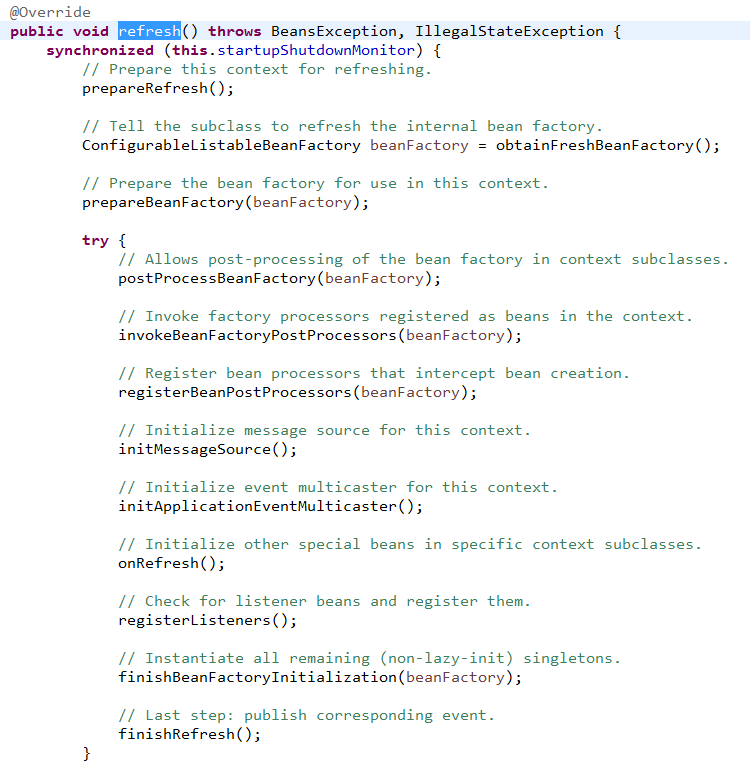
# 容器基本实现与组成

**IOC容器处理流程（其实就是研究一下refresh()下的这些方法）**



Spring容器的refresh()【创建刷新】;

1、prepareRefresh()刷新前的预处理;

1）、initPropertySources()抽象方法,初始化一些属性设置;子类自定义个性化的属性设置方法,把servletConfig和servletContext放到spring容器的propertySources容器里面.；

2）、getEnvironment().validateRequiredProperties();检验属性的合法等

3）、earlyApplicationEvents= new LinkedHashSet<ApplicationEvent>();保存容器中的一些早期的事件；主要还是初始化

2、obtainFreshBeanFactory();获取BeanFactory；

1）、refreshBeanFactory(); 抽象方法 刷新【创建】BeanFactory；

如果有工厂容器,先销毁,后创建

**return** **new** DefaultListableBeanFactory(getInternalParentBeanFactory())

设置id；

customizeBeanFactory(beanFactory);设置属性allowBeanDefinitionOverriding和allowCircularReferences

loadBeanDefinitions(beanFactory);抽象方法,加载BeanDefineitions, DefaultListableBeanFactory实现了BeanDefinitionRegistry.方法registerBeanDefinition注册beanDefinition到beanDefinitionMap

2）、getBeanFactory();返回刚才GenericApplicationContext创建的BeanFactory对象；

3）、将创建的BeanFactory【DefaultListableBeanFactory】返回；

3、prepareBeanFactory(beanFactory);BeanFactory的预准备工作（以上创建了beanFactory,现在对BeanFactory对象进行一些设置属性）；

1）、设置BeanFactory的类加载器、支持表达式解析器...

2）、添加部分BeanPostProcessor【ApplicationContextAwareProcessor】

3）、设置忽略的自动装配的接口EnvironmentAware、EmbeddedValueResolverAware、xxx；

4）、注册可以解析的自动装配；我们能直接在任何组件中自动注入：

BeanFactory、ResourceLoader、ApplicationEventPublisher、ApplicationContext

5）、添加BeanPostProcessor【ApplicationListenerDetector】

6）、添加编译时的AspectJ；

7）、给BeanFactory中注册一些能用的组件；

environment【ConfigurableEnvironment】、

systemProperties【Map<String, Object>】、

systemEnvironment【Map<String, Object>】

4、postProcessBeanFactory(beanFactory);BeanFactory准备工作完成后进行的后置处理工作；

1）、子类通过重写这个方法来在BeanFactory创建并预准备完成以后做进一步的设置

======================以上是BeanFactory的创建及预准备工作==================================

5、invokeBeanFactoryPostProcessors(beanFactory);执行BeanFactoryPostProcessor的方法；

BeanFactoryPostProcessor：BeanFactory的后置处理器。在BeanFactory标准初始化之后执行的；

两个接口：BeanFactoryPostProcessor、BeanDefinitionRegistryPostProcessor

1）、执行BeanFactoryPostProcessor的方法；

先执行BeanDefinitionRegistryPostProcessor

1）、83行：获取所有的BeanDefinitionRegistryPostProcessor；

2）、86行：看先执行实现了PriorityOrdered优先级接口的BeanDefinitionRegistryPostProcessor、

postProcessor.postProcessBeanDefinitionRegistry(registry)

3）、99行：在执行实现了Ordered顺序接口的BeanDefinitionRegistryPostProcessor；

postProcessor.postProcessBeanDefinitionRegistry(registry)

4）、109行：最后执行没有实现任何优先级或者是顺序接口的BeanDefinitionRegistryPostProcessors；

postProcessor.postProcessBeanDefinitionRegistry(registry)

再执行BeanFactoryPostProcessor的方法

1）、139行：获取所有的BeanFactoryPostProcessor

2）、147行：看先执行实现了PriorityOrdered优先级接口的BeanFactoryPostProcessor、

postProcessor.postProcessBeanFactory()

3）、167行：在执行实现了Ordered顺序接口的BeanFactoryPostProcessor；

postProcessor.postProcessBeanFactory()

4）、175行：最后执行没有实现任何优先级或者是顺序接口的BeanFactoryPostProcessor；

postProcessor.postProcessBeanFactory()

6、registerBeanPostProcessors(beanFactory);注册BeanPostProcessor（Bean的后置处理器）【 intercept bean creation】

不同接口类型的BeanPostProcessor；在Bean创建前后的执行时机是不一样的

BeanPostProcessor、

DestructionAwareBeanPostProcessor、

InstantiationAwareBeanPostProcessor、

SmartInstantiationAwareBeanPostProcessor、

MergedBeanDefinitionPostProcessor【internalPostProcessors】、

1）、189行：获取所有的 BeanPostProcessor;后置处理器都默认可以通过PriorityOrdered、Ordered接口来执行优先级

2）、204行：先注册PriorityOrdered优先级接口的BeanPostProcessor；

把每一个BeanPostProcessor；添加到BeanFactory中

beanFactory.addBeanPostProcessor(postProcessor);

3）、224行：再注册Ordered接口的

4）、236行：最后注册没有实现任何优先级接口的

5）、最终注册MergedBeanDefinitionPostProcessor；

6）、注册一个ApplicationListenerDetector；来在Bean创建完成后检查是否是ApplicationListener，如果是

applicationContext.addApplicationListener((ApplicationListener<?>) bean);

7、initMessageSource();初始化MessageSource组件（做国际化功能；消息绑定，消息解析）；

1）、718行：获取BeanFactory

2）、719行：看容器中是否有id为messageSource的，类型是MessageSource的组件

如果有赋值给messageSource，如果没有自己创建一个DelegatingMessageSource；

MessageSource：取出国际化配置文件中的某个key的值；能按照区域信息获取；

3）、739行：把创建好的MessageSource注册在容器中，以后获取国际化配置文件的值的时候，可以自动注入MessageSource；

beanFactory.registerSingleton(MESSAGE\_SOURCE\_BEAN\_NAME, this.messageSource);

