

Spring源码解析——bean的加载（四）

一、概述

1.1> doGetBean(...)

1.2> doGetBean(...)

二、FactoryBean的用法

三、getSingleton(beanName)

四、getObjectForBeanInstance(...)

五、getSingleton(beanName, singletonFactory)

六、createBean(...)

七、循环依赖

八、doCreateBean(...)

8.1> 概述

8.2> createBeanInstance()创建bean的实例

8.2.1> autowireConstructor(...)有参数的实例化构造

8.2.2> instantiateBean(...)无参数的实例化构造

8.2.3> instantiate(...)

8.3> getEarlyBeanReference(...)记录创建bean的ObjectFactory

8.4> populateBean(...)属性注入

8.4.1> autowireByName(...)根据名称进行注入

8.4.2> autowireByType(...)根据类型进行注入

8.4.3> applyPropertyValues(...)

8.5> initializeBean(...)初始化bean

8.5.1> invokeAwareMethods(...)激活Aware方法

8.5.2> invokeInitMethods(...)激活自定义的init方法

8.6> registerDisposableBeanIfNecessaryn(...)注册DisposableBean

一、概述

- 在前几讲中，我们着重地分析了Spring对xml配置文件的[解析](#)和[注册](#)过程。那么，本节内容，将会

试图分析一下bean的加载过程。具体代码，如下图所示：

```
BeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("oldbean.xml"));
User user = (User) beanFactory.getBean("user");
System.out.println("user = " + user);
```

```
AbstractBeanFactory.java
206      @Override
207      public Object getBean(String name) throws BeansException {    name: "user"
208          return doGetBean(name, null, null, false);    name: "user"
209      }
```

1.1> doGetBean(...)

- 针对bean的创建和加载，我们可以看出来逻辑都是在 `doGetBean(...)` 这个方法中的，所以，如下就是针对于这个方法的整体源码注释：

code

```
1  @SuppressWarnings("unchecked")
2  protected <T> T doGetBean(String name, Class<T> requiredType, @Nullable Object[] args, boolean typeCheckOnly) {
3      String beanName = transformedBeanName(name); // 提取真正的beanName
        (去除'&'或者将别名name转化为beanName)
4
5      /** 步骤1: 尝试根据beanName, 从缓存中获得单例对象 */
6      Object beanInstance, sharedInstance = getSingleton(beanName);
7      if (sharedInstance != null && args == null)
8          beanInstance = getObjectForBeanInstance(sharedInstance, name,
        beanName, null); // 从bean中获得真正的实例对象
9
10     /** 步骤2: 缓存中不存在实例, 则采取自主创建实例对象 */
11     else {
12         // 如果【原型模式】出现循环依赖, 则无法处理, 直接抛出异常
13         if (isPrototypeCurrentlyInCreation(beanName)) throw new BeanC
        urrentlyInCreationException(beanName);
14
15         /** 步骤3: 如果存在parentBeanFactory, 并且配置中也没有beanName的配置
        信息, 则尝试从parentBeanFactory中获取实例 */
16         BeanFactory parentBeanFactory = getParentBeanFactory();
17         if (parentBeanFactory != null && !containsBeanDefinition(bean
        Name)) {
18             String nameToLookup = originalBeanName(name);
19             if (parentBeanFactory instanceof AbstractBeanFactory)
20                 return ((AbstractBeanFactory) parentBeanFactory).doGe
        tBean(nameToLookup, requiredType, args, typeCheckOnly);
21             else if (args != null)
22                 return (T) parentBeanFactory.getBean(nameToLookup, ar
        gs);
23             else if (requiredType != null)
24                 return parentBeanFactory.getBean(nameToLookup, requir
        edType);
25             else
26                 return (T) parentBeanFactory.getBean(nameToLookup);
27         }
28
29         if (!typeCheckOnly) markBeanAsCreated(beanName); // 如果不执行
        类型检查, 则将beanName保存到alreadyCreated中
30         StartupStep beanCreation = this.applicationStartup.start("spr
        ing.beans.instantiate").tag("beanName", name);
31         try {
32             if (requiredType != null) beanCreation.tag("beanType", re
        quiredType::toString);
```

```

33
34         /** 步骤4: 将GenericBeanDefinition转换为RootBeanDefinition,
如果是子Bean, 则与父类的相关属性进行合并 */
35         RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
36
37         checkMergedBeanDefinition(mbd, beanName, args);
38
39         /** 步骤5: 如果存在依赖, 那么需要递归每一个依赖的bean并对其进行实例
化创建 */
40         String[] dependsOn = mbd.getDependsOn();
41         if (dependsOn != null) {
42             for (String dep : dependsOn) {
43                 // 如果发生了循环依赖, 则直接抛出异常
44                 if (isDependent(beanName, dep)) throw new BeanCreationException(...);
45                 registerDependentBean(dep, beanName); // 缓存依赖调用
46             }
47             try {
48                 getBean(dep); // 创建每一个依赖 (dep) 的实例Bean
49             } catch (NoSuchBeanDefinitionException ex) {throw
new BeanCreationException(...);}
50         }
51
52         /** 步骤6: 创建单例对象 */
53         if (mbd.isSingleton()) {
54             sharedInstance = getSingleton(beanName, () -> {
55                 try {
56                     return createBean(beanName, mbd, args); // 创建Bean实例对象
57                 } catch (BeansException ex) {destroySingleton(beanName); throw ex;}
58             });
59             beanInstance = getObjectForBeanInstance(sharedInstance, name, beanName, mbd); // 获得真正的bean
60         }
61
62         /** 步骤7: 创建原型对象 */
63         else if (mbd.isPrototype()) {
64             Object prototypeInstance = null;
65             try {
66                 beforePrototypeCreation(beanName);
67                 prototypeInstance = createBean(beanName, mbd, args); // 创建Bean实例对象
68             } finally {
69                 afterPrototypeCreation(beanName);
70             }
71             beanInstance = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd); //获得真正的bean

```

```

70     }
71     /** 步骤8: 创建指定scope类型的对象 */
72     else {
73         String scopeName = mbd.getScope();
74         if (!StringUtils.hasLength(scopeName)) throw new IllegalStateException(...);
75
76         Scope scope = this.scopes.get(scopeName);
77         if (scope == null) throw new IllegalStateException(...);
78
79         try {
80             Object scopedInstance = scope.get(beanName, () -
81 > {
82             beforePrototypeCreation(beanName);
83             try {
84                 return createBean(beanName, mbd, args);
85             } finally {
86                 afterPrototypeCreation(beanName);
87             }
88             });
89             beanInstance = getObjectForBeanInstance(scopedInstance, name, beanName, mbd); //获得真正的bean
90         } catch (IllegalStateException ex) {throw new ScopeNotActiveException(...);}
91     }
92 }
93 catch (BeansException ex) {
94     beanCreation.tag("exception", ex.getClass().toString());
95     beanCreation.tag("message", String.valueOf(ex.getMessage()));
96 });
97 cleanupAfterBeanCreationFailure(beanName);
98 throw ex;
99 }
100 finally {
101     beanCreation.end();
102 }
103 }
104
105 /** 步骤9: 检查需要的类型是否符合bean的实际类型, 如果不同, 则对其进行类型转换 */
106 return adaptBeanInstance(name, beanInstance, requiredType);
107 }

```

1.2> doGetBean(...)

