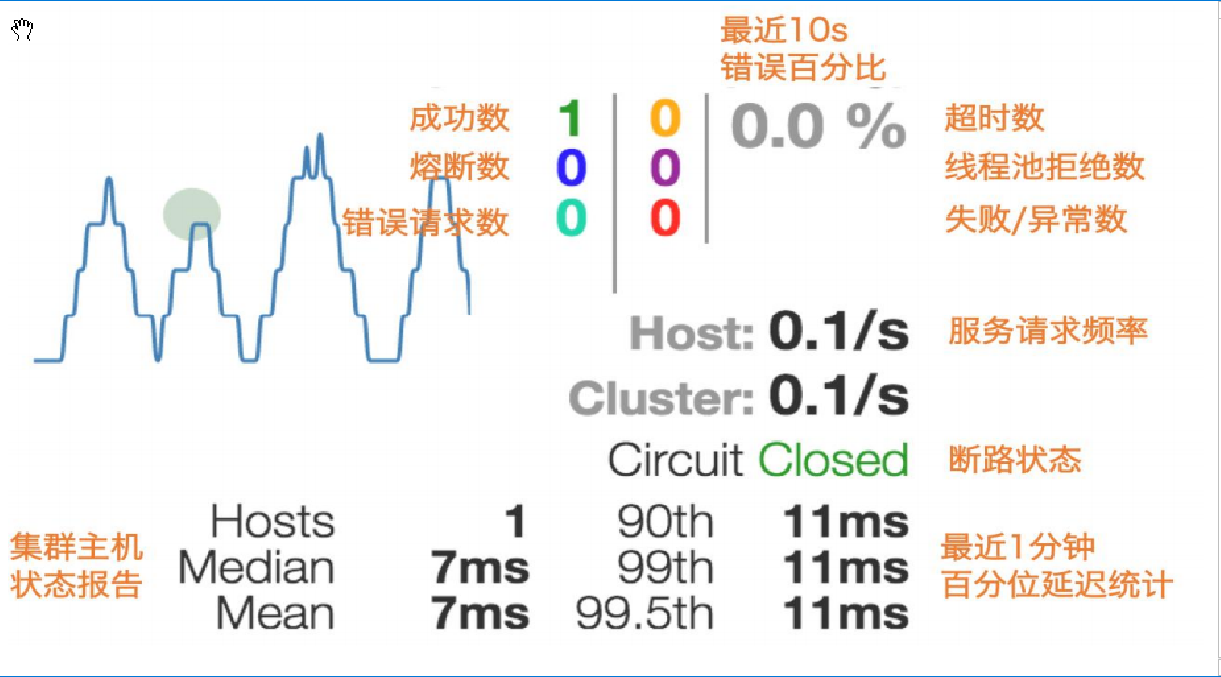
1. 引入 springboot的 actuator [ˈæktjuˌeɪtər](马达 机器拖动装置) 捕捉微服务之间的信息调用，并以 json的方式输出
2. SpringClould F 版本以后 必须要配置 include, 这样actuator才能捕捉项目信息

management.endpoint.web.exposure.include: '\*'

1. HystrixDashbord 解析actuator输出的信息 以界面的方式展示 ，使用需引入 HystrixDashbord 依赖

**这样 HystrixDashbord 就能监控到 跟actuator模块的任何一个微服务**

1. .hystrixDashBord 监控见面 各参数说明



**实心圆：**共有两种含义。它通过颜色的变化代表了实例的健康程度，它的健康度从绿色

该实心圆除了颜色的变化之外，它的大小也会根据实例的请求流量发生变化，流量越大 该实心圆就越大。所以通过该实心圆的展示，就可以在大量的实例中快速的发现故障实 例和高压力实例。

**曲线：**用来记录2分钟内流量的相对变化，可以通过它来观察到流量的上升和下降趋势。

**Config分布式配置中心**

1. Config模块作为一个单独的服务 ， 用于管理各模块的 yml配置文件

配置文件 可以放在github、gitlab 、码云上 。 然后Config模块 连接远程的 github 、 gitlab等 进行统一的管理 然后模块连接 Config 即可