MessageSource.getMessage(String code, Object[] args, String defaultMessage, Locale locale);以后可通过getMessage获取

8、initApplicationEventMulticaster();初始化事件派发器；

1）、753行：获取BeanFactory

2）、754行：从BeanFactory中获取applicationEventMulticaster的ApplicationEventMulticaster；

3）、762行：如果上一步没有配置；创建一个SimpleApplicationEventMulticaster

4）、763行：将创建的ApplicationEventMulticaster添加到BeanFactory中，以后其他组件直接自动注入

9、onRefresh();留给子容器（子类）

1、子类重写这个方法，在容器刷新的时候可以自定义逻辑；

10、registerListeners();给容器中将所有项目里面的ApplicationListener注册进来；

1、822行：从容器中拿到所有的ApplicationListener

2、824行：将每个监听器添加到事件派发器中；

getApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName);

3、832行：派发之前步骤产生的事件；

11、finishBeanFactoryInitialization(beanFactory);初始化所有剩下的单实例bean；

1、867行：beanFactory.preInstantiateSingletons();初始化后剩下的单实例bean，跟进

1）、734行：获取容器中的所有Bean，依次进行初始化和创建对象

2）、738行：获取Bean的定义信息；RootBeanDefinition

3）、739行：Bean不是抽象的，是单实例的，是懒加载；

1）、740行：判断是否是FactoryBean；是否是实现FactoryBean接口的Bean；

2）、760行：不是工厂Bean。利用getBean(beanName);创建对象

0、199行：getBean(beanName)； ioc.getBean();

1、doGetBean(name, null, null, false);

2、246行： getSingleton(beanName)先获取缓存中保存的单实例Bean《跟进去其实就是从MAP中拿》。如果能获取到说明这个Bean之前被创建过（所有创建过的单实例Bean都会被缓存起来）

从private final Map<String, Object> singletonObjects = new ConcurrentHashMap<String, Object>(256);获取的

3、缓存中获取不到，开始Bean的创建对象流程；

4、287行：标记当前bean已经被创建（防止多线程同时创建，使用synchronized）

5、291行:获取Bean的定义信息；

6、295行：getDependsOn()，bean.xml里创建person时，加depend-on="jeep,moon"是先把jeep和moon创建出来

【获取当前Bean依赖的其他Bean;如果有按照getBean()把依赖的Bean先创建出来；】

7、启动单实例Bean的创建流程；

1）、462行：createBean(beanName, mbd, args);

2）、490行：Object bean = resolveBeforeInstantiation(beanName, mbdToUse);让BeanPostProcessor先拦截返回代理对象；

【InstantiationAwareBeanPostProcessor】：提前执行；

先触发：postProcessBeforeInstantiation()；

如果有返回值：触发postProcessAfterInitialization()；

3）、如果前面的InstantiationAwareBeanPostProcessor没有返回代理对象；调用4）

4）、501行：Object beanInstance = doCreateBean(beanName, mbdToUse, args);创建Bean

1）、541行：【创建Bean实例】；createBeanInstance(beanName, mbd, args);

利用工厂方法或者对象的构造器创建出Bean实例；

2）、applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);

调用MergedBeanDefinitionPostProcessor的postProcessMergedBeanDefinition(mbd, beanType, beanName);

3）、578行：【Bean属性赋值】populateBean(beanName, mbd, instanceWrapper);

赋值之前：

1）、拿到InstantiationAwareBeanPostProcessor后置处理器；

1305行：postProcessAfterInstantiation()；

2）、拿到InstantiationAwareBeanPostProcessor后置处理器；

1348行：postProcessPropertyValues()；

=====赋值之前：===

3）、应用Bean属性的值；为属性利用setter方法等进行赋值；

applyPropertyValues(beanName, mbd, bw, pvs);

4）、【Bean初始化】initializeBean(beanName, exposedObject, mbd);

1）、1693行：【执行Aware接口方法】invokeAwareMethods(beanName, bean);执行xxxAware接口的方法

BeanNameAware\BeanClassLoaderAware\BeanFactoryAware

2）、1698行：【执行后置处理器初始化之前】applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);

BeanPostProcessor.postProcessBeforeInitialization（）;

3）、1702行：【执行初始化方法】invokeInitMethods(beanName, wrappedBean, mbd);

1）、是否是InitializingBean接口的实现；执行接口规定的初始化；

2）、是否自定义初始化方法；

4）、1710行：【执行后置处理器初始化之后】applyBeanPostProcessorsAfterInitialization

BeanPostProcessor.postProcessAfterInitialization()；

5）、将创建的Bean添加到缓存中singletonObjects；sharedInstance = getSingleton(beanName, ()跟进去

254行：addSingleton（），放到MAP中

ioc容器就是这些Map；很多的Map里面保存了单实例Bean，环境信息。。。。；

所有Bean都利用getBean创建完成以后；

检查所有的Bean是否是SmartInitializingSingleton接口的；如果是；就执行afterSingletonsInstantiated()；

12、finishRefresh();完成BeanFactory的初始化创建工作；IOC容器就创建完成；

1）、882行：initLifecycleProcessor();初始化和生命周期有关的后置处理器；LifecycleProcessor

默认从容器中找是否有lifecycleProcessor的组件【LifecycleProcessor】；如果没有new DefaultLifecycleProcessor();

加入到容器；

自己也可以尝试写一个LifecycleProcessor的实现类，可以在BeanFactory

void onRefresh();

void onClose();

2）、 885行：getLifecycleProcessor().onRefresh();

拿到前面定义的生命周期处理器（BeanFactory）；回调onRefresh()；

3）、888行：publishEvent(new ContextRefreshedEvent(this));发布容器刷新完成事件；

4）、891行：liveBeansView.registerApplicationContext(this);

======总结===========

1）、Spring容器在启动的时候，先会保存所有注册进来的Bean的定义信息；

1）、xml注册bean；<bean>

2）、注解注册Bean；@Service、@Component、@Bean、xxx

2）、Spring容器会合适的时机创建这些Bean

1）、用到这个bean的时候；利用getBean创建bean；创建好以后保存在容器中；

2）、统一创建剩下所有的bean的时候；finishBeanFactoryInitialization()；

3）、后置处理器；BeanPostProcessor

1）、每一个bean创建完成，都会使用各种后置处理器进行处理；来增强bean的功能；

AutowiredAnnotationBeanPostProcessor:处理自动注入

AnnotationAwareAspectJAutoProxyCreator:来做AOP功能；

xxx....

增强的功能注解：

AsyncAnnotationBeanPostProcessor

....

4）、事件驱动模型；

ApplicationListener；事件监听；

ApplicationEventMulticaster；事件派发：

# 装配bean的方式

配置bean的三种方式: Xml、注解、javaConfig

配置scope:

prototype: 多实例：IOC容器启动并不会去调用方法创建对象放在容器中，而是 每次获取的时候才会调用方法创建对象,见test02

singleton: 单实例（默认）：IOC容器启动会调用方法创建对象放到IOC容器中

以后每交获取就是直接从容器（理解成从map.get对象）中拿

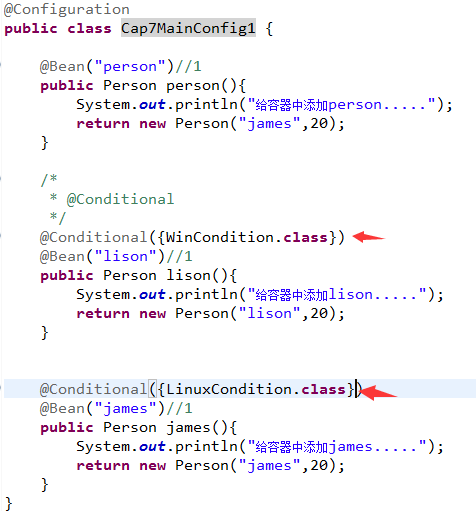
request: 主要针对WEB应用，同一次请求创建一个实例

session: 同一个session创建一个实例（**后面两个用得不多，了解即可**）

@Conditional：**条件注册bean（了解）**

1,将IOC容器注册bean时, 当操作系统为WINDOWS时,注册Lison实例; 当操作系统为LINUX时, 注册James实例,此时要用得@Conditional注解进行定制化条件选择注册bean;