通过上面针对doGetBean(...)方法的源码注释，我们可以将其主要的流程总结一下：

- 1：对beanName进行解析和转换——transformedBeanName(name)

- 第1步：去除FactoryBean的修饰符“&”，因为如果beanName是以“&”开头的，则表明是FactoryBean。所以需要去掉“&”前缀。
- 第2步：如果beanName传入的是alias值，则通过 aliasMap 获取真正的beanName。

- 2：尝试从缓存中获取单例实例——getSingleton(beanName)

- 因为单例在Spring的同一个容器内只会被创建一次，后续再获取bean，就直接从单例缓存 singletonObjects 中获取了。所以，首先会尝试从缓存中加载bean，如果加载不到，再尝试从 singletonFactories 中加载。
- 因为在创建单例bean的时候会存在依赖注入的情况，而在创建以来的时候，为了避免循环依赖，所以Spring不等bean创建完成就会将创建bean的 ObjectFactory 提早曝光加入到缓存中，一旦另外的bean创建时候需要依赖这个bean的时候，则直接使用 ObjectFactory#getObject() 方法来获得单例实例。具体逻辑如下所示：
 - 第1步：尝试从 singletonObjects 中获得单例；
 - 第2步：如果当前 beanName 所对应的实例正处于创建中，则尝试从 earlySingletonObjects 中获得单例；
 - 第3步：尝试从 singletonFactories 中获得 ObjectFactory 对象，然后通过调用 getObject() 方法获得单例；

- 3：bean的实例化——getObjectForBeanInstance(...)

- 其实我们从缓存中获得的是bean的原始状态，并不一定是我们最终想要的bean。比如：我们需要对工厂bean进行处理，那么这里得到的其实是工厂bean的初始状态，而我们真正需要的是工厂bean中定义的 factory-method 方法中返回的bean，那么 getObjectForBeanInstance 就可以完成这样的工作。

- 4：原型模式的依赖检查——isPrototypeCurrentlyInCreation(beanName)

- 只有单例才可以解决循环依赖，而原型模式如果发生了循环依赖，则直接抛异常。

- 5：parentBeanFactory相关逻辑处理——getParentBeanFactory()

- 如果存在parentBeanFactory，并且当前所加载的XML配置信息中不包含beanName，那么我们就只能通过 parentBeanFactory#getBean() 方法来获得beanName对应的实例对象。

- 6：将GenericBeanDefinition转换为RootBeanDefinition——getMergedLocalBeanDefinition(beanName)

- 因为从XML配置文件中读取到的bean信息是存储在GenericBeanDefinition中的，但是后续的所有bean处理都是针对RootBeanDefinition的，所以这里需要进行一下类型转换，在转换的同时，如果父类bean不为空的话，那么会合并父类的属性。
- 7: 针对所有依赖的bean执行初始化操作——mbd.getDependsOn()
 - 在Spring的加载顺序中，初始化一个bean的时候，首先优先初始化这个bean所对应的所有依赖。
- 8: 针对不同的scope进行bean的创建——createBean(beanName, mbd, args)
 - 此处会针对单例（Singleton）、原型（Prototype）和其他scope进行不同的初始化策略。但是最终都是会调用 createBean方法 来创建bean。
- 9: 类型转换——adaptBeanInstance(name, beanInstance, requiredType)
 - 只有当requiredType不为null的时候，才会执行类型转换。而我们调用的 getBean(String name) 方法中所调用的 doGetBean(name, null, null, false) 方法的第2个参数，就是 requiredType，而这里硬编码传入的就是null，所以不会执行类型转换操作。
 - 但是如果requiredType传入了一个类型，与bean的类型不同，则要执行类型转换操作。在Spring中提供了各种各样的转换器，用户也可以自己扩展转换器来满足需求。

二、FactoryBean的用法

- 如果在某些情况下，实例化bean的过程比较复杂，如果在<bean>中进行配置的话，需要提供大量的配置信息，这种情况下的配置就失去了灵活性。所以，此时我们可以采取编码的方式来实例化这个bean，即：通过实现 FactoryBean接口，在 getObject() 方法中去实现bean的创建过程。那么当Spring发现配置文件中<bean>的class属性配置的实现类是FactoryBean的子类时，就会通过调用 FactoryBean#getObject() 方法返回bean的实例对象。如下是演示例子：

▼

Car.java

Java

复制代码

```

1  @Data
2  @AllArgsConstructor
3  @NoArgsConstructor
4  public class Car {
5      private int maxSpeed;
6      private String brand;
7      private double price;
8  }

```

```

1 public interface FactoryBean<T> {
2     String OBJECT_TYPE_ATTRIBUTE = "factoryBeanObjectType";
3
4     @Nullable
5     T getObject() throws Exception;
6
7     @Nullable
8     Class<?> getObjectType();
9
10    default boolean isSingleton() {
11        return true;
12    }
13 }

```

```

@Data
public class CarFactoryBean implements FactoryBean<Car> {
    // 存储car的信息
    private String carInfo;

    @Override
    public Car getObject() {
        String[] carInfoArr = carInfo.split(",");
        Car car = new Car(Integer.valueOf(carInfoArr[0]), carInfoArr[1], Double.valueOf(carInfoArr[2]));
        return car;
    }

    @Override
    public Class<?> getObjectType() { return Car.class; }

    @Override
    public boolean isSingleton() { return FactoryBean.super.isSingleton(); }
}

```

```

<bean id="car" class="com.muse.springbootdemo.entity.factorybean.CarFactoryBean">
    <property name="carInfo" value="280, 奥迪A4L, 300000"/>
</bean>

```

```

13 @Slf4j
14 @SpringBootApplication
15 public class SpringbootDemoApplication {
16     public static void main(String[] args) throws Throwable {
17         XmlBeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("oldbean.xml"));
18
19         // 演示factoryBean
20         Car car = ((Car) beanFactory.getBean("car"));
21         System.out.println("car1 = " + car);
22
23         CarFactoryBean carFactoryBean = ((CarFactoryBean) beanFactory.getBean("&car"));
24         System.out.println("car2 = " + carFactoryBean.getObject());
25     }
}

```

Console Endpoints

```

/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...
09:09:34.768 [main] DEBUG org.springframework.beans.factory.xml.XmlBeanDefinitionReader - Loaded 4 bean definitions from
09:09:34.772 [main] DEBUG org.springframework.beans.factory.xml.XmlBeanFactory - Creating shared instance of singleton
car1 = Car(maxSpeed=280, brand= 奥迪A4L, price=300000.0)
car2 = Car(maxSpeed=280, brand= 奥迪A4L, price=300000.0)

Process finished with exit code 0

```


- 如上面例子所示，当我们需要获取Car实例对象时，通过调用`getBean("car")`即可；那么，如果我们就是想要获得CarFactoryBean的实例对象，则可以通过调用`getBean("&car")`即可。

三、getSingleton(beanName)

- 由于单例在Spring容器中只会被创建一次，即：创建出来的单例实例对象就会被缓存到 `singletonObjects` 中。所以，当要获得某个beanName的实例对象时，会首先尝试从 `singletonObjects` 中加载，如果加载不到，则再尝试从 `singletonFactories` 中加载。
- 因为在创建单例bean的时候可能会存在依赖注入的情况，所以为了避免循环依赖，Spring创建bean的原则是不等bean创建完成就会将创建bean的 `ObjectFactory` 提早曝光加入到缓存 `singletonFactories` 中，一旦下一个bean创建时需要依赖上一个bean，则直接使用 `ObjectFactory`。具体代码逻辑，请见下图所示：

```

167 public Object getSingleton(String beanName) {
168     return getSingleton(beanName, true);
169 }
170     允许早期依赖
171
172 /** Return the (raw) singleton object registered under the given name. ... */
173 @Nullable
174 protected Object getSingleton(String beanName, boolean allowEarlyReference) {
175     // Quick check for existing instance without full singleton lock
176     Object singletonObject = this.singletonObjects.get(beanName);  尝试从缓存中获取实例对象
177     if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {
178         singletonObject = this.earlySingletonObjects.get(beanName);  如果 bean 正在创建中，则获取实例对象
179         if (singletonObject == null && allowEarlyReference) {
180             synchronized (this.singletonObjects) {
181                 // Consistent creation of early reference within full singleton lock
182                 singletonObject = this.singletonObjects.get(beanName);
183                 if (singletonObject == null) {
184                     singletonObject = this.earlySingletonObjects.get(beanName);
185                     if (singletonObject == null) {
186                         尝试获得 ObjectFactory
187                         ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);
188                         if (singletonFactory != null) {
189                             singletonObject = singletonFactory.getObject();  获得实例对象
190                             this.earlySingletonObjects.put(beanName, singletonObject);
191                             this.singletonFactories.remove(beanName);
192                         }
193                     }
194                 }
195             }
196         }
197     }
198     return singletonObject;
199 }
200
201 }
202
203 }
204 }

```

- `singletonObjects`：用于保存beanName和bean实例之间的关系。
- `singletonFactories`：用于保存beanName和创建bean的工厂之间的关系。
- `earlySingletonObjects`：用于保存beanName和bean实例之间的关系。与singletonObjects的不同之处在于，当一个单例bean被放到这里面后，那么当bean还在创建过程中，就可以通过 `getBean` 方法 获取到了，其目的是用来检测循环引用。

- registeredSingletons：用来保存当前所有已注册的bean。

四、getObjectForBeanInstance(...)