应用Nacos 也可以作为配置中心

**分布式配置中心SpringClould分布式链路跟踪组件 sleuth**[sluːθ] + ZipKin

1. sleuth的主要作用 ：

(1)用于快速定位问题

在微服务较多的前提下，如果出现问题报错 并不知道 是哪一步出现问题导致的， 可以使用sleuth 解决这种问题 。 能记录微服务调用链处理业务的情况

(2)各个微服务之间调用的性能分析

在请求到达各个微服务的过程中，sleuth都会记录时间戳这样 就可以知道各个微 服务的处理业务的响应时间

1. .sleuth的原理

(包括 spring 的各个组件的请求交互)

trance id ：记录一个完整请求的 唯一标识

span id ： 当请求到达 各个微服务时，再用span id 进行记录 ， 并且会记录 请求的状态 请求接收状态 ，请求发送状态 以及时间戳 这些信息都是以 json的形式进行存储 然后可以整合 zipkin server , zipkin client 以界面的形式展示数据

注意 ： 官网不建议自己搭建 zipkinserver ， 最好使用 docker

**NACOS**

1. .nacos 功能 ： 即可作为 配置中心 、 又可作为服务注册中心

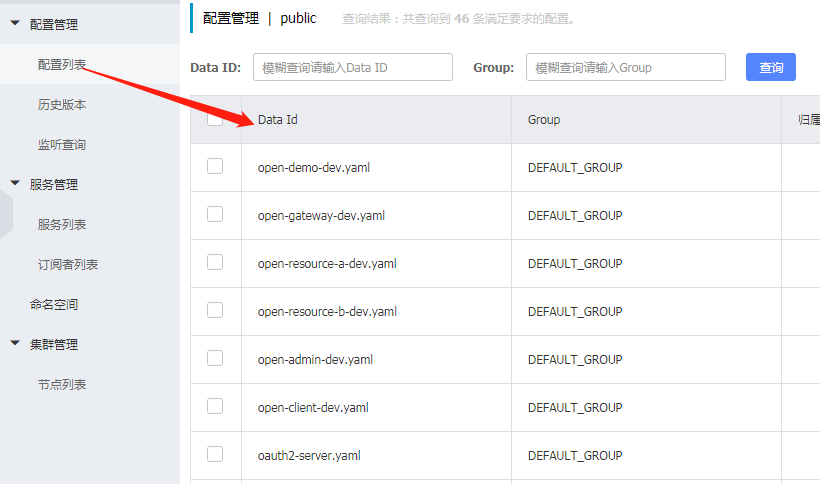
并且可以进行 AP(默认) 和 CP(一般配合k8s) 之间切换

AP 模式 ：注册的微服务实例 是保存在内存中（临时的）

CP 模式 ：会将 存储的数据进行持久化 （生成环境使用）

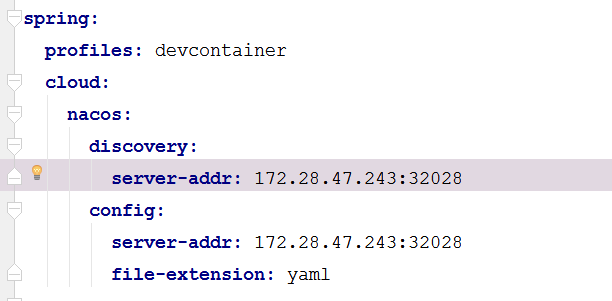
1. 特点 ： 几乎兼容市面上所有的微服务组件 ： dubbo、springcloud、k8s等
2. 如何跟spring cloud整合 ：
   1. 安装 nacos 服务端
   2. 登录 nacos （nacos 整合 sringSecurity 做了用户认证 用户/密码 nacos/nacos）
3. 配置列