2,新建Cap7MainConfig1.java,



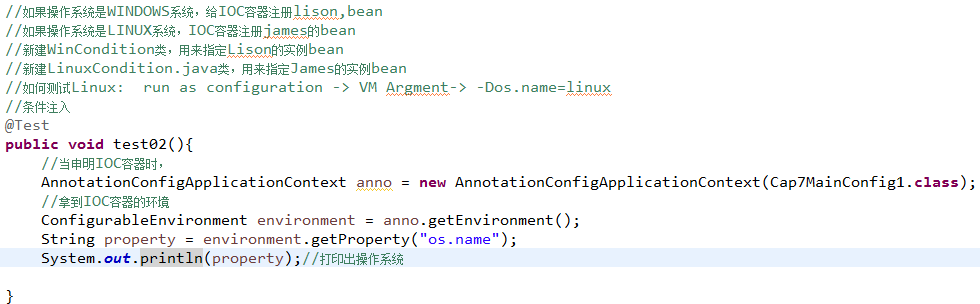
新建WinCondition.java类做为条件类, 同时必须得实现spring提供的Confition接口



新建 LinuxCondition条件类, 用来处理操作系统为LINUX的bean注册



新建test02(), 测试@Conditional 条件加载bean到IOC容器(**加上@Conditinal**)



当引入@Conditional时, 容器可以选择性的注册bean.

@Import(Dog.class) ：注解引入类

ImportSelector可以批量导入组件的全类名数组,自定义逻辑返回需要导入的



除了以上,还可以通过ImportBeanDefinitionRegistrar自定义注册,向容器中注册bean;

@Import({Dog.**class**,Cat.**class**,JamesImportSelector.**class**,JamesImportBeanDefinitionRegistrar.**class**})

新建**JamesImportBeanDefinitionRegistrar**自定义注册类,实现bean注册



当然除了以上加载方式,还可以通过实现FactoryBean接口方式来加载bean

\* 使用spring提供的FactoryBean(工厂bean)

\* beans.factory.FatoryBean源码跟进去

\* 容器调用getObject()返回对象，把对象放到容器中；

\* getObjectType()返回对象类型

\* isSingleton()是否单例进行控制

\* 新建JamesFactoryBean实现FactoryBean

\* 在config里新建jamesFactoryBean()方法

\* 写完test03测试用例后:

\* a,默认获取到的是工厂bean调用getObject创建的对象

\* b,要获取工厂Bean本身,需要在id前加个 &jamesFactoryBean

\*/

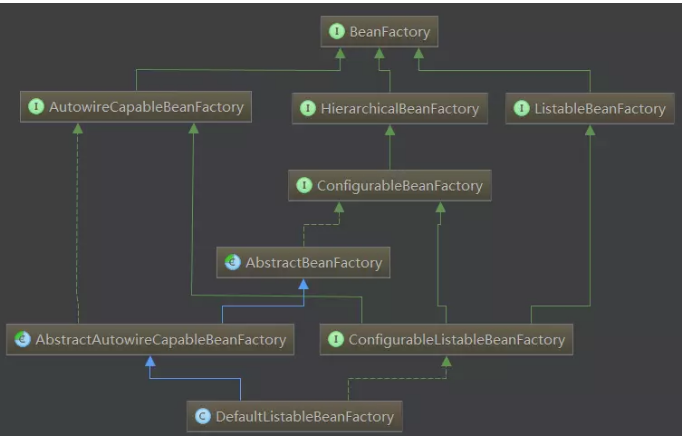
# BeanFactory源码分析

Spring的核心是容器，而容器并不唯一，框架本身就提供了很多个容器的实现，大概分为两种类型：

一种是不常用的BeanFactory，这是最简单的容器，只能提供基本的DI功能；

一种就是继承了BeanFactory后派生而来的ApplicationContext(应用上下文)，它能提供更多企业级的服务，例如解析配置文本信息等等，这也是ApplicationContext实例对象最常见的应用场景。

**spring Ioc容器的实现，从根源上是beanfactory，但真正可以作为一个可以独立使用的ioc容器还是DefaultListableBeanFactory，因此可以这么说，  
DefaultListableBeanFactory 是整个spring ioc的始祖。**



1. **BeanFactory接口：**  
    是Spring bean容器的根接口，提供获取bean，是否包含bean,是否单例与原型，获取bean类型，bean 别名的方法 。它最主要的方法就是getBean(String beanName)。  
   **2.BeanFactory的三个子接口：**  
    \* HierarchicalBeanFactory：提供父容器的访问功能

**两个方法:** getParentBeanFactory 获取父类工厂

containsLocalBean (String name) 本容器中是否包含

 \* ListableBeanFactory：提供了批量获取Bean的方法  
 \* AutowireCapableBeanFactory：在BeanFactory基础上实现对已存在实例的管理  
**3.ConfigurableBeanFactory：**  
主要单例bean的注册，生成实例，以及统计单例bean  
**4.ConfigurableListableBeanFactory：**  
继承了上述的所有接口，增加了其他功能：比如类加载器,类型转化,属性编辑器,BeanPostProcessor,作用域,bean定义,处理bean依赖关系, bean如何销毁…  
**5.实现类DefaultListableBeanFactory**[**详细介绍**](https://www.cnblogs.com/sten/p/5758161.html)**：**  
实现了ConfigurableListableBeanFactory，实现上述BeanFactory所有功能。它还可以注册BeanDefinition

接口详细介绍请参考:[揭秘BeanFactory](https://blog.csdn.net/u011179993/article/details/51636742)

**，扩展原理－BeanFactoryPostProcessor**

BeanFactoryPostProcessor：beanFactory的后置处理器；

作用如下：

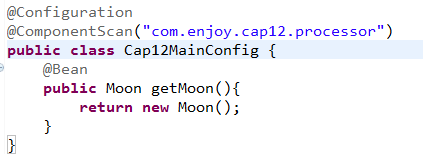
a),在BeanFactory标准初始化之后调用，来定制和修改BeanFactory的内容；

b),所有的bean定义已经保存加载到beanFactory，但是bean的实例还未创建

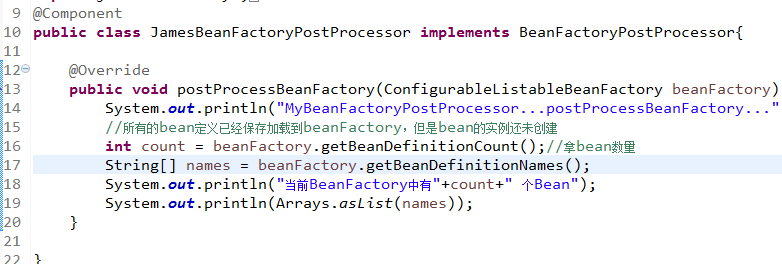
**注意：**之前也讲过BeanPostProcessor，它是bean后置处理器，bean创建对象初始化前后进行拦截工作的