- 当我们得到bean的实例之后，要做的第一步就是调用 `getObjectForBeanInstance(...)` 方法来检测正确性，即：[检测当前bean是否是FactoryBean类型](#)，如果是，那么需要调用它的 `getObject()` 方法作为返回值。源码注释如下所示：

code

```
1  Object getObjectForBeanInstance(Object beanInstance, String name, String beanName, RootBeanDefinition mbd) {
2      /** 步骤1: 如果name是以"&"开头的, 则表示就是要返回FactoryBean的实例对象, 不需要处理, 直接返回即可 */
3      if (BeanFactoryUtils.isFactoryDereference(name)) {
4          if (beanInstance instanceof NullBean) return beanInstance;
5          if (!(beanInstance instanceof FactoryBean)) throw new BeansInFactoryException(...);
6          if (mbd != null) mbd.isFactoryBean = true;
7          return beanInstance;
8      }
9
10     /** 步骤2: 如果beanInstance不是FactoryBean类型的实例对象, 则不需要处理, 直接返回即可 */
11     if (!(beanInstance instanceof FactoryBean)) return beanInstance;
12
13     /** 步骤3: beanInstance是FactoryBean类型的实例对象, 则调用getObject()方法获取的真实的bean对象 */
14     Object object = null;
15     if (mbd != null) mbd.isFactoryBean = true;
16     else object = getCacheObjectForFactoryBean(beanName); // 首先尝试从缓存中获得bean
17     if (object == null) {
18         FactoryBean<?> factory = (FactoryBean<?>) beanInstance;
19         // 将存储XML配置信息的GenericBeanDefinition转换为RootBeanDefinition, 如果指定beanName是子bean, 则合并父类属性
20         if (mbd == null && containsBeanDefinition(beanName)) mbd = getMergedLocalBeanDefinition(beanName);
21         boolean synthetic = (mbd != null && mbd.isSynthetic()); // true: 用户自定义的 false: 应用程序定义的
22         object = getObjectFromFactoryBean(factory, beanName, !synthetic); // 调用getObject()方法获取的真实的bean对象
23     }
24     return object;
25 }
```

- 上面代码比较简单, 大多是一些辅助代码以及一些功能性的判断, 而真正的核心代码是 `getObjectFromFactoryBean(factory, beanName, !synthetic)`, 下面我们来着重分析一下这个方法, 源码注释如下所示:

```
1  protected Object getObjectFromFactoryBean(FactoryBean<?> factory, String beanName, boolean shouldPostProcess) {
2      /**
3       * 步骤1: 如果factory是单例的, 并且beanName对应的bean已经被创建了;
4       * 如果没有创建缓存, 则向缓存factoryBeanObjectCache中添加beanName与Bean实例对象的对应关系;
5       */
6      if (factory.isSingleton() && containsSingleton(beanName)) {
7          synchronized (getSingletonMutex()) {
8              Object object = this.factoryBeanObjectCache.get(beanName);
9              // 尝试从缓存中获得Bean实例对象
10             if (object == null) {
11                 // 创建bean的实例对象。即: 调用FactoryBean#getObject()获得
12                 object = doGetObjectFromFactoryBean(factory, beanName)
13             ;
14
15             Object alreadyThere = this.factoryBeanObjectCache.get(
16                 beanName);
17             if (alreadyThere != null) object = alreadyThere;
18             else {
19                 if (shouldPostProcess) { // 是否执行后置处理 (xxxPost
20                     Process)
21                     if (isSingletonCurrentlyInCreation(beanName))
22                         return object; // 如果bean正在创建, 则直接返回
23                     beforeSingletonCreation(beanName); // 将beanName加入到缓存singletonsCurrentlyInCreation中
24                 try {
25                     // bean在初始化后会调用所有注册的BeanPostProcessor类的postProcessAfterInitialization方法
26                     object = postProcessObjectFromFactoryBean(
27                         object, beanName);
28                 } catch (Throwable ex) {
29                     throw new BeanCreationException(...);
30                 } finally {
31                     afterSingletonCreation(beanName); // 将beanName从缓存singletonsCurrentlyInCreation中移除
32                 }
33             }
34             // 如果bean已经被创建, 则向缓存中维护beanName与FactoryBean实例对象的对应关系
35             if (containsSingleton(beanName)) this.factoryBeanObjectCache.put(beanName, object);
36         }
37     }
38     return object;
39 }
```

```

33     }
34 }
35 /** 步骤2: 如果factory不是单例的, 或者bean没有被创建, 则只获得bean实例, 不需要维护到factoryBeanObjectCache中 */
36 else {
37     // 创建bean的实例对象。即: 调用FactoryBean#getObject()获得
38     Object object = doGetObjectFromFactoryBean(factory, beanName);
39
40     if (shouldPostProcess) {
41         try {
42             // bean在初始化后会调用所有注册的BeanPostProcessor类的postProcessAfterInitialization方法
43             object = postProcessObjectFromFactoryBean(object, beanName);
44         } catch (Throwable ex) {throw new BeanCreationException(
45             ...);}
46     }
47     return object;
48 }

```

- 在上面代码中, 我们还需要再谈一谈 `postProcessObjectFromFactoryBean(object, beanName)` 方法, 它的作用其实就是尽量保证所有bean在初始化之后, 都会调用所有注册了的 `BeanPostProcessor` 类的 `postProcessAfterInitialization(result, beanName)` 方法, 在实际开发过程中可以针对此特性设计自己的业务逻辑。源码如下所述:

```

AbstractAutowireCapableBeanFactory.java
1927 @Override
1928 protected Object postProcessObjectFromFactoryBean(Object object, String beanName) {
1929     return applyBeanPostProcessorsAfterInitialization(object, beanName);
1930 }

```

```

AbstractAutowireCapableBeanFactory.java
431 @Override
432 public Object applyBeanPostProcessorsAfterInitialization(Object existingBean, String beanName)
433     throws BeansException {
434
435     Object result = existingBean;
436     for (BeanPostProcessor processor : getBeanPostProcessors()) {
437         Object current = processor.postProcessAfterInitialization(result, beanName);
438         if (current == null) {
439             return result;
440         }
441         result = current;
442     }
443     return result;
444 }

```

五、getSingleton(beanName, singletonFactory)

