**public Object** *getBean*(**String** name) **throws** BeansException {  
 **return** doGetBean(name, **null**, **null**, **false**);  
}

进入doGetBean()方法进行获取：

一 : 解析获取bean名称，比如别名、FactoryBean接口的bean

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

二 : 尝试从缓存中获取bean

**Object** sharedInstance = getSingleton(beanName);

1. 如果缓存中没有（第一次获取时肯定为null）
2. 如果容器中没有，则到父容器中去获取，也是调用getBean()
3. 容器中有：

1.1：xml中定义了depends-on属性的bean

String[] dependsOn = mbd.getDependsOn();  
**if** (dependsOn != **null**) {  
 **for** (String dependsOnBean : dependsOn) {  
 getBean(dependsOnBean);  
 registerDependentBean(dependsOnBean, beanName);  
 }  
}

1.2：调用getSingleton(beanName, objectFactory)

**if** (mbd.isSingleton()) {  
 sharedInstance = getSingleton(beanName, **new ObjectFactory**<**Object**>() {  
 **public Object** *getObject*() **throws** BeansException {  
 **try** {  
 **return** createBean(beanName, mbd, args);  
 }  
 **catch** (BeansException ex) {  
 destroySingleton(beanName);  
 **throw** ex;  
 }  
 }  
 });  
 bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);  
}

-----------------------------------------------------------------------------------

beforeSingletonCreation(beanName);//记录单例正在创建中的状态，如下

**protected void** beforeSingletonCreation(**String** beanName) {  
 **if** (!**this**.inCreationCheckExclusions.containsKey(beanName) &&  
 **this**.singletonsCurrentlyInCreation.put(beanName, **Boolean**.*TRUE*) != **null**) {  
 **throw new** BeanCurrentlyInCreationException(beanName);  
 }  
}

----------------------------------------------------------------------------

singletonObject = singletonFactory.getObject();//回调方法进行实际创建过程

afterSingletonCreation(beanName);//去除该单例正在创建状态

addSingleton(beanName, singletonObject);

**protected void** addSingleton(**String** beanName, **Object** singletonObject) {  
 **synchronized** (**this**.singletonObjects) {  
 **this**.singletonObjects.put(beanName, (singletonObject != **null** ? singletonObject : *NULL\_OBJECT*));//放入缓存中

**this**.singletonFactories.remove(beanName);//移除提前暴露的bean工厂，是在//singletonFactory.getObject()中放入的  
 **this**.earlySingletonObjects.remove(beanName);//移除提前暴露的bean  
 **this**.registeredSingletons.add(beanName);//注册单例  
 }  
}

1.3：singletonFactory.getObject();方法执行创建

// Give BeanPostProcessors a chance to return a proxy instead of the target bean instance.  
**Object** bean = resolveBeforeInstantiation(beanName, mbd);//调用了 //ibp.postProcessBeforeInstantiation(beanClass, beanName)，如果不为null，则继续调用**postProcessAfterInstantiation，否则在实例化后的popularBean方法中调用实例化后处理器 postProcessAfterInstantiation**  
**if** (bean != **null**) {  
 **return** bean;  
}

如果没有动态代理，则调用如下方法：

**Object** beanInstance = doCreateBean(beanName, mbd, args);// Instantiate the bean.

instanceWrapper = createBeanInstance(beanName, mbd, args);//实例化bean，产生bean实例

//解析@AutoWired、@Value  
**synchronized** (mbd.postProcessingLock) {  
 **if** (!mbd.postProcessed) {  
 applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);  
 mbd.postProcessed = **true**;  
 }  
}

//用户检测循环依赖，提前暴露bean的工厂，初始化之前暴露  
**boolean** earlySingletonExposure = (mbd.isSingleton() && **this**.allowCircularReferences &&  
 isSingletonCurrentlyInCreation(beanName));  
**if** (earlySingletonExposure) {  
 addSingletonFactory(beanName, **new ObjectFactory**<**Object**>() {  
 **public Object** *getObject*() **throws** BeansException {  
 **return** getEarlyBeanReference(beanName, mbd, bean);//该方法中，判断如果需要代理的话，会返回代理对象。  
 }  
 });  
}

// Initialize the bean instance. Bean已经实例化，下一步即初始化(设置属性值等)

populateBean(beanName, mbd, instanceWrapper);

-------------------------------------------------------------------------------

**for** (**BeanPostProcessor** bp : getBeanPostProcessors()) {  
 **if** (bp **instanceof InstantiationAwareBeanPostProcessor**) {  
 **InstantiationAwareBeanPostProcessor** ibp=(**InstantiationAwareBeanPostProcessor**) bp;  
 **if** (**!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)**) {  
 continueWithPropertyPopulation = **false**;  
 **break**;  
 }  
 }  
}

/////下一步是autowiredByName和autoWiredByType，这两个方法中继续调用beanFactory的getBean()获取依赖注入的bean，如果这个依赖的bean中和该bean存在循环依赖，则会从提前暴露的bean工厂中获取该bean的实例，实现循环依赖。

populateBean(beanName, mbd, instanceWrapper)方法完了之后调用下面的方法

exposedObject = initializeBean(beanName, exposedObject, mbd);

invokeAwareMethods(beanName, bean); //1 调用实现Aware接口的setXXXAware()方法

Object wrappedBean = bean; //2 调用postProcessBeforeInitialization  
**if** (mbd == **null** || !mbd.isSynthetic()) {  
 wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);  
}//调用BeanPostProcessor的postProcessBeforeInitialization(result, beanName)方法

invokeInitMethods(beanName, wrappedBean, mbd);//3 调用init-method

wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);//4调用BeanPostProcessor的beanProcessor.postProcessAfterInitialization()方法

1. 如果缓存中有（不解释）

Spring中扩展点:BeanFactoryPostProcessor及BeanPostProcessor