操作步骤：

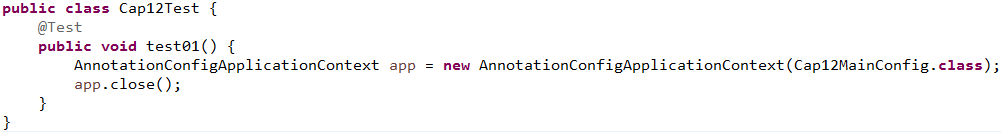
1〉新建ExtConfig.java配置类



新建JamesBeanFactoryPostProcessor.java处理器类（在com.enjoy.cap12.processor目录下）

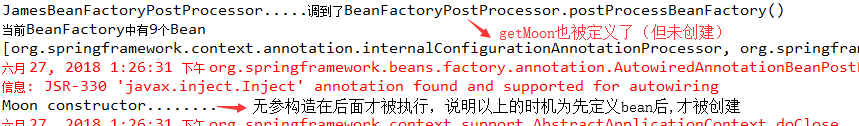


BeanFacotry是在bean组件创建之前还是之后生成的呢？写一个测试用例



不难发现，在“Moon constructor”创建之前，当前9个bean已被加载到beanFactory中了。

测试结果结下：



那么以上BeanFactoryPostProcessor执行原理是怎么样的呢，打断点F5调试一下:



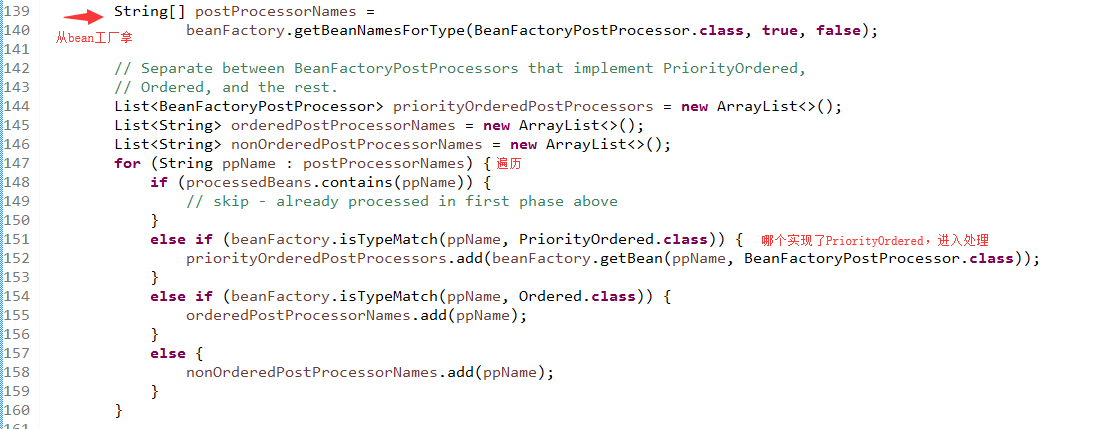
跟踪debug栈,不难发现以下步骤如下：

1)、ioc容器创建对象

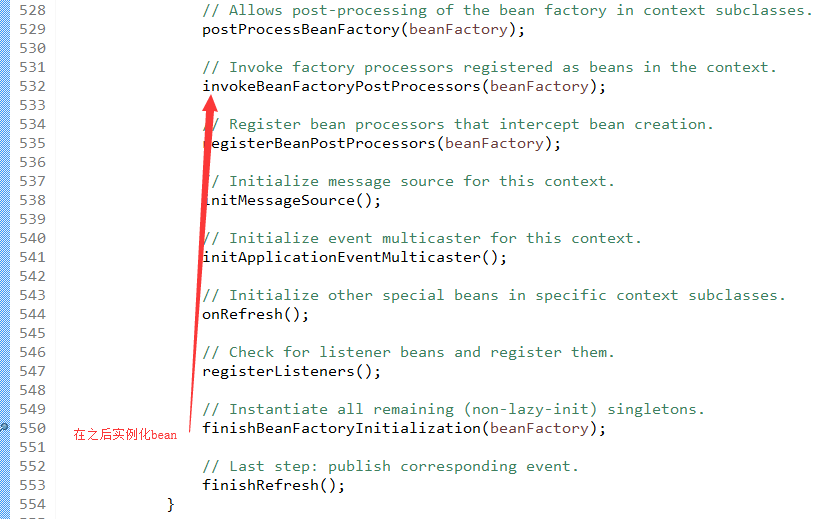
2)、invokeBeanFactoryPostProcessors(beanFactory);

如何找到所有的BeanFactoryPostProcessor并执行他们的方法；

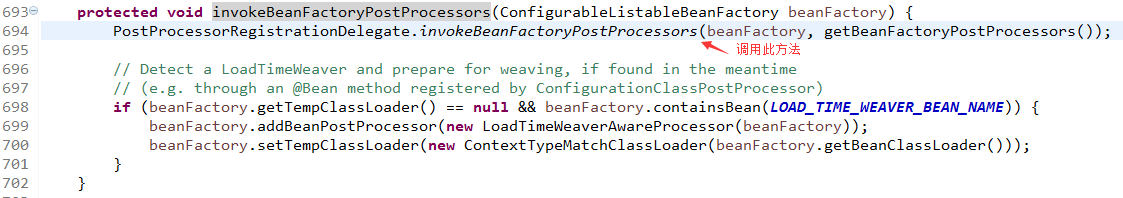
a）、直接在BeanFactory中找到所有类型是BeanFactoryPostProcessor的组件，并执行他们的方法