- 在上面的文章中我们已经介绍过 `getSingleton(beanName)` 方法了，它的主要作用就是从缓存中获取单例对象。那么下面我们要介绍的方法 `getSingleton(beanName, singletonFactory)`，是针对于缓存中并不存在单例bean的时候的处理流程。源码注释如下所示：

code

```
1 public Object getSingleton(String beanName, ObjectFactory<?> singleton
  Factory) {
2     Assert.notNull(beanName, "Bean name must not be null");
3     synchronized (this.singletonObjects) {
4         /** 首先, 尝试从缓存中获取bean实例 */
5         Object singletonObject = this.singletonObjects.get(beanName);
6         if (singletonObject == null) {
7             if (this.singletonsCurrentlyInDestruction) throw new BeanC
              reationNotAllowedException(...);
8
9             // 将beanName加入到缓存inCreationCheckExclusions和缓存singlet
              onsCurrentlyInCreation中
10            beforeSingletonCreation(beanName);
11
12            boolean newSingleton = false, recordSuppressedExceptions =
              (this.suppressedExceptions == null);
13            if (recordSuppressedExceptions) this.suppressedExceptions
              = new LinkedHashSet<>();
14
15            try {
16                /** 其次, 尝试调用ObjectFactory#getObject()方法, 获取bean实
                  例 */
17                singletonObject = singletonFactory.getObject(); // Obj
                  ectFactory是接口, 所以getObject()方法需要实现
18                newSingleton = true;
19            } catch (IllegalStateException ex) {
20                singletonObject = this.singletonObjects.get(beanName);
21                if (singletonObject == null) throw ex;
22            } catch (BeanCreationException ex) {
23                if (recordSuppressedExceptions)
24                    for (Exception suppressedException : this.suppress
                      edExceptions)
25                        ex.addRelatedCause(suppressedException);
26                throw ex;
27            } finally {
28                if (recordSuppressedExceptions) this.suppressedExcepti
                  ons = null;
29                afterSingletonCreation(beanName); // 将beanName从single
                  tonsCurrentlyInCreation中移除掉
30            }
31            if (newSingleton)
32                // 添加缓存: singletonObjects 和 registeredSingletons
33                // 移除缓存: singletonFactories 和 earlySingletonObjects
34                addSingleton(beanName, singletonObject);
35        }
```

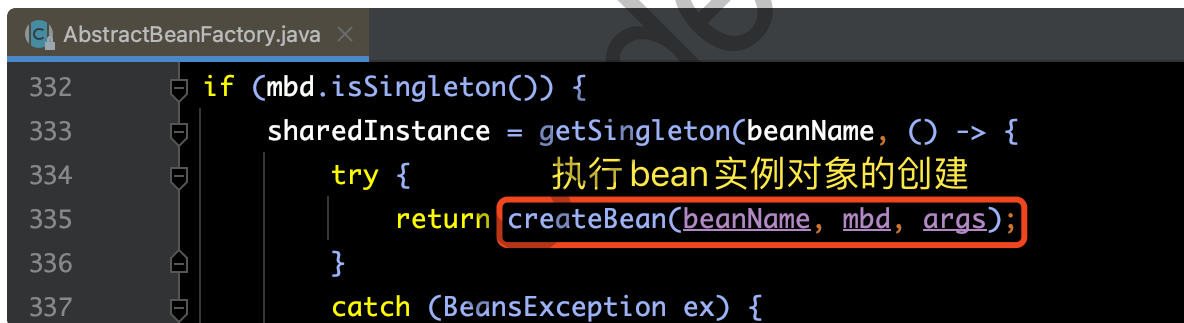
```
36         return singletonObject;
37     }
38 }
```

【解释】通过源码可以看到如下流程：

- 首先，尝试从缓存`singletonObjects`中获取bean实例，如果获取到了，就执行return返回该实例对象。
- 其次，如果没有从缓存中获取到bean实例，则通过调用`ObjectFactory#getObject()`方法，获取bean实例。由于ObjectFactory是接口，所以`getObject()`方法需要单独实现。
- 最后，将实例对象return返回即可。

六、createBean(...)

- 在上面我们介绍 `getSingleton(beanName, singletonFactory)` 方法源码的时候，提到了其中的`singletonFactory`，它是`ObjectFactory`类型的，它是一个接口，并且这个接口只提供了一个方法 `getObject()`，需要单独实现这个方法，来完成bean实例对象的创建，那么具体创建代码在哪个地方呢，即下图中红框的`createBean(beanName, mbd, args)`方法。



```
AbstractBeanFactory.java x
332     if (mbd.isSingleton()) {
333         sharedInstance = getSingleton(beanName, () -> {
334             try {
335                 return createBean(beanName, mbd, args);
336             }
337             catch (BeansException ex) {
```

- 其中`createBean(beanName, mbd, args)`方法的源码注释如下所示：


```

1  protected Object createBean(String beanName, RootBeanDefinition mbd, @
   Nullable Object[] args) {
2      RootBeanDefinition mbdToUse = mbd;
3      /** 步骤1: 根据class属性或className来获得Class实例对象, 如果mbd中没有设置beanClass, 则创建新的mbdToUse, 设置beanClass */
4      Class<?> resolvedClass = resolveBeanClass(mbd, beanName);
5      if (resolvedClass != null && !mbd.hasBeanClass() && mbd.getBeanClassName() != null) {
6          mbdToUse = new RootBeanDefinition(mbd);
7          mbdToUse.setBeanClass(resolvedClass);
8      }
9
10     /** 步骤2: 验证和准备覆盖的方法 (MethodOverrides) */
11     try {
12         mbdToUse.prepareMethodOverrides();
13     } catch (BeanDefinitionValidationException ex) {throw new BeanDefinitionStoreException(...);}
14
15     /** 步骤3: 给BeanPostProcessors一个机会, 来返回一个替代真正实例的代理对象, 并直接return返回*/
16     try {
17         Object bean = resolveBeforeInstantiation(beanName, mbdToUse);
18         if (bean != null) return bean; // 如果存在代理对象, 则返回代理对象
19     } catch (Throwable ex) {throw new BeanCreationException(...);}
20
21     /** 步骤4: 真正开始创建bean实例对象 */
22     try {
23         Object beanInstance = doCreateBean(beanName, mbdToUse, args);
24         return beanInstance;
25     } catch (Throwable ex) {throw new BeanCreationException(...);}
26 }

```

- 针对“步骤1”的[resolveBeanClass\(mbd, beanName\)](#)方法的源码和注释如下所示:

```

1  protected Class<?> resolveBeanClass(RootBeanDefinition mbd, String beanName, Class<?>... typesToMatch) {
2      try { ... } catch (PrivilegedActionException pae) { ... }
13 }

```

```
1 private Class<?> doResolveBeanClass(RootBeanDefinition mbd, Class<?>
  >... typesToMatch) {
2     ClassLoader beanClassLoader = getBeanClassLoader(), dynamicLoader
    = beanClassLoader;
3     if (!ObjectUtils.isEmpty(typesToMatch)) { // eg: typesToMatch为null, 不执行这段代码
4         ClassLoader tempClassLoader = getTempClassLoader();
5         if (tempClassLoader != null) {
6             dynamicLoader = tempClassLoader;
7             freshResolve = true;
8             if (tempClassLoader instanceof DecoratingClassLoader) {
9                 DecoratingClassLoader dcl = (DecoratingClassLoader) tempClassLoader;
10                for (Class<?> typeToMatch : typesToMatch)
11                    dcl.excludeClass(typeToMatch.getName());
12            }
13        }
14    }
15
16    boolean freshResolve = false;
17    String className = mbd.getBeanClassName();
18    /** 1: 如果可以获得className, 并且evaluated与className不同, 则以evaluated为准 */
19    if (className != null) {
20        // 针对mbd (可能将其作为表达式进行解析), 解析出className或者Class实例
21        Object evaluated = evaluateBeanDefinitionString(className, mbd);
22        if (!className.equals(evaluated)) {
23            if (evaluated instanceof Class) return (Class<?>) evaluated;
24            // 返回通过mbd解析出来的evaluated实例
25            else if (evaluated instanceof String) {
26                className = (String) evaluated;
27                freshResolve = true;
28            } else throw new IllegalStateException(...);
29        }
30        if (freshResolve) {
31            if (dynamicLoader != null) {
32                try {
33                    return dynamicLoader.loadClass(className); // 返回通过mbd解析出来的evaluated的Class实例
34                } catch (ClassNotFoundException ex) {...}
35            }
36            return ClassUtils.forName(className, dynamicLoader);
37        }
38    }
```

```

38
39     /** 2: 以mbd.getBeanClassName()为准, 来创建Class实例对象 */
40     return mbd.resolveBeanClass(beanClassLoader);
41 }

