# 创建大致流程：以TestBeanAnnotation测试为准

**New** AnnotationConfigApplicationContext(FactoryBeanConfigration.**class**)—->refresh()-->finishBeanFactoryInitialization(beanFactory)—->beanFactory.preInstantiateSingletons()注意：调试到beanname是自己想要的那个F5进去—-> getBean(beanName) --> doGetBean(name, **null**, **null**, **false**)--> doCreateBean(beanName, mbdToUse, args)—-> createBean(beanName, mbd, args)注意：提前在后面的那个方法上加断点—-> doCreateBean(beanName, mbdToUse, args)--> createBeanInstance(beanName, mbd, args)--> instantiateUsingFactoryMethod(beanName, mbd, args)—->其他的方法调用全局配置文件中@bean型式的dog

## 单个流程：refresh

**public** **void** refresh() **throws** BeansException, IllegalStateException {

**synchronized** (**this**.startupShutdownMonitor) {

// Prepare this context for refreshing.---准备上下文环境，预刷新，检查刷新条件

prepareRefresh();

// Tell the subclass to refresh the internal bean factory.--- 告诉子类刷新内部bean工厂（上下文的配置解析bean）

ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

// Prepare the bean factory for use in this context.---准备bean工厂准备使用上下文环境

prepareBeanFactory(beanFactory);

**try** {

// Allows post-processing of the bean factory in context subclasses.--允许在上下文子类中对bean工厂进行后处理。

postProcessBeanFactory(beanFactory);

// Invoke factory processors registered as beans in the context.-- 调用在上下文中注册为bean的工厂处理器。

invokeBeanFactoryPostProcessors(beanFactory);

// Register bean processors that intercept bean creation.-- 注册bean处理程序，拦截bean的创建（Aopbean）

**registerBeanPostProcessors(beanFactory);**

// Initialize message source for this context.—为上下文初始化国际资源

initMessageSource();

// Initialize event multicaster for this context.--- 为这个上下文初始化事件多播器

initApplicationEventMulticaster();

// Initialize other special beans in specific context subclasses. -- 在特定上下文中初始化其他特殊bean，实际上是空的

需要自己实现 onRefresh();

// Check for listener beans and register them.-- 检查侦听器bean并注册它们

registerListeners();

// Instantiate all remaining (non-lazy-init) singletons.-- 实例化所有剩余的（非lazy-init）单例（普通bean）

**finishBeanFactoryInitialization(beanFactory);**

// Last step: publish corresponding event.-- 最后一步：发布相应的事件

finishRefresh();

}

**catch** (BeansException ex) {

**if** (logger.isWarnEnabled()) {

logger.warn("Exception encountered during context initialization - " +

"cancelling refresh attempt: " + ex);

}

// Destroy already created singletons to avoid dangling resources.

destroyBeans();

// Reset 'active' flag.

cancelRefresh(ex);

// Propagate exception to caller.

**throw** ex;

}

**finally** {

// Reset common introspection caches in Spring's core, since we

// might not ever need metadata for singleton beans anymore...

resetCommonCaches();

}

}

}

## 单个流程：preInstantiateSingletons

**public** **void** preInstantiateSingletons() **throws** BeansException {

**if** (**this**.logger.isDebugEnabled()) {

**this**.logger.debug("Pre-instantiating singletons in " + **this**);

}

// Iterate over a copy to allow for init methods which in turn register new bean definitions.

// While this may not be part of the regular factory bootstrap, it does otherwise work fine.

List<String> beanNames = **new** ArrayList<>(**this**.beanDefinitionNames);

// Trigger initialization of all non-lazy singleton beans...

**for** (String beanName : beanNames) {//包括后置处理器，注解bean，如：@aotuwired修饰的bean 全局配置bean等

RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);

**if** (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {

**if** (isFactoryBean(beanName)) {

Object bean = getBean(***FACTORY\_BEAN\_PREFIX*** + beanName);//eg：getBean(”&dog”)

**if** (bean **instanceof** FactoryBean) {

**final** FactoryBean<?> factory = (FactoryBean<?>) bean;

**boolean** isEagerInit;

**if** (System.*getSecurityManager*() != **null** && factory **instanceof** SmartFactoryBean) {

isEagerInit = AccessController.*doPrivileged*((PrivilegedAction<Boolean>)

((SmartFactoryBean<?>) factory)::isEagerInit,

getAccessControlContext());

}

**else** {

isEagerInit = (factory **instanceof** SmartFactoryBean &&

((SmartFactoryBean<?>) factory).isEagerInit());

}

**if** (isEagerInit) {

**getBean(beanName);**

}

}

}

**else** {

**getBean(beanName);**

}

}

}

// Trigger post-initialization callback for all applicable beans...