\* b）、在初始化创建其他组件前面执行



然后再点进去,



# BeanDefinition源码分析

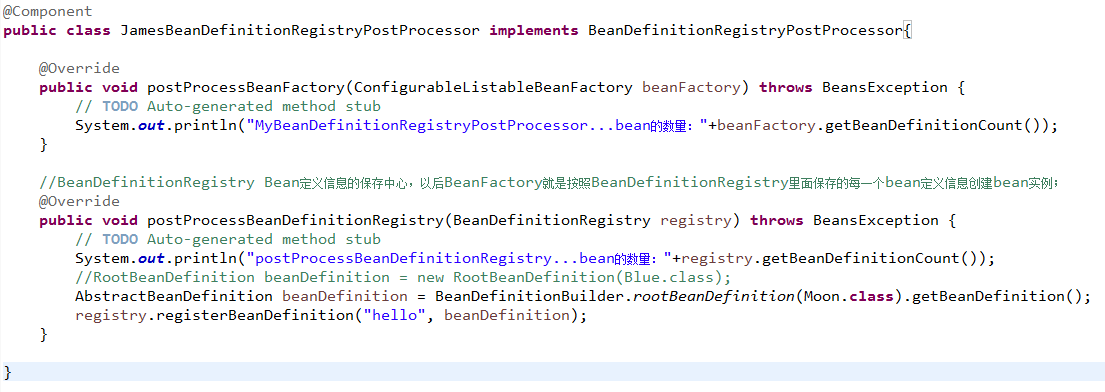
**BeanDefinitionRegistryPostProcessor**

postProcessBeanDefinitionRegistry();在所有bean定义信息将要被加载，bean实例还未创建的；

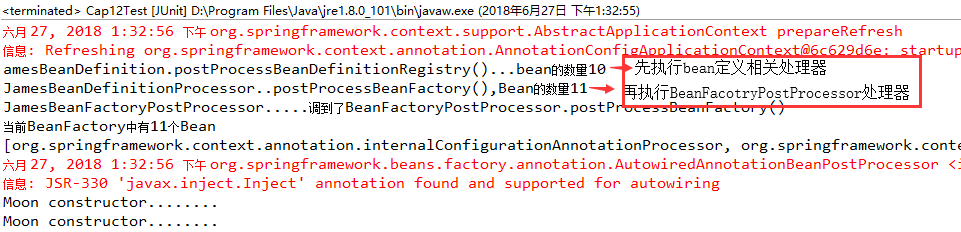
操作步骤：

新建JamesBeanDefinitionRegistryPostProcessor

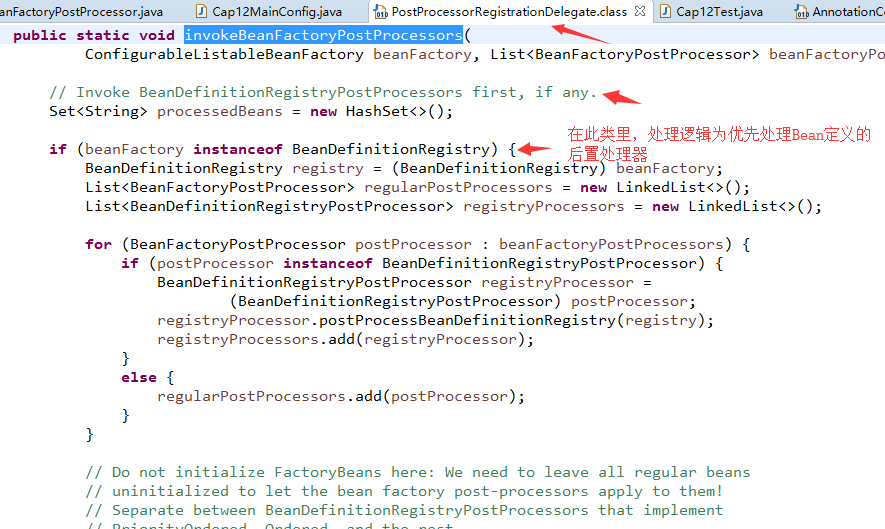
**注意**：可使用BeanDefinitionBuilder构建器来创建bean定义信息



测试结果如下：



源码分析：从refresh()-->invokeBeanFactoryPostProcessors()-->PostProcessorRegistrationDelegate.invokeBeanFactoryPostProcessors(), 跟进去即发现代码逻辑如下：



所以，在任何情况下都会优先执行BeanDefinitionRegistryPostProcessor的处理器，而BeanFactoryPostProcessor的处理器在它后面执行

# Bean生命周期

bean的生命周期:指 bean创建-----初始化----销毁 的过程

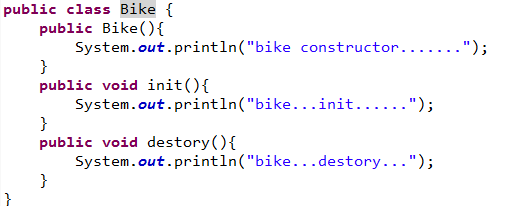
bean的生命周期是由容器进行管理的

我们可以自定义 bean初始化和销毁 方法: 容器在bean进行到当前生命周期的时候, 来调用自定义的初始化和销毁方法

如何定义和销毁(4种方式):

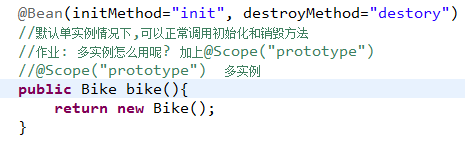
**1)** 指定初始化和销毁方法 <之前在beanx.xml, 可以指定init-method和destory-mothod>

用注释如何做: 新建Bike.java



1.1> 指定初始化和销毁方法

在配置类里通过@Bean(initMethod="init", destroyMethod="destroy")指定



1.2> 单实例: 当容器关闭的时候,会调用destroy消耗



多实例: 容器只负责初始化,但不会管理bean, 容器关闭不会调用销毁方法

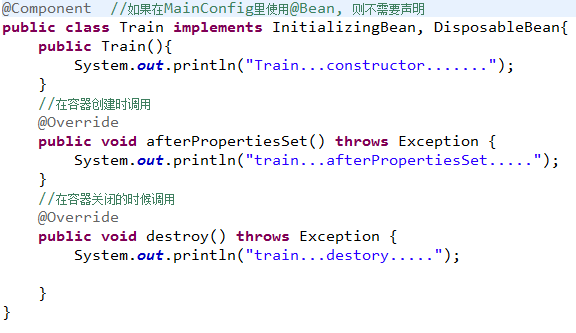
**2)** 让Bean实现 InitializingBean 和 DisposableBean接口

A, InitializingBean(定义初始化逻辑,可点进去看此类):afterPropertiesSet()方法:当beanFactory创建好对象,且把bean所有属性设置好之后,会调这个方法,相当于初始化方法

B, DisposableBean(定义销毁逻辑,可点进去看此类):destory()方法,当bean销毁时,会把单实例bean进行销毁

操作步骤:

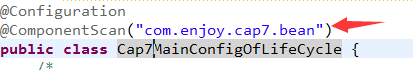
2.1> 新建Train.java类, 实现 InitializingBean, DisposableBean 接口



2.2> 加载bean方式

:@Bean public Train train()或@Component public class Train

或[在Config加上扫描@ComponentScan("com.enjoy.cap7.bean")](mailto:在Config加上扫描@ComponentScan(%22com.enjoy.cap12.bean%22)),用以下方式



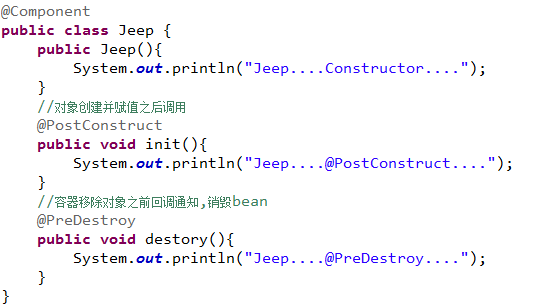
测试用例只要加载容器和关闭容器即可.

**3)** 可以使用JSR250规则定义的(java规范)两个注解来实现

@PostConstruct: 在Bean创建完成,且属于赋值完成后进行初始化,属于JDK规范的注解

@PreDestroy: 在bean将被移除之前进行通知, 在容器销毁之前进行清理工作

步骤:新建Jeep.java



三者调用顺序为:

1. Constructor
2. beanpostProcessor.postProcessBeforeInitialization
3. @PostConstrut
4. 实现InitialingBean的afterPropertiesSet
5. Inti-method xml配置

此时实例化

1. BeanPostProcessor.postProcessAfterInitializaion

销毁

1. @preDestory 注解
2. 实现DisposableBean的preDestory
3. Destroy-method xml配置

-----只有以上三种,以下是后置处理器,负责在初始化方法前后作用--------

BeanPostProcessor类[interface]: bean的后 置处理器,在bean初始化之前调用进行拦截

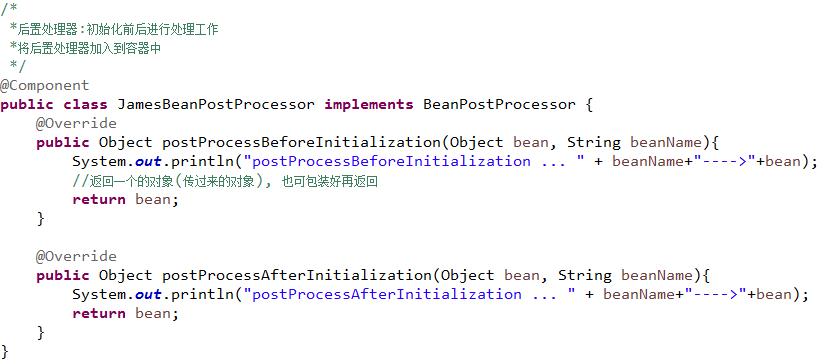
作用:在bean初始化前后进行一些处理工作, 打开此类

a> postProcessBeforeInitialization():在初始化之前进行后置处理工作(在init-method之前),

什么时候调用:它任何初始化方法调用之前(比如在InitializingBean的afterPropertiesSet初始化之前,或自定义init-method调用之前使用)

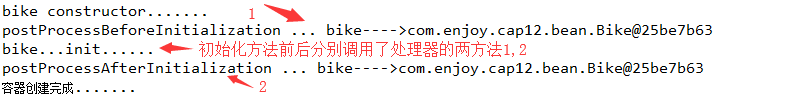
b> postProcessAfterInitialization():在初始化之后进行后置处理工作, 比如在InitializingBean的afterPropertiesSet()