```

【解释】主要执行如下几个步骤：

- 步骤1：尝试从mbd中**获得beanClass**—— `mbd.getBeanClass()`。
- 步骤2：如果无法获得 `beanClass`，那么再尝试根据mbd的配置文件内容，**解析出beanClass**。
 - 2-1：从mbd中**获得beanClassName**—— `mbd.getBeanClassName()`。
 - 2-2：再针对mbd（可能将其作为表达式进行解析），解析出**evaluated**（有可能是className或者Class实例）
 - 2-3：如果 `beanClassName` 与 `evaluated` 不同，则**以evaluated为准**。
 - 2-4：否则，通过 `beanClassName` 获得它所对应的Class实例对象。
- 针对“步骤2”的**`mbdToUse.prepareMethodOverrides()`**方法是用于**检查查找方法是否存在并确定其重载状态**，其源码和注释如下所示：

▼ AbstractBeanDefinition.java Java | [复制代码](#)

```

1  public void prepareMethodOverrides() throws BeanDefinitionValidationEx
    ception {
2      if (hasMethodOverrides()) // 如果配置中存在lookup-method和replace-met
        hod, 那么hasMethodOverrides()返回true
3          getMethodOverrides().getOverrides().forEach(this::prepareMetho
        dOverride);
4  }
5
6  protected void prepareMethodOverride(MethodOverride mo) throws BeanDef
    initionValidationException {
7      // 根据在lookup-method和replace-method上配置的方法名, 去bean类中查找相同
        方法名称的方法数量
8      int count = ClassUtils.getMethodCountForName(getBeanClass(), mo.ge
        tMethodName());
9      if (count == 0) throw new BeanDefinitionValidationException(...);
        // 如果存在0个, 则抛出异常
10     else if (count == 1) mo.setOverloaded(false); // 如果存在1个, 则标记
        为未重载, 以避免arg类型检查的开销
11 }

```

- 在Spring中，虽然没有 `override-method` 这样的配置，但是针对配置的**lookup-method**和**replace-method**会被的存放在BeanDefinition中的 `methodOverrides` 属性里。
- 然后，会通过MethodOverride中的方法名，来校验bean类中是否存在对应的方法。并且，**如果只**

匹配到了1个方法，那么将重写标记为未重载，以避免arg类型检查的开销。因为对于方法的匹配来说，如果在一个类中存在多个重载方法，那么在函数调用及增强的时候，还需要根据参数类型进行匹配，这样才能最终确认当前调用的到底是哪个方法。但是，Spring将一部分匹配工作在这里完成了，即：如果当前类中匹配的方法只有1个，那么就设置重载该方法为false，这样在后续调用的时候就可以直接使用这个方法，而不需要进行方法的参数匹配操作了。

- 针对“步骤3”的[resolveBeforeInstantiation\(beanName, mbdToUse\)](#)方法的源码和注释如下所示：

```
▼ AbstractAutowireCapableBeanFactory.java      Java | 复制代码

1  protected Object resolveBeforeInstantiation(String beanName, RootBeanDefinition mbd) {
2      Object bean = null;
3      // 默认beforeInstantiationResolved为null，所以会进入if语句中
4      if (!Boolean.FALSE.equals(mbd.beforeInstantiationResolved)) {
5          // 如果不是自定义的mbd，并且配置了一些InstantiationAwareBeanPostProcessor
6          if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
7              Class<?> targetType = determineTargetType(beanName, mbd);
              // 获得beanClass
8              if (targetType != null) {
9                  // 调用InstantiationAwareBeanPostProcessor的postProcessBeforeInstantiation方法
10                 bean = applyBeanPostProcessorsBeforeInstantiation(targetType, beanName);
11                 if (bean != null) {
12                     // 调用BeanPostProcessor的postProcessAfterInitialization方法
13                     bean = applyBeanPostProcessorsAfterInitialization(bean, beanName);
14                 }
15             }
16         }
17         mbd.beforeInstantiationResolved = (bean != null); // 设置是否执行了beforeInstantiation的解析操作
18     }
19     return bean;
20 }
```

- [applyBeanPostProcessorsBeforeInstantiation\(\)](#)方法如下所示：

```

1  protected Object applyBeanPostProcessorsBeforeInstantiation(Class<?> beanClass, String beanName) {
2      for (InstantiationAwareBeanPostProcessor bp : getBeanPostProcessorCache().instantiationAware) {
3          Object result = bp.postProcessBeforeInstantiation(beanClass, beanName);
4          if (result != null) return result; // 只要返回的result值【不为空】，则中断循环调用，返回结果
5      }
6      return null;
7  }

```

【解释】实例化前的后处理器应用——即：创建bean的代理对象。会在bean的实例化操作之前进行调用，也就是将AbstractBeanDefinition转换为BeanWrapper前的处理。给子类一个修改BeanDefinition的机会，也就是说当程序经过这个方法后，**bean可能已经不是我们认为的那个bean了**，而是或许成为了一个经过处理的代理bean，或者可能是通过cglib生成的bean，也可能是通过某些其他技术生成的bean。

- `applyBeanPostProcessorsAfterInstantiation()`方法如下所示：

```

1  public Object applyBeanPostProcessorsAfterInitialization(Object existingBean, String beanName) {
2      Object result = existingBean;
3      for (BeanPostProcessor processor : getBeanPostProcessors()) {
4          Object current = processor.postProcessAfterInitialization(result, beanName);
5          if (current == null) return result; // 只要出现了返回的result值为【空】，则中断循环调用，返回结果
6          result = current;
7      }
8      return result;
9  }

```

【解释】实例化后的后处理器应用——即：对bean进行后置处理。Spring会在bean的初始化后尽可能保证将注册的后处理器的 `postProcessAfterInitialization` 方法应用到这个bean中，**因为如果返回的bean不为空，那么便不会再次经历普通bean的创建过程，所以只能在这里应用后处理器的 `postProcessAfterInitialization` 方法。**

- 针对“步骤4”的`doCreateBean(beanName, mbdToUse, args)`方法的源码解析，我们会再下面其他章节中进行详细解析和说明，此处暂略。

七、循环依赖

- 对于循环依赖，就是A类中引用了B类，B类中引用了C类，而C类中引用了A类，那么这样就会出现循环依赖的情况。针对循环依赖，有如下情况：
- 单例类型——构造器循环依赖，则无法被解决。

```
Java | 复制代码
1 <bean id="testA" class="com.muse.TestA">
2   <constructor-arg index="0" ref="testB"/>
3 </bean>
4 <bean id="testB" class="com.muse.TestB">
5   <constructor-arg index="0" ref="testC"/>
6 </bean>
7 <bean id="testC" class="com.muse.TestC">
8   <constructor-arg index="0" ref="testA"/>
9 </bean>
```

- 单例类型——setter循环依赖，可以通过提前暴露刚完成构造器注入但未完成其他步骤的bean来解决。

```
Java | 复制代码
1 <bean id="testA" class="com.muse.TestA">
2   <property name="testB" ref="testB"/>
3 </bean>
4 <bean id="testB" class="com.muse.TestB">
5   <property name="testC" ref="testC"/>
6 </bean>
7 <bean id="testC" class="com.muse.TestC">
8   <property name="testA" ref="testA"/>
9 </bean>
```

- 原型类型——无法被解决。

```
1 <bean id="testA" class="com.muse.TestA" scope="prototype">
2   <property name="testB" ref="testB"/>
3 </bean>
4 <bean id="testB" class="com.muse.TestB" scope="prototype">
5   <property name="testC" ref="testC"/>
6 </bean>
7 <bean id="testC" class="com.muse.TestC" scope="prototype">
8   <property name="testA" ref="testA"/>
9 </bean>
```

八、doCreateBean(...)