**for** (String beanName : beanNames) {

Object singletonInstance = getSingleton(beanName);

**if** (singletonInstance **instanceof** SmartInitializingSingleton) {

**final** SmartInitializingSingleton smartSingleton = (SmartInitializingSingleton) singletonInstance;

**if** (System.*getSecurityManager*() != **null**) {

AccessController.*doPrivileged*((PrivilegedAction<Object>) () -> {

smartSingleton.afterSingletonsInstantiated();

**return** **null**;

}, getAccessControlContext());

}

**else** {

smartSingleton.afterSingletonsInstantiated();

}

}

}

}

## 单个流程：doGetBean

**protected** <T> T doGetBean(**final** String name, @Nullable **final** Class<T> requiredType,

@Nullable **final** Object[] args, **boolean** typeCheckOnly) **throws** BeansException {

**final** String beanName = transformedBeanName(name);

Object bean;

// Eagerly check singleton cache for manually registered singletons.

Object sharedInstance = getSingleton(beanName);

**if** (sharedInstance != **null** && args == **null**) {

**if** (logger.isDebugEnabled()) {

**if** (isSingletonCurrentlyInCreation(beanName)) {

logger.debug("Returning eagerly cached instance of singleton bean '" + beanName +

"' that is not fully initialized yet - a consequence of a circular reference");

}

**else** {

logger.debug("Returning cached instance of singleton bean '" + beanName + "'");

}

}

bean = getObjectForBeanInstance(sharedInstance, name, beanName, **null**);

}

**else** {

// Fail if we're already creating this bean instance:

// We're assumably within a circular reference.

**if** (isPrototypeCurrentlyInCreation(beanName)) {

**throw** **new** BeanCurrentlyInCreationException(beanName);

}

// Check if bean definition exists in this factory.

BeanFactory parentBeanFactory = getParentBeanFactory();

**if** (parentBeanFactory != **null** && !containsBeanDefinition(beanName)) {

// Not found -> check parent.

String nameToLookup = originalBeanName(name);

**if** (parentBeanFactory **instanceof** AbstractBeanFactory) {

**return** ((AbstractBeanFactory) parentBeanFactory).doGetBean(

nameToLookup, requiredType, args, typeCheckOnly);

}

**else** **if** (args != **null**) {

// Delegation to parent with explicit args.

**return** (T) parentBeanFactory.getBean(nameToLookup, args);

}

**else** {

// No args -> delegate to standard getBean method.

**return** parentBeanFactory.getBean(nameToLookup, requiredType);

}

}

**if** (!typeCheckOnly) {

markBeanAsCreated(beanName);

}

**try** {

**final** RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);

checkMergedBeanDefinition(mbd, beanName, args);

// Guarantee initialization of beans that the current bean depends on.

String[] dependsOn = mbd.getDependsOn();

**if** (dependsOn != **null**) {

**for** (String dep : dependsOn) {

**if** (isDependent(beanName, dep)) {

**throw** **new** BeanCreationException(mbd.getResourceDescription(), beanName,

"Circular depends-on relationship between '" + beanName + "' and '" + dep + "'");

}

registerDependentBean(dep, beanName);

**try** {

getBean(dep);

}

**catch** (NoSuchBeanDefinitionException ex) {

**throw** **new** BeanCreationException(mbd.getResourceDescription(), beanName,

"'" + beanName + "' depends on missing bean '" + dep + "'", ex);

}

}

}

// Create bean instance.

**if** (mbd.isSingleton()) {

sharedInstance = getSingleton(beanName, () -> {

**try** {

**return** **createBean(beanName, mbd, args)**;

}

**catch** (BeansException ex) {

// Explicitly remove instance from singleton cache: It might have been put there

// eagerly by the creation process, to allow for circular reference resolution.