步骤: 新建后置处理器类JamesBeanPostProcessor



总结:bean的整个生命周期我们都能控制

可以分析Bike的日志证明在init初始化之前调用了**postProcessBeforeInitialization(),**

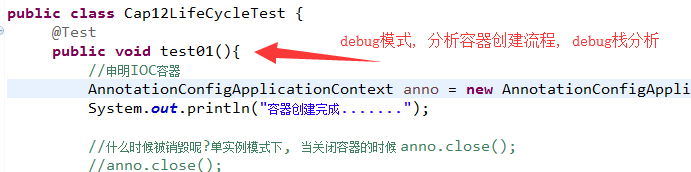


**容器启动及BeanPostProcessor源码分析**

针对以上cap12进行debug测试;



加完断点后, 测试用例debug, 分析容器创建流程



BeanPostProcessor原理:

可从容器类跟进顺序为:

AnnotationConfigApplicationContext-->refresh()-->

finishBeanFactoryInitialization(beanFactory)--->

beanFactory.preInstantiateSingletons()-->

760行getBean(beanName)--->

199行doGetBean(name, **null**, **null**, **false**)-->

317行createBean(beanName, mbd, args)-->

501行doCreateBean(beanName, mbdToUse, args)-->

541行createBeanInstance(beanName, mbd, args)(完成bean创建)-->

578行populateBean(beanName, mbd, instanceWrapper)(属性赋值)-->

579行initializeBean(beanName, exposedObject, mbd)(Bean初始化)->

1069行到1710行,后置处理器完成对init方法的前后处理.

最终得到如下如下

createBeanInstance(beanName, mbd, args)(完成bean创建)

populateBean(beanName, mbd, instanceWrapper); 给bean进行属性赋值

initializeBean() //初始化Bean方法内容如下,后置处理器对init方法的前后处理

{

applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);

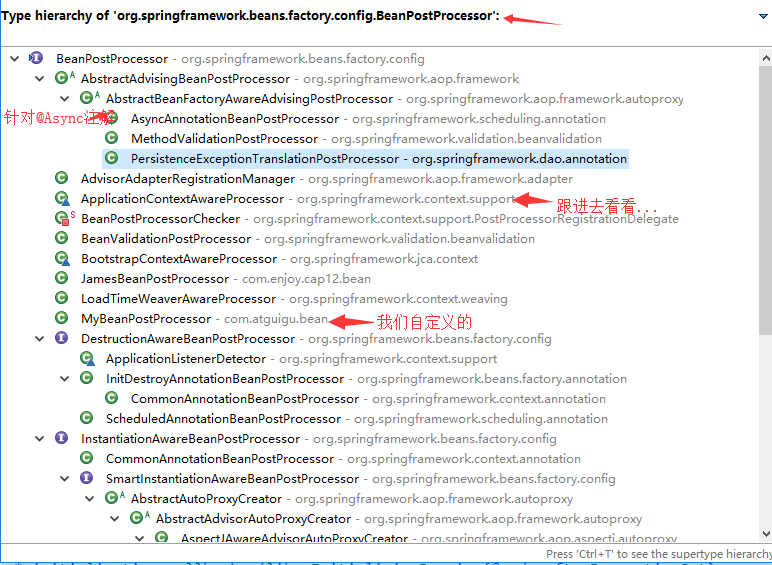
**invokeInitMethods(beanName, wrappedBean, mbd)** //执行自定义初始化

applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName)

}

从以上分析不难发现,bean的生命周期为bean的创建, 初始化, 当容器关闭时对单实例的bean进行销毁.

**Spring底层对BeanPostProcessor的使用**



**1,ApplicationContextAwareProcessor实现分析:**

此类帮我们组建IOC容器,跟进ApplicationContextAwareProcessor我们发现, 这个后置处理器其实就是判断我们的bean有没有实现ApplicationContextAware 接口,并处理相应的逻辑,其实所有的后置处理器原理均如此.

那么怎么组建呢? 只需要实现 ApplicationContextAware 接口

步骤:

1>, 新建Plane.java(将Jeep.java复制一份即可)

**class** Plane **implements** ApplicationContextAware



分析一下ApplicationContextAwareProcessor类的方法



a,在创建Plane对象,还没初始化之前, 先判断是不是实现了ApplicationContextAware接口,

如果是的话就调用invokeAwareInterfaces方法, 并给里面注入值;

b,进入invokeAwareInterfaces()方法,判断是哪个aware, 如果是ApplicationContextAware, 就将当前的bean转成ApplicationContextAware类型, 调用setApplicationContext(), 把IOC容器注入到Plane里去;

c,用debug调用; 测试用例打断点测试

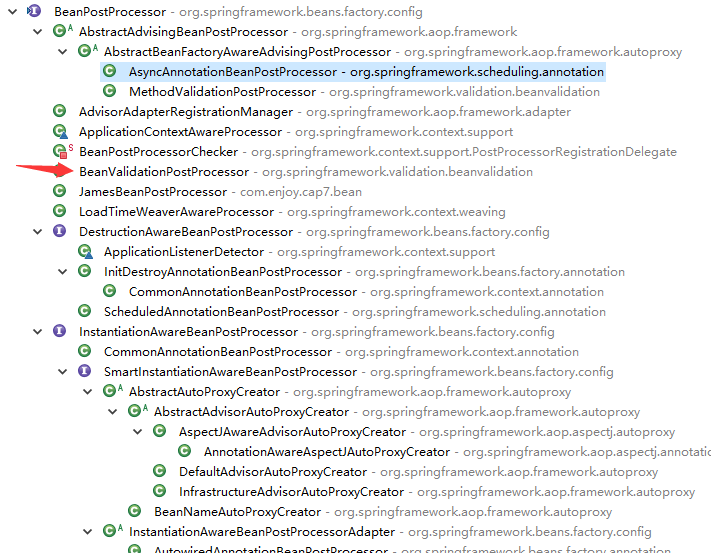


d,也可看debug调用栈来分析;

**注**: debug可以打在ApplicationContextAwareProcessor处理器类的applyBeanPostProcessorsBeforeInitialization()方法里, 方便调试, 当bean为Plane类型时,F5跟进看, 最终在 InvokeAwareInterfaces()方法里返回我们的IOC容器applicationContext.

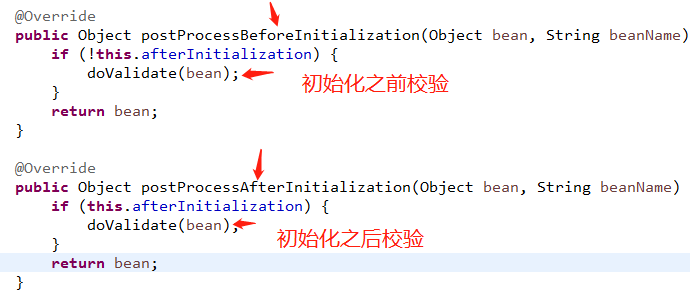
**BeanValidationPostProcess分析:数据校验,**

**看BeanPostProcessor接口实现CTRL+T**

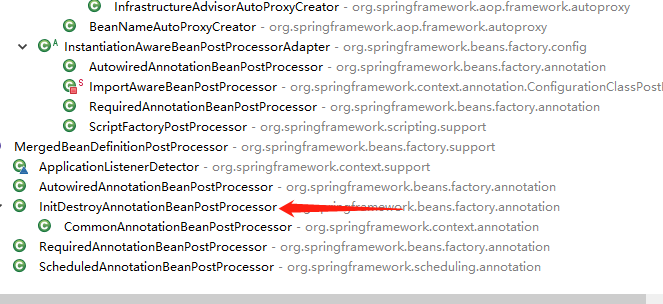


了解即可,处理器的原理和其它处理器一致.