将 yml文件 存放 nacos ， 然后 本地yml配置连接nacos



Nacos 自带本地数据库 ，当nacos客户端重启后 配置文件仍然还在

1. 服务注册列表
2. 服务模块引入 nacos客户端依赖 ，Yml配置文件连接nacos 服务
3. 开启 @EnableDiscoveryClient





1. Nacos 配置中心 文件命令规则

前缀(项目名) - active.后缀 (**如果出现多个 - 只匹配最后一个 -**)

前缀： spring.application.name(**前缀可以配置两**

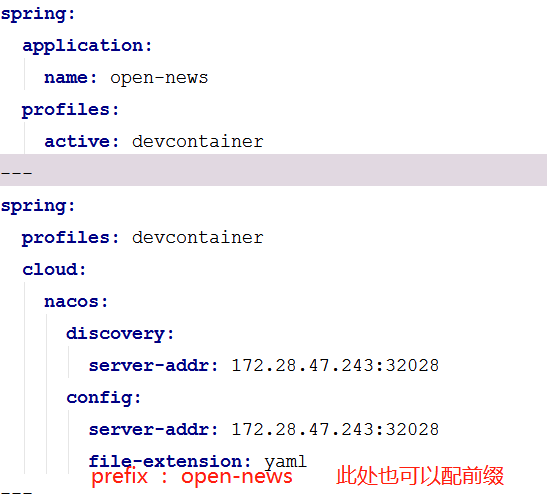
**也可以配 spring.cloud.nacos.config.prefix ,但是使用时只能配其中一个** )

activ e： spring.profiles.active

后缀： spring.cloud.nacos.config.file-extension

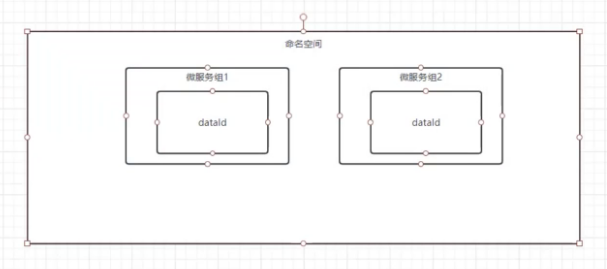
如果 active 不配 ，直接去读取： 前缀.后缀

open-news-devcontainer.yml 配置如下：



spring.cloud.nacos.config 还可以指定 组，如果没有配置 则使用默认的 DEFAULT\_GROUP

1. 命名空间 、 groupid 、dataid之间的关系



命名空间可以配置多个 ，默认是public

每个命名空间中的用户组也可以配置多个，并且互相隔离

1. nacos的热加载 @RefreshScope 注解

使用@RefreshScope 注解 ，如果修改了 nacos配置文件 ，在项目无需重新启动的情况下 就能热加载到

热加载的实现原理 ： 每个配置文件 nacos客户端都会对其监听，一旦服务端数据发 生变化就触发监听 ，进行热加载 。 通过底层通过反射将@RefreshScope动态改变的 属性进行修改

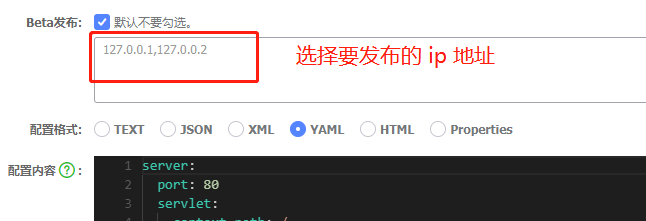
1. Nacos集群



也可以不用 nginx通过直连的方式连接

1. 如果做 nacos 集群是不能使用本地数据库的,由于集群要数据共享需要连接 mysql
2. 执行nacos提供的数据库脚本
3. 修改配置文件
4. 灰度发布