// Also remove any beans that received a temporary reference to the bean.

destroySingleton(beanName);

**throw** ex;

}

});

bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);

}

**else** **if** (mbd.isPrototype()) {

// It's a prototype -> create a new instance.

Object prototypeInstance = **null**;

**try** {

beforePrototypeCreation(beanName);

prototypeInstance = createBean(beanName, mbd, args);

}

**finally** {

afterPrototypeCreation(beanName);

}

bean = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);

}

**else** {

String scopeName = mbd.getScope();

**final** Scope scope = **this**.scopes.get(scopeName);

**if** (scope == **null**) {

**throw** **new** IllegalStateException("No Scope registered for scope name '" + scopeName + "'");

}

**try** {

Object scopedInstance = scope.get(beanName, () -> {

beforePrototypeCreation(beanName);

**try** {

**return** createBean(beanName, mbd, args);

}

**finally** {

afterPrototypeCreation(beanName);

}

});

bean = getObjectForBeanInstance(scopedInstance, name, beanName, mbd);

}

**catch** (IllegalStateException ex) {

**throw** **new** BeanCreationException(beanName,

"Scope '" + scopeName + "' is not active for the current thread; consider " +

"defining a scoped proxy for this bean if you intend to refer to it from a singleton",

ex);

}

}

}

**catch** (BeansException ex) {

cleanupAfterBeanCreationFailure(beanName);

**throw** ex;

}

}

// Check if required type matches the type of the actual bean instance.

**if** (requiredType != **null** && !requiredType.isInstance(bean)) {

**try** {

T convertedBean = getTypeConverter().convertIfNecessary(bean, requiredType);

**if** (convertedBean == **null**) {

**throw** **new** BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());

}

**return** convertedBean;

}

**catch** (TypeMismatchException ex) {

**if** (logger.isDebugEnabled()) {

logger.debug("Failed to convert bean '" + name + "' to required type '" +

ClassUtils.*getQualifiedName*(requiredType) + "'", ex);

}

**throw** **new** BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());

}

}

**return** (T) bean;

}

## 单个流程：createBean

**protected** Object createBean(String beanName, RootBeanDefinition mbd, @Nullable Object[] args)

**throws** BeanCreationException {

**if** (logger.isDebugEnabled()) {

logger.debug("Creating instance of bean '" + beanName + "'");

}

RootBeanDefinition mbdToUse = mbd;

// Make sure bean class is actually resolved at this point, and

// clone the bean definition in case of a dynamically resolved Class

// which cannot be stored in the shared merged bean definition.

Class<?> resolvedClass = resolveBeanClass(mbd, beanName);

**if** (resolvedClass != **null** && !mbd.hasBeanClass() && mbd.getBeanClassName() != **null**) {

mbdToUse = **new** RootBeanDefinition(mbd);

mbdToUse.setBeanClass(resolvedClass);

}

// Prepare method overrides.

**try** {

mbdToUse.prepareMethodOverrides();

}

**catch** (BeanDefinitionValidationException ex) {

**throw** **new** BeanDefinitionStoreException(mbdToUse.getResourceDescription(),

beanName, "Validation of method overrides failed", ex);

}

**try** {

// Give BeanPostProcessors a chance to return a proxy instead of the target bean instance.

**Object bean = resolveBeforeInstantiation(beanName, mbdToUse);**

**if** (bean != **null**) {

**return** bean;

}

}

**catch** (Throwable ex) {

**throw** **new** BeanCreationException(mbdToUse.getResourceDescription(), beanName,

"BeanPostProcessor before instantiation of bean failed", ex);

}

**try** {

Object beanInstance = **doCreateBean(beanName, mbdToUse, args);**

**if** (logger.isDebugEnabled()) {

logger.debug("Finished creating instance of bean '" + beanName + "'");

}

**return** beanInstance;

}

**catch** (BeanCreationException | ImplicitlyAppearedSingletonException ex) {

// A previously detected exception with proper bean creation context already,

// or illegal singleton state to be communicated up to DefaultSingletonBeanRegistry.