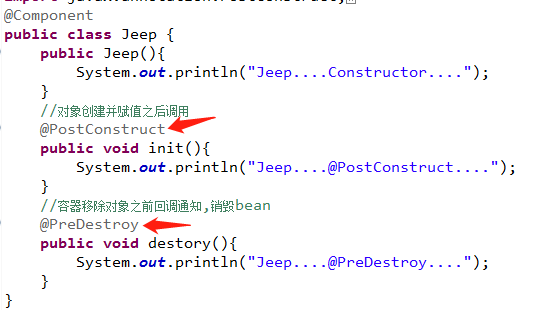
**当对象创建完,给bean赋值后,在WEB用得特别多;把页面提交的值进行校验**



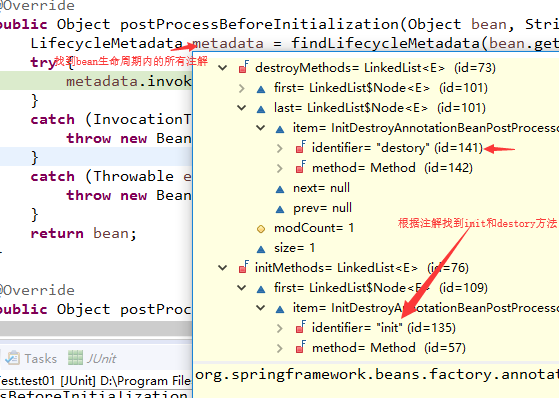
**3,InitDestroyAnnotationBeanPostProcessor**

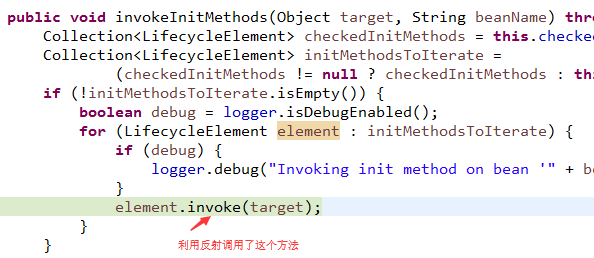


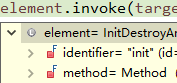
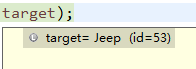
此处理器用来处理@PostConstruct, @PreDestroy, 怎么知道这两注解是前后开始调用的呢, 就是 **InitDestroyAnnotationBeanPostProcessor这个处理的**



以@PostConstruct为例, 为什么声明这个注解后就能找到初始化init方法呢?





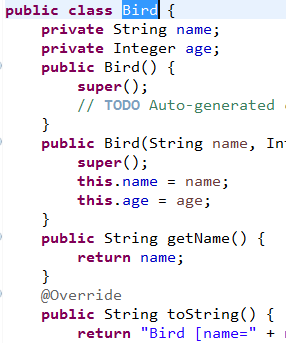
总结: Spring底层对BeanPostProcessor的使用, 包括bean的赋值, 注入其它组件, 生命周期注解功能,@Async, 等等

# 依赖实现

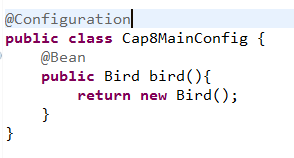
## @Value赋值

新建cap8目录;

1, 新建Bird.java类



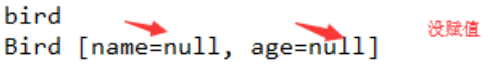
2,新建Cap8MainConfig.java配置类



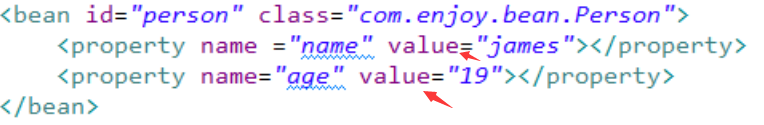
3,新建测试用例Cap8Test.java, 从容器获取bean并打印



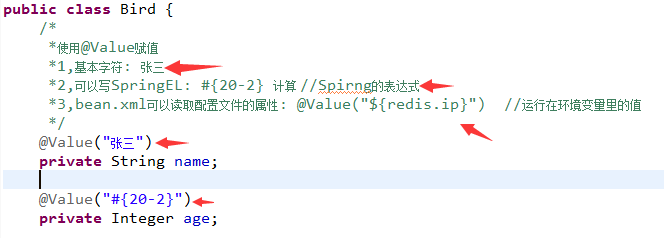
打印结果如下: 主要是没设值



4,以前使用bean.xml配置文件进行赋值



5,使用@Value赋值如何赋值呢?见下



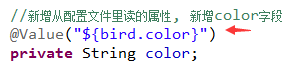
6,从配置文件[properties]读取, 新建test.properties



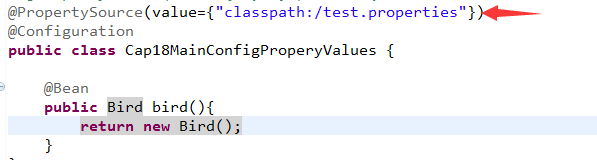
7,test.properties内容为bird.color=red



8,在Bird类新增private String color及set和get方法;



9,将test.properties配置文件加载起来



10,再运行test01()用例, 打印出以下结果



11,test.properties值是加在运行环境变量里:

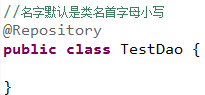
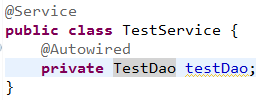
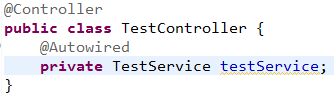


## @Autowired自动装配

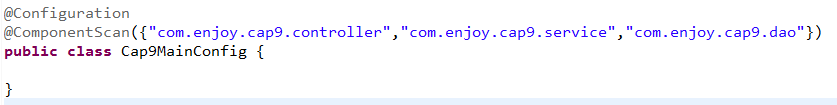
自动装配:spring利用依赖注入(DI), 完成对IOC容器中的各个组件的依赖关系赋值

1,新建TestController.java TestService.java TestDao; 分别建在指定的包内,可看步骤2.

这些所有JAVA 类的对象扫描后都是保存在IOC容器中管理的;

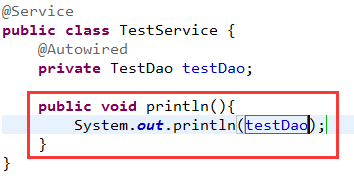
  

2,新建配置类Cap9MainConfig.java(),扫描并将以上bean都扫描并加载到容器



3, 针对以上基础类建立完成后, 可以先做个测试

在TestService.java, 使用Autowired注入,并把testDao打印出来(在测试时方便对比)



4, 新建Cap9Test.java测试用例,比较TestService拿到testDao与直接从容器中拿到的testDao是否为同一个?



结果很明显是同一个testDao,地址一样

**小结**:

@Autowired表示默认优先按**类型**去容器中找对应的组件,相当于anno.getBean(TestDao.class)去容器获取id为testDao的bean, 并注入到TestService的bean中;

使用方式如下:

TestService{

@Autowired

private TestDao testDao;//默认去容器中找id为”testDao”的bean

}

### 5, 注意事项

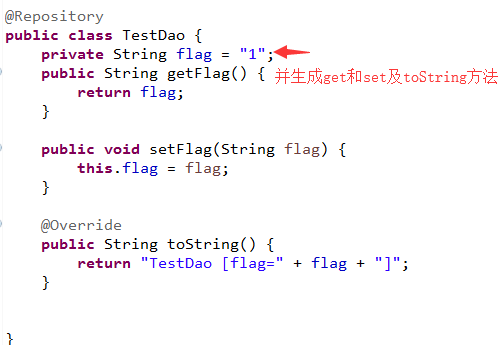
5.1如果容器中找到多个testDao, 会加载哪个testDao呢?