用于生产环境上的项目发布 ， 当修改nacos配置后 并不知道是否对 现有程序有影响，可以灰度发布 进行测试 ，测试没问题再同步到所有集群



**Sentinel [ˈsentɪnl]**

1. Sentinel 的功能

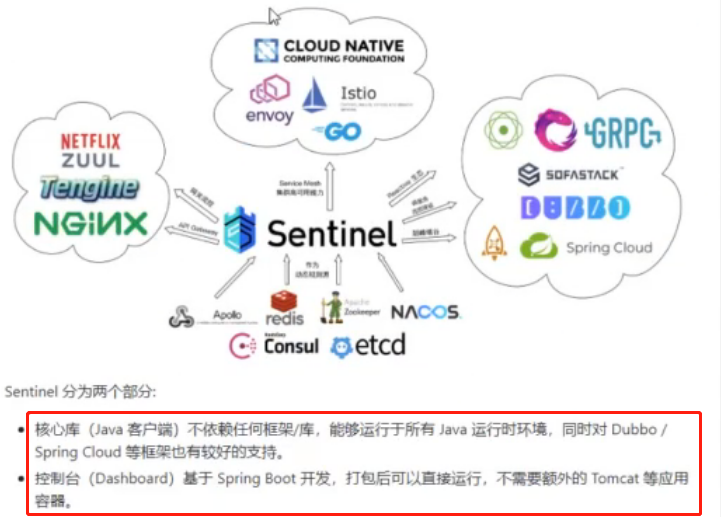
集群流量控制 (可以精确到接口、方法)

网关流量控制 (可以精确到接口、方法)

黑白名单控制

限流(系统自适应限流)、降级、熔断

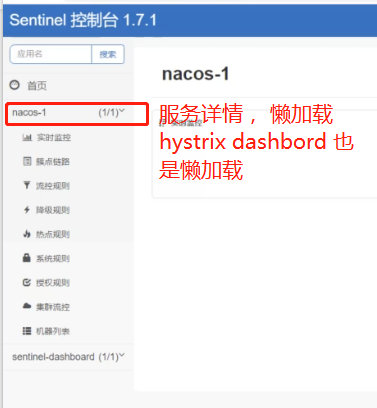
1. sentinel架构 ： 几乎可以整合市面上java相关的所有应用



客户端 ： 整合微服务模块等

服务端 ： 监控系统、配置流控规则

Sentinel 服务端是以springboot项目 ， 下载后直接java -jar 可以运行服务



**每个微服务都的不同节点都可以有如下设置 ， 并且设置的参数是存在 客户端的，如果客户端的微服务进行了重启，这些参数就会被重置成默认的 。 也可以添加持久化功能 来保存这些参数**

**实时监控**：

**触点链路**： 展示所有controller接口， 并可以动态的修改接口进行限流



限流的方式 ：

qps限流 ： 控制系统的每秒流量

线程数限流 ： 用于控制系统资源，根据系统的性能合理的取线程数

**流控模型 ：**

1. 直接 ： 只针对一个接口
2. 关联 ： 针对两个远程相互调用的接口 ， 一个资源名接口 和 关 联资源接口 。 如果配置 关联接口的qps小于 当前资 源接口 .那么配置的关联接口会失效 。

**因为 ：**假如调订单接口减库存操作 ， 库存接口的qps 设 置的参 数小于订单接口。 那么当qps刚好超过减库 存的阈值时， 减库存不能处理请求 ，但是还能下订单， 这样会有大量的数据订单数据进来，减库存不去操作 。 这种情况是不允许出现的。

1. .链路 ：

**流控资源：**

1. 快速失败 ： 超过 qps 或者 线程处理的能力 直接报错
2. 预热 ： 如果系统访问被访问的频率很低，突然来了大量的访问(恶 意访问) 系统可能承受不了， 可以设置预热 。