**throw** ex;

}

**catch** (Throwable ex) {

**throw** **new** BeanCreationException(

mbdToUse.getResourceDescription(), beanName, "Unexpected exception during bean creation", ex);

}

}

## 单个流程：doCreateBean

**protected** Object doCreateBean(**final** String beanName, **final** RootBeanDefinition mbd, **final** @Nullable Object[] args)

**throws** BeanCreationException {

// Instantiate the bean.

BeanWrapper instanceWrapper = **null**;

**if** (mbd.isSingleton()) {

instanceWrapper = **this**.factoryBeanInstanceCache.remove(beanName);

}

**if** (instanceWrapper == **null**) {

instanceWrapper = **createBeanInstance(beanName, mbd, args);**

}

**final** Object bean = instanceWrapper.getWrappedInstance();

Class<?> beanType = instanceWrapper.getWrappedClass();

**if** (beanType != NullBean.**class**) {

mbd.resolvedTargetType = beanType;

}

// Allow post-processors to modify the merged bean definition.

**synchronized** (mbd.postProcessingLock) {

**if** (!mbd.postProcessed) {

**try** {

applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);

}

**catch** (Throwable ex) {

**throw** **new** BeanCreationException(mbd.getResourceDescription(), beanName,

"Post-processing of merged bean definition failed", ex);

}

mbd.postProcessed = **true**;

}

}

// Eagerly cache singletons to be able to resolve circular references

// even when triggered by lifecycle interfaces like BeanFactoryAware.

**boolean** earlySingletonExposure = (mbd.isSingleton() && **this**.allowCircularReferences &&

isSingletonCurrentlyInCreation(beanName));

**if** (earlySingletonExposure) {

**if** (logger.isDebugEnabled()) {

logger.debug("Eagerly caching bean '" + beanName +

"' to allow for resolving potential circular references");

}

addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));

}

// Initialize the bean instance.

Object exposedObject = bean;

**try** {

**populateBean(beanName, mbd, instanceWrapper);**

exposedObject = **initializeBean(beanName, exposedObject, mbd);**

}

**catch** (Throwable ex) {

**if** (ex **instanceof** BeanCreationException && beanName.equals(((BeanCreationException) ex).getBeanName())) {

**throw** (BeanCreationException) ex;

}

**else** {

**throw** **new** BeanCreationException(

mbd.getResourceDescription(), beanName, "Initialization of bean failed", ex);

}

}

**if** (earlySingletonExposure) {

Object earlySingletonReference = getSingleton(beanName, **false**);

**if** (earlySingletonReference != **null**) {

**if** (exposedObject == bean) {

exposedObject = earlySingletonReference;

}

**else** **if** (!**this**.allowRawInjectionDespiteWrapping && hasDependentBean(beanName)) {

String[] dependentBeans = getDependentBeans(beanName);

Set<String> actualDependentBeans = **new** LinkedHashSet<>(dependentBeans.length);

**for** (String dependentBean : dependentBeans) {

**if** (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {

actualDependentBeans.add(dependentBean);

}

}

**if** (!actualDependentBeans.isEmpty()) {

**throw** **new** BeanCurrentlyInCreationException(beanName,

"Bean with name '" + beanName + "' has been injected into other beans [" +

StringUtils.*collectionToCommaDelimitedString*(actualDependentBeans) +

"] in its raw version as part of a circular reference, but has eventually been " +

"wrapped. This means that said other beans do not use the final version of the " +

"bean. This is often the result of over-eager type matching - consider using " +

"'getBeanNamesOfType' with the 'allowEagerInit' flag turned off, for example.");

}

}

}

}

// Register bean as disposable.

**try** {

registerDisposableBeanIfNecessary(beanName, bean, mbd);

}

**catch** (BeanDefinitionValidationException ex) {

**throw** **new** BeanCreationException(

mbd.getResourceDescription(), beanName, "Invalid destruction signature", ex);

}

**return** exposedObject;

}

## 单个流程：createBeanInstance

Root bean: class [null]; scope=singleton; abstract=false; lazyInit=false; autowireMode=3; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=configration; factoryMethodName=getDog1; initMethodName=null; destroyMethodName=(inferred); defined in com.crbank.base.bean.config.Configration mbd这个root Bean的信息 beanName=dog（beanId）

**protected** BeanWrapper createBeanInstance(String beanName, RootBeanDefinition mbd, @Nullable Object[] args) {

// Make sure bean class is actually resolved at this point.