8.1> 概述

- 我们跟踪了这么多Spring代码，经历了这么多函数，或多或少也会发现这么一个规律，就是：一个真正干活的函数，大多是以do开头命名的。那么，我们马上要介绍的这个 `doCreateBean(...)` 方法，就是负责常规bean创建的。相关的源码和注释如下所示：

```
1  protected Object doCreateBean(String beanName, RootBeanDefinition mbd,
2      @Nullable Object[] args) {
3      /** 步骤1: 获得BeanWrapper实例对象instanceWrapper */
4      BeanWrapper instanceWrapper = null;
5      if (mbd.isSingleton()) instanceWrapper = this.factoryBeanInstanceC
6      ache.remove(beanName); // 清除缓存
7      if (instanceWrapper == null) instanceWrapper = createBeanInstance(
8      beanName, mbd, args); // 创建BeanWrapper实例
9
10     /** 步骤2: 调用所有配置了MergedBeanDefinitionPostProcessor实现类的postP
11     rocessMergedBeanDefinition方法 */
12     Object bean = instanceWrapper.getWrappedInstance();
13     Class<?> beanType = instanceWrapper.getWrappedClass();
14     if (beanType != NullBean.class) mbd.resolvedTargetType = beanType;
15     synchronized (mbd.postProcessingLock) {
16         if (!mbd.postProcessed) {
17             try {
18                 // MergedBeanDefinitionPostProcessor#postProcessMerged
19                 BeanDefinition(mbd, beanType, beanName)
20                 applyMergedBeanDefinitionPostProcessors(mbd, beanType,
21                 beanName);
22             } catch (Throwable ex) {...}
23             mbd.postProcessed = true;
24         }
25     }
26
27     /** 步骤3: 针对“正在创建”的“允许循环依赖”的“单例”执行【提前曝光】 */
28     boolean earlySingletonExposure = (mbd.isSingleton() && // 是否是单例
29     的
30     this.allowCircularReferences &&
31     // 是否允许循环依赖
32     isSingletonCurrentlyInCreation(b
33     eanName)); // 单例bean是否正在创作中
34     if (earlySingletonExposure)
35         addSingletonFactory(beanName, () -> getEarlyBeanReference(bean
36     Name, mbd, bean));
37
38     Object exposedObject = bean;
39     try {
40         /** 步骤4: 对bean进行填充操作, 将各个属性值进行注入 */
41         populateBean(beanName, mbd, instanceWrapper);
42
43         /** 步骤5: 调用初始化方法, 例如: init-method */
44         exposedObject = initializeBean(beanName, exposedObject, mbd);
45     } catch (Throwable ex) {...}
```



```

36
37
38     /** 步骤6: 针对需要执行“提前曝光”的单例 */
39     if (earlySingletonExposure) {
40         Object earlySingletonReference = getSingleton(beanName, false);
41
42         if (earlySingletonReference != null) {
43             if (exposedObject == bean) exposedObject = earlySingletonReference; // bean没有被增强改变
44             else if (!this.allowRawInjectionDespiteWrapping && hasDependentBean(beanName)) {
45                 String[] dependentBeans = getDependentBeans(beanName);
46                 // 获得所有依赖
47                 Set<String> actualDependentBeans = new LinkedHashSet<>(dependentBeans.length);
48                 for (String dependentBean : dependentBeans)
49                     if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) // 执行依赖检测
50                         actualDependentBeans.add(dependentBean);
51                 // 因为bean创建后，它所依赖的bean一定创建了，那么不为空则表示所依赖的bean没有全部创建完，即：存在循环依赖
52                 if (!actualDependentBeans.isEmpty()) throw new BeanCurrentlyInCreationException(...);
53             }
54         }
55     }
56
57     try {
58         /** 步骤7: 如果配置了destroy-method，这里需要注册以便于在销毁时候进行调用 */
59         registerDisposableBeanIfNecessary(beanName, bean, mbd);
60     } catch (BeanDefinitionValidationException ex) {...}
61
62     return exposedObject;
63 }

```

- 下面我们就针对流程中的重要逻辑进行更深入的源码解析。

8.2> createBeanInstance()创建bean的实例

- 首先，我们先来分析一下用于创建bean的实例的 `createBeanInstance(beanName, mbd, args)` 方法。相关的源码和注释如下所示：

```
1  protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition mbd, @Nullable Object[] args) {
2      /** 步骤1: 解析beanClass */
3      Class<?> beanClass = resolveBeanClass(mbd, beanName);
4      if (beanClass != null && !Modifier.isPublic(beanClass.getModifiers()) && !mbd.isNonPublicAccessAllowed())
5          throw new BeanCreationException(...);
6
7      /** 步骤2: 如果配置了instanceSupplier, 则通过调用Supplier#get()方法来创建bean的实例, 并封装为BeanWrapper实例 */
8      Supplier<?> instanceSupplier = mbd.getInstanceSupplier();
9      if (instanceSupplier != null) return obtainFromSupplier(instanceSupplier, beanName);
10
11     /** 步骤3: 如果配置了factoryMethodName或者配置文件中存在factory-method, 则使用工厂方法创建bean的实例 */
12     if (mbd.getFactoryMethodName() != null) return instantiateUsingFactoryMethod(beanName, mbd, args);
13
14     boolean resolved = false, autowireNecessary = false;
15     if (args == null) {
16         synchronized (mbd.constructorArgumentLock) {
17             // 一个类有多个构造函数, 每个构造函数有不同的参数, 所以调用前需要先根据参数锁定构造函数或者对应的工厂方法
18             if (mbd.resolvedConstructorOrFactoryMethod != null) {
19                 resolved = true;
20                 autowireNecessary = mbd.constructorArgumentsResolved;
21             }
22         }
23     }
24
25     /** 步骤4: 如果已经解析过 (resolved=true), 那么就使用解析好的构造函数方法, 不需要再次锁定 */
26     if (resolved) {
27         if (autowireNecessary)
28             return autowireConstructor(beanName, mbd, null, null);
29         // 构造函数自动注入
30         else
31             return instantiateBean(beanName, mbd); // 使用默认构造函数构造
32
33     /** 步骤5: 如果没解析过, 那么则需要根据参数解析构造函数 */
34     Constructor<?>[] ctors = determineConstructorsFromBeanPostProcessors(beanClass, beanName);
```

```

35     if (ctors != null || mbd.getResolvedAutowireMode() == AUTOWIRE_CON
36     STRUCTOR ||
37         mbd.hasConstructorArgumentValues() || !ObjectUtils.isEmpty
38         (args)) {
39         return autowireConstructor(beanName, mbd, ctors, args); // 构造
40         函数自动注入
41     }
42     /** 步骤6: 尝试获取默认构造的首选构造函数 */
43     ctors = mbd.getPreferredConstructors();
44     if (ctors != null) return autowireConstructor(beanName, mbd, ctors
45     , null); // 构造函数自动注入
46     /** 步骤7: 如果以上都不行, 则使用默认构造函数构造bean实例 */
47     return instantiateBean(beanName, mbd);
48 }

```

- 在上面的代码中, 主要负责创建bean的相关方法有两个, 分别是 `autowireConstructor(...)` 和 `instantiateBean(...)`, 下面我们就针对这两个方法进行源码解析。

8.2.1> autowireConstructor(...)有参数的实例化构造

- 首先是 `autowireConstructor(...)` 方法, 它是负责有参数的实例化构造, 这部分流程比较复杂, 下面是该方法的源码和注释:

```

▼ AbstractAutowireCapableBeanFactory.java      Java | 复制代码
1  BeanWrapper autowireConstructor(String beanName, RootBeanDefinition mbd
2  , Constructor<?>[] ctors,
3  Object[] explicitArgs){
4  return new ConstructorResolver(this).autowireConstructor(beanName,
5  mbd, ctors, explicitArgs);
6  }