使用 ：配置正常的 qps (10)，但是正常访问的qps是除以3

等于3。 当系统有大量访问过来时， 等待一段时间后则会自动切换为10 。 当超过配置的时间后 ，会重置配置 降为3

1. 排队等待 ：超过 qps 或者 线程处理的能力排队等待，如果等到 超时直接报错

**流控规则**：

**降级规则**：



1. RT ： 平均响应时间，如果超过了 这个时间 ，将触发熔断报错

(2)异常比例 ： 0.0 - 1.0 取值范围 .描述**1s内**异常请求的比例，超过比例 触 发熔断报错

(3)异常数 ： 1分钟内 ， 到达异常数 触发熔断报错

**熔断后如何对请求降级** ：

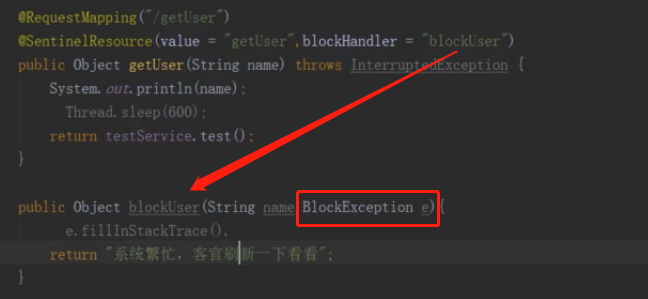
@SentinelResource(请求地址，fallback=””，blockHandler=””,blockHandlerClass=“”)

Fallback、blockHandler、blockHandlerClass 都是降级方法， 但不同

blockHandler ： 是流控触发 调用的降级方法。

Fallback ：是没触发流控，但是接口调用异常 调用的降级方法

blockHandlerClass : 是定义一个降级处理类



注意 ： @requestMapping 和 SentinelResource 如果value不同 ，是表 示不同的资源 ， 如果给 getUser配置流控 。 访问url是 /getUser ， 是不会触发流控的

**热点规则：根据访问接口 传入的具体参数，进行控制限流 ，如敏感性的 关键字**



1. .

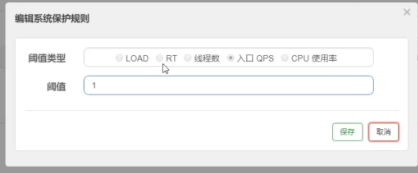
授权规则：

系统规则：

用于定义全局的规则 ， 如果某个资源有自己的规则

会先去走系统默认的规则 ，如果能通过 再走方法的规则。

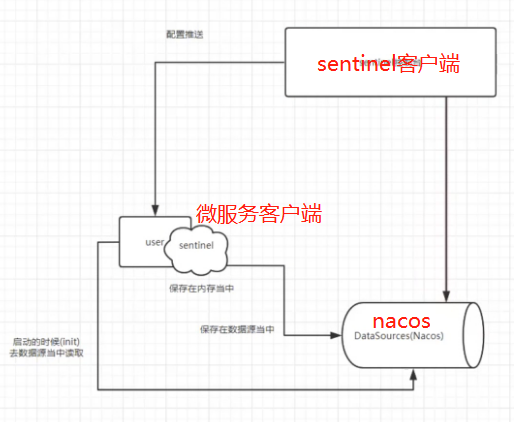
系统规则不通过 则直接熔断或降级



LOAD : 表示 在Linux系统下的系统核数 \* 2.5 去处理请求

1. Sentinel 持久化

Sentinel 客户端设置的规则 ，默认是基于内存的 。 当客户端微服务重启启动后之 前设置的 规则互丢失 。 因此生成环境要做 sentinel持久化 。官网只提供了 文 件持久化的方式。 可以整合 nacos 数据以文件的形式同步至 nacos



1. 微服务客户端 加 sentinel DataSource 依赖
2. Sentinel 微服务客户端重启时 ，去连接nacos 配置 使用@PostConstract 在 spring初始化前去读取 。
3. Sentinel 客户端注册至nacos ，当客户端规则改变时 再同步至 nacos

但是 sentinel 除了客户端能配置规则 ，没有提供代码 写规则的功能

2. .
3. .