操作步骤:

在Cap9MainConfig.java声明@Bean(“testDao2”)



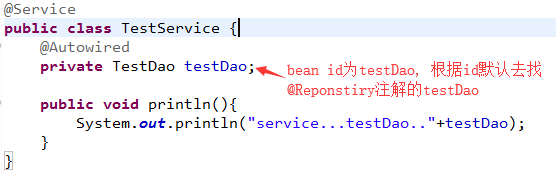
并将TestDao加入flag属性和set, get及toString方法,用来分辨加载了哪个bean.



如何区分TestService是使用了(**@Reponstry的testDao的flag=1**)的bean还是(**testDao2的flag=2**)的bean?

测试步骤如下:

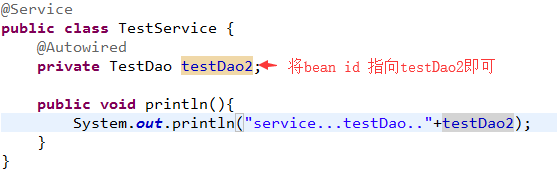
**1,**直接使用@Autowired, 将testDao注入到TestService



测试结果



**2,**如果一定要使用容器中的testDao2呢?操作如下:



测试结果



**3,**虽然以上定义了private TestDao testDao2, 但还是想加载bean id为testDao(flag=1)的bean,怎么办?此时可以使用@Autowired和@Qualifier结合来指定注入哪一个bean,

操作如下:

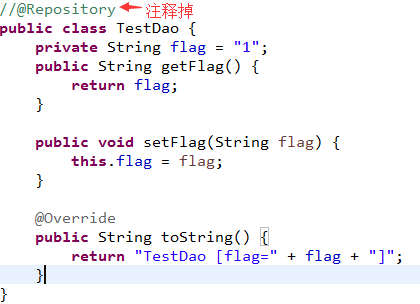
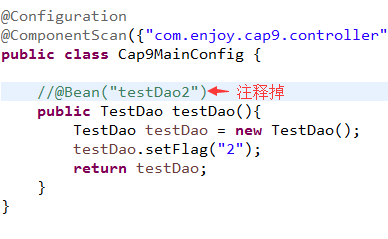


测试结果



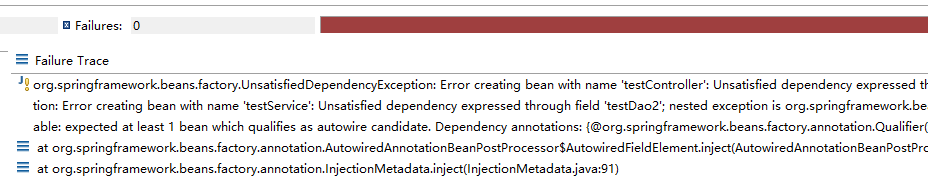
**4,**如果容器中没有任何一个testDao, 会出现什么状况呢?

操作如下: 注释掉@Repository和@Bean("testDao2")

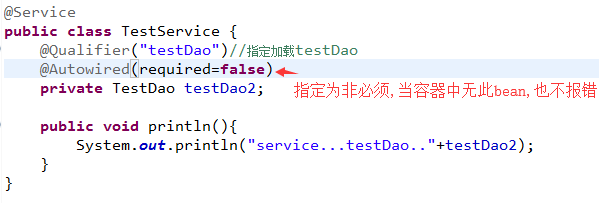
此时容器启动时这两个bean都不会加载(因为注解被注释啦.......)

测试结果如下:



很明显报错了, [因为@Autowired注解里的属性默认required=true.必须找到bean](mailto:因为@Autowired注解里的属性默认required=true.必须找到bean)

那怎么解决呢?

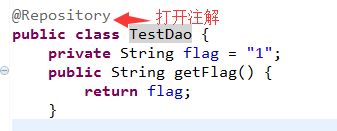


测试结果如下:



**5,**@Primary注解指定bean如何加载呢?

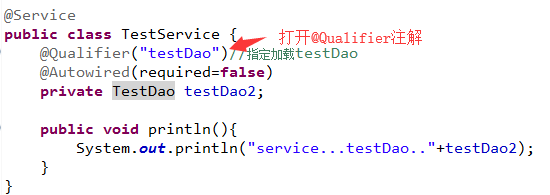
(**注**:将以上原注释掉的@Repository和@Bean("testDao2") 恢复,见下图)

**重要:为了验证@Qualifier与@Primary两注解的加载顺序,测试如下**

当对于testDao在容器中同时存在多个时, 且@Qualifier与@Primary注解同时存在,会发生什么呢?

见下操作: 打开@Qualifier与@Primary注解.

测试结果:

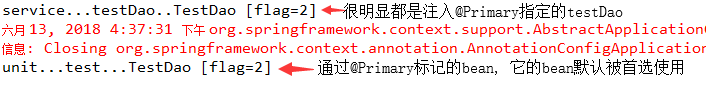


此时只能说明一点: @Qualifier是根据bean id指定获取testDao, 不受@Primary影响.

那么@Primary的功能在哪呢?继续测试.....



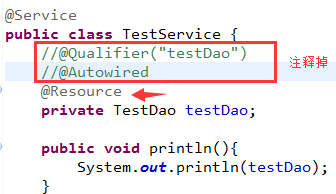
测试结果:



## @Resource

除了@Autowired, 是不是还用过@Resource(JSR250) 和@Inject(JSR330)

将Qualifier和Autowired注释掉(注意: 此时@Primary 还没注释......)



测试结果:



效果也是一样的, 但它不先优先装配@Primary的bean

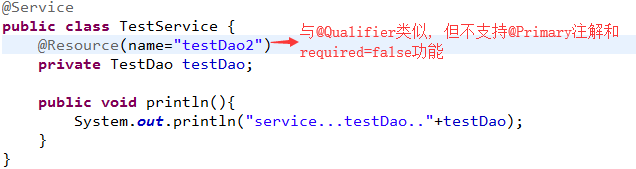
**小结:@Resource和Autowired的区别如下:**

@Resource和Autowired一样可以装配bean

@Resource缺点: 不能支持@Primary功能

不能支持@Autowired(required = false)的功能

当然也可以在TestService里按以下方式指定要注入的Bean



测试结果:



## @Inject自动装配的使用:

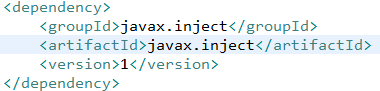
**注:@Inject与@Autowired的区别如下:**

@Inject和Autowired一样可以装配bean, 并支持@Primary功能, 可用于非spring框架.

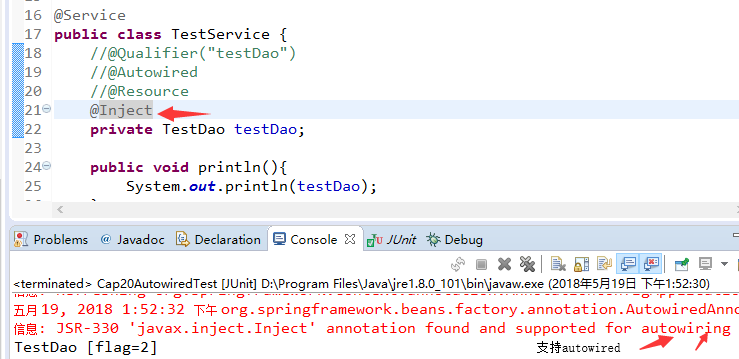
@Inject缺点: 但不能支持@Autowired(required = false)的功能,需要引入第三方包javax.inject

操作步骤:

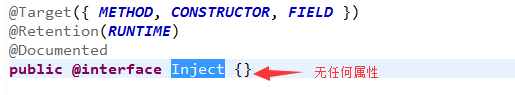
1,pom.xml导入javax.inject包



2,使用@Inject注解



结论:@Inject不支持required=false, 但支持primary



Autowired属于spring的, 不能脱离spring, @Resource和@Inject都是JAVA规范

推荐大家使用@Autowired