```

```
1 public BeanWrapper autowireConstructor(String beanName, RootBeanDefinition mbd, Constructor<?>[] chosenCtors,
2                                     Object[] explicitArgs) {
3     BeanWrapperImpl bw = new BeanWrapperImpl();
4     this.beanFactory.initBeanWrapper(bw);
5     Constructor<?> constructorToUse = null;
6     ArgumentsHolder argsHolderToUse = null;
7
8     /** 步骤1: 尝试获得构造函数 (constructorToUse) 和方法入参 (argsToUse) */
9     Object[] argsToUse = null;
10    if (explicitArgs != null) // case1: 如果getBean方法调用的时候指定了方法参数, 则直接使用
11        argsToUse = explicitArgs;
12    else { // case2: 如果没有指定explicitArgs, 则尝试从mbd中获取构造函数入参argsToUse和构造函数constructorToUse
13        Object[] argsToResolve = null;
14        synchronized (mbd.constructorArgumentLock) {
15            constructorToUse = (Constructor<?>) mbd.resolvedConstructorOrFactoryMethod;
16            if (constructorToUse != null && mbd.constructorArgumentsResolved) {
17                argsToUse = mbd.resolvedConstructorArguments;
18                if (argsToUse == null) argsToResolve = mbd.preparedConstructorArguments;
19            }
20        }
21        if (argsToResolve != null)
22            // 转换参数类型。假设构造函数为A(int, int), 可以通过如下方法将入参的("1", "1")转换为(1, 1)
23            argsToUse = resolvePreparedArguments(beanName, mbd, bw, constructorToUse, argsToResolve);
24    }
25
26    /** 步骤2: 如果constructorToUse和argsToUse没有全部解析出来, 则尝试从配置文件中解析获取 */
27    if (constructorToUse == null || argsToUse == null) {
28        Constructor<?>[] candidates = chosenCtors;
29        if (candidates == null) { // 如果入参chosenCtors为空, 则获取bean中所有的构造方法作为“候选”构造方法
30            Class<?> beanClass = mbd.getBeanClass();
31            try {
32                candidates = (mbd.isNonPublicAccessAllowed() ?
33                    beanClass.getDeclaredConstructors() :
34                    beanClass.getConstructors());
35            } catch (Throwable ex) {...}
```

```

36     }
37
38     // 如果类中只有1个无参的构造函数，则创建bean的实例对象并且return
39     if (candidates.length == 1 && explicitArgs == null && !mbd.hasConstructorArgumentValues()) {
40         Constructor<?> uniqueCandidate = candidates[0];
41         if (uniqueCandidate.getParameterCount() == 0) {
42             synchronized (mbd.constructorArgumentLock) {
43                 mbd.resolvedConstructorOrFactoryMethod = uniqueCandidate;
44                 mbd.constructorArgumentsResolved = true;
45                 mbd.resolvedConstructorArguments = EMPTY_ARGS;
46             }
47             bw.setBeanInstance(instantiate(beanName, mbd, uniqueCandidate, EMPTY_ARGS));
48             return bw;
49         }
50     }
51
52     boolean autowiring = (chosenCtors != null ||
53         mbd.getResolvedAutowireMode() == AutowireCapableBeanFactory.AUTOWIRE_CONSTRUCTOR);
54     ConstructorArgumentValues resolvedValues = null;
55     // 解析构造函数参数个数minNrOfArgs
56     int minNrOfArgs;
57     if (explicitArgs != null) minNrOfArgs = explicitArgs.length;
58     else {
59         ConstructorArgumentValues cargs = mbd.getConstructorArgumentValues(); // 提取配置文件中配置的构造函数参数
60         resolvedValues = new ConstructorArgumentValues(); // 用于承载解析后的构造函数参数的值
61         minNrOfArgs = resolveConstructorArguments(beanName, mbd, bw, cargs, resolvedValues); // 解析参数个数
62     }
63
64     // 对构造函数执行排序操作，其中：public构造函数优先且参数数量降序排列，然后是非public构造函数参数数量降序排列
65     AutowireUtils.sortConstructors(candidates);
66
67     // 遍历所有构造函数，对每个构造函数进行参数匹配操作
68     int minTypeDiffWeight = Integer.MAX_VALUE;
69     Set<Constructor<?>> ambiguousConstructors = null;
70     Deque<UnsatisfiedDependencyException> causes = null;
71     for (Constructor<?> candidate : candidates) {
72         int parameterCount = candidate.getParameterCount();
73         if (constructorToUse != null && argsToUse != null && argsToUse.length > parameterCount) break;
74         if (parameterCount < minNrOfArgs) continue;

```

```

75
76         /** 创建构造函数的"参数持有者 (ArgumentsHolder) "实例对象argsHo
77         lder */
78         ArgumentsHolder argsHolder;
79         Class<?>[] paramTypes = candidate.getParameterTypes();
80         // 获得构造函数的参数类型集合
81         if (resolvedValues != null) { // 只有当explicitArgs等于null
82             时, resolvedValues才满足不为空
83             try {
84                 // 获得@ConstructorProperties({"x", "y"})注解里配置
85                 的参数名称
86                 String[] paramNames = ConstructorPropertiesChecke
87                 r.evaluate(candidate, parameterCount);
88                 if (paramNames == null) {
89                     // 从BeanFactory中获得配置的ParameterNameDiscov
90                     erer实现类
91                     ParameterNameDiscoverer pnd = this.beanFactor
92                     y.getParameterNameDiscoverer();
93                     if (pnd != null)
94                         paramNames = pnd.getParameterNames(candid
95                         ate); // 获得构造函数的参数名称
96                 }
97                 // 根据【paramTypes】和【paramNames】创建参数持有者Arg
98                 umentsHolder
99                 argsHolder = createArgumentArray(beanName, mbd, r
100                 esolvedValues, bw, paramTypes, paramNames,
101                 getUserDeclaredConstructor(candidate), au
102                 towiring, candidates.length == 1);
103             } catch (UnsatisfiedDependencyException ex) {...}
104         } else {
105             if (parameterCount != explicitArgs.length) continue;
106             argsHolder = new ArgumentsHolder(explicitArgs);
107         }
108
109         /** 探测是否有不确定性的构造函数存在, 例如: 不同构造函数的参数为父子
110         关系 */
111         int typeDiffWeight = (mbd.isLenientConstructorResolution(
112         ) ?
113             argsHolder.getTypeDifferenceWeight(paramTypes) : args
114             Holder.getAssignabilityWeight(paramTypes));
115         if (typeDiffWeight < minTypeDiffWeight) { // 如果它代表着当
116             前最接近的匹配, 则选择作为构造函数
117             constructorToUse = candidate;
118             argsHolderToUse = argsHolder;
119             argsToUse = argsHolder.arguments;
120             minTypeDiffWeight = typeDiffWeight;
121             ambiguousConstructors = null;
122         }

```

```

108         } else if (constructorToUse != null && typeDiffWeight ==
109 minTypeDiffWeight) {
110             if (ambiguousConstructors == null) {
111                 ambiguousConstructors = new LinkedHashSet<>();
112                 ambiguousConstructors.add(constructorToUse);
113             }
114             ambiguousConstructors.add(candidate);
115         }
116     }
117     if (constructorToUse == null) {
118         if (causes != null) {
119             UnsatisfiedDependencyException ex = causes.removeLast
120 ();
121             for (Exception cause : causes)
122                 this.beanFactory.onSuppressedException(cause);
123             throw ex;
124         }
125         throw new BeanCreationException(...);
126     } else if (ambiguousConstructors != null && !mbd.isLenientCon
127 structorResolution())
128         throw new BeanCreationException(...);
129
130     // 将解析的构造函数加入到缓存中
131     if (explicitArgs == null && argsHolderToUse != null)
132         argsHolderToUse.storeCache(mbd, constructorToUse);
133
134     Assert.state(argsToUse != null, "Unresolved constructor argument
135 s");
136
137     /** 步骤3: 通过constructorToUse和argsToUse创建bean的实例对象, 并存储到B
138 eanWrapper中 */
139     bw.setBeanInstance(instantiate(beanName, mbd, constructorToUse, a
140 rgsToUse));
141     return bw;
142 }