Class<?> beanClass = resolveBeanClass(mbd, beanName);

**if** (beanClass != **null** && !Modifier.*isPublic*(beanClass.getModifiers()) && !mbd.isNonPublicAccessAllowed()) {

**throw** **new** BeanCreationException(mbd.getResourceDescription(), beanName,

"Bean class isn't public, and non-public access not allowed: " + beanClass.getName());

}

Supplier<?> instanceSupplier = mbd.getInstanceSupplier();

**if** (instanceSupplier != **null**) {

**return** obtainFromSupplier(instanceSupplier, beanName);

}

**if** (mbd.getFactoryMethodName() != **null**) {

**return** **instantiateUsingFactoryMethod(beanName, mbd, args)**;

}

// Shortcut when re-creating the same bean...

**boolean** resolved = **false**;

**boolean** autowireNecessary = **false**;

**if** (args == **null**) {

**synchronized** (mbd.constructorArgumentLock) {

**if** (mbd.resolvedConstructorOrFactoryMethod != **null**) {

resolved = **true**;

autowireNecessary = mbd.constructorArgumentsResolved;

}

}

}

**if** (resolved) {

**if** (autowireNecessary) {

**return** autowireConstructor(beanName, mbd, **null**, **null**);

}

**else** {

**return** instantiateBean(beanName, mbd);

}

}

// Need to determine the constructor...

Constructor<?>[] ctors = determineConstructorsFromBeanPostProcessors(beanClass, beanName);

**if** (ctors != **null** ||

mbd.getResolvedAutowireMode() == RootBeanDefinition.***AUTOWIRE\_CONSTRUCTOR*** ||

mbd.hasConstructorArgumentValues() || !ObjectUtils.*isEmpty*(args)) {

**return** autowireConstructor(beanName, mbd, ctors, args);

}

// No special handling: simply use no-arg constructor.

**return** instantiateBean(beanName, mbd);

}

# 初始化大致流程

在创建流程的doCreateBean中**initializeBean(beanName, exposedObject, mbd)就是初始化**

initializeBean(beanName, exposedObject, mbd)—- invokeInitMethods(beanName, wrappedBean, mbd)—((InitializingBean)bean).afterPropertiesSet()

## 单个流程initializeBean

**protected** Object initializeBean(**final** String beanName, **final** Object bean, @Nullable RootBeanDefinition mbd) {

**if** (System.*getSecurityManager*() != **null**) {

AccessController.*doPrivileged*((PrivilegedAction<Object>) () -> {

invokeAwareMethods(beanName, bean);

**return** **null**;

}, getAccessControlContext());

}

**else** {

invokeAwareMethods(beanName, bean);

}

Object wrappedBean = bean;

**if** (mbd == **null** || !mbd.isSynthetic()) {

wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);

}

**try** {

invokeInitMethods(beanName, wrappedBean, mbd);

}

**catch** (Throwable ex) {

**throw** **new** BeanCreationException(

(mbd != **null** ? mbd.getResourceDescription() : **null**),

beanName, "Invocation of init method failed", ex);

}

**if** (mbd == **null** || !mbd.isSynthetic()) {

wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);

}

**return** wrappedBean;

}

### 流程中的方法applyBeanPostProcessorsBeforeInitialization

把bean进行判断是否实现了aware接口，如果又实现对应的aware接口，则把容器中对应的组件赋值

### 流程中的方法invokeAwareMethods

**private** **void** invokeAwareMethods(**final** String beanName, **final** Object bean) {

**if** (bean **instanceof** Aware) {

**if** (bean **instanceof** BeanNameAware) {

((BeanNameAware) bean).setBeanName(beanName);

}

**if** (bean **instanceof** BeanClassLoaderAware) {

ClassLoader bcl = getBeanClassLoader();

**if** (bcl != **null**) {

((BeanClassLoaderAware) bean).setBeanClassLoader(bcl);

}

}

**if** (bean **instanceof** BeanFactoryAware) {

((BeanFactoryAware) bean).setBeanFactory(AbstractAutowireCapableBeanFactory.**this**);

}

}

}