```

• 针对上面的源码内容，我们可以总结出如下几个步骤：

- 1、确定构造函数的参数**argsToUse**：
 - 根据 `explicitArgs` 参数进行判断
 - 尝试从 `mbd` 中获取
 - 尝试从 `配置文件` 中获取
- 2、确定构造函数**constructorToUse**。
- 3、根据确定的构造函数转换对应的参数类型。

- 4、构造函数不确定性的验证。
- 5、根据实例化策略类中的 `instantiate(mbd, beanName, this)` 方法以及 `constructor` `ToOneUse` 和 `argsToOneUse` 来实例化bean，并封装到`BeanWrapper`中。

8.2.2> instantiateBean(...)无参数的实例化构造

- 上面我们介绍了带参数的构造方法解析，那么下面我们就针对不带参数的构造函数的实例化过程进行解析操作，其相关注释和源码如下所示：

```
AbstractAutowireCapableBeanFactory.java  Java | 复制代码

1  protected BeanWrapper instantiateBean(String beanName, RootBeanDefinition mbd) {
2      try {
3          Object beanInstance;
4          if (System.getSecurityManager() != null)
5              beanInstance = AccessController.doPrivileged((PrivilegedAction<Object>) () -> getInstantiationStrategy().instantiate(mbd, beanName, this), getAccessControlContext());
6          else
7              beanInstance = getInstantiationStrategy().instantiate(mbd, beanName, this); //实例化策略
8          BeanWrapper bw = new BeanWrapperImpl(beanInstance);
9          initBeanWrapper(bw); // 将创建好的实例封装为BeanWrapper对象
10         return bw;
11     }
12     catch (Throwable ex) {throw new BeanCreationException(...);}
13 }
```

- 通过上面针对instantiateBean方法源码之后，我们会发现，主要只有两个操作：
- 1> 通过实例化策略类的 `instantiate(mbd, beanName, this)` 方法创建bean实例对象。
- 2> 将创建的bean实例封装为 `BeanWrapper` 对象。

8.2.3> instantiate(...)

- 在上面我们提到的“通过实例化策略类的`instantiate(mbd, beanName, this)`方法创建bean实例对象”，那么下面我们就来分析一下这个方法的内部逻辑：


```
1 public Object instantiate(RootBeanDefinition bd, @Nullable String bean
  Name, BeanFactory owner) {
2     /** 步骤1: 如果没有配置lookup-method或replace-method, 则直接使用反射创建b
    ean的实例对象即可 */
3     if (!bd.hasMethodOverrides()) {
4         Constructor<?> constructorToUse;
5         synchronized (bd.constructorArgumentLock) {
6             constructorToUse = (Constructor<?>) bd.resolvedConstructor
  OrFactoryMethod;
7             // 获得构造函数实例
8             if (constructorToUse == null) {
9                 final Class<?> clazz = bd.getBeanClass();
10                if (clazz.isInterface()) throw new BeanInstantiationEx
  ception(...);
11                try {
12                    if (System.getSecurityManager() != null)
13                        constructorToUse = AccessController.doPrivileg
  ed((PrivilegedExceptionAction<Constructor<?>>) clazz::getDeclaredConst
  ructor);
14                    else
15                        constructorToUse = clazz.getDeclaredConstructo
  r();
16                    bd.resolvedConstructorOrFactoryMethod = constructo
  rToUse;
17                }
18                catch (Throwable ex) {throw new BeanInstantiationExcep
  tion(...);}
19            }
20        }
21        return BeanUtils.instantiateClass(constructorToUse); // 通过反
    射创建bean实例
22    }
23    /** 步骤2: 否则, 需要使用cglib创建代理对象, 将动态方法织入到bean的实例对象中
    */
24    else
25        return instantiateWithMethodInjection(bd, beanName, owner);
    // Must generate CGLIB subclass.
26 }
```

- 通过instantiateWithMethodInjection(bd, beanName, owner)方法, 使用 cglib 创建代理对象

```
1  protected Object instantiateWithMethodInjection(RootBeanDefinition bd,
2      String beanName, BeanFactory owner) {
3      return instantiateWithMethodInjection(bd, beanName, owner, null);
4  }
5  protected Object instantiateWithMethodInjection(RootBeanDefinition bd,
6      String beanName, BeanFactory owner,
7      Constructor<?> ctor, 0
8      Object... args) {
9      return new CglibSubclassCreator(bd, owner).instantiate(ctor, args)
10     ;
11 }
12
13 public Object instantiate(@Nullable Constructor<?> ctor, Object... arg
14 s) {
15     /** 步骤1: 创建Cglib代理类 */
16     Class<?> subclass = createEnhancedSubclass(this.beanDefinition);
17
18     /** 步骤2: 创建Cglib代理类实例对象 */
19     Object instance;
20     if (ctor == null) instance = BeanUtils.instantiateClass(subclass);
21     else {
22         try {
23             Constructor<?> enhancedSubclassConstructor = subclass.getCo
24 nstructor(ctor.getParameterTypes());
25             instance = enhancedSubclassConstructor.newInstance(args);
26         }
27         catch (Exception ex) {...}
28     }
29
30     /** 步骤3: 将代理对象封装成Factory实例对象，并注入lookup-method和replace-
31 mehtod */
32     Factory factory = (Factory) instance;
33     factory.setCallbacks(new Callback[] {NoOp.INSTANCE,
34         new LookupOverrideMethodInterceptor(this.beanDefinition, t
35 his.owner), // 注入lookup-method
36         new ReplaceOverrideMethodInterceptor(this.beanDefinition,
37 this.owner)}); // 注入replace-mehtod
38     return instance;
39 }
40
41 private Class<?> createEnhancedSubclass(RootBeanDefinition beanDefinit
42 ion) {
43     Enhancer enhancer = new Enhancer();
44     enhancer.setSuperclass(beanDefinition.getBeanClass());
```

```

36     enhancer.setNamingPolicy(SpringNamingPolicy.INSTANCE);
37     if (this.owner instanceof ConfigurableBeanFactory) {
38         ClassLoader cl = ((ConfigurableBeanFactory) this.owner).getBeanClassLoader();
39         enhancer.setStrategy(new ClassLoaderAwareGeneratorStrategy(cl)
40     );
41     }
42     enhancer.setCallbackFilter(new MethodOverrideCallbackFilter(beanDefinition));
43     enhancer.setCallbackTypes(CALLBACK_TYPES);
44     return enhancer.createClass(); // 创建Cglib代理类
    }

```

8.3> getEarlyBeanReference(...)记录创建bean的ObjectFactory

- 好了，经过上面8.2章节的一大段解析之后，我们还是要把视角放到 `doCreateBean(...)`，在这个方法里，有如下一段代码，是用来处理单例提前曝光逻辑的：

```

▼ AbstractAutowireCapableBeanFactory.java      Java | 复制代码

1  /** 步骤3：判断是否【提前曝光】单例 */
2  boolean earlySingletonExposure = (mbd.isSingleton() && // 单例
3      this.allowCircularReferences && // 允许循环依赖
4      isSingletonCurrentlyInCreation(beanName)); // 正在创建的单例
5  if (earlySingletonExposure)
6      addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));

```

- 对于 `addSingletonFactory(...)` 方法，主要是为了避免后期循环依赖，可以在bean初始化完成前将用于创建bean实例的 `ObjectFactory` 加入缓存中

```

1  protected void addSingletonFactory(String beanName, ObjectFactory<?> singletonFactory) {
2      Assert.notNull(singletonFactory, "Singleton factory must not be null");
3      synchronized (this.singletonObjects) {
4          if (!this.singletonObjects.containsKey(beanName)) {
5              this.singletonFactories.put(beanName, singletonFactory);
6              this.earlySingletonObjects.remove(beanName);
7              this.registeredSingletons.add(beanName);
8          }
9      }
10 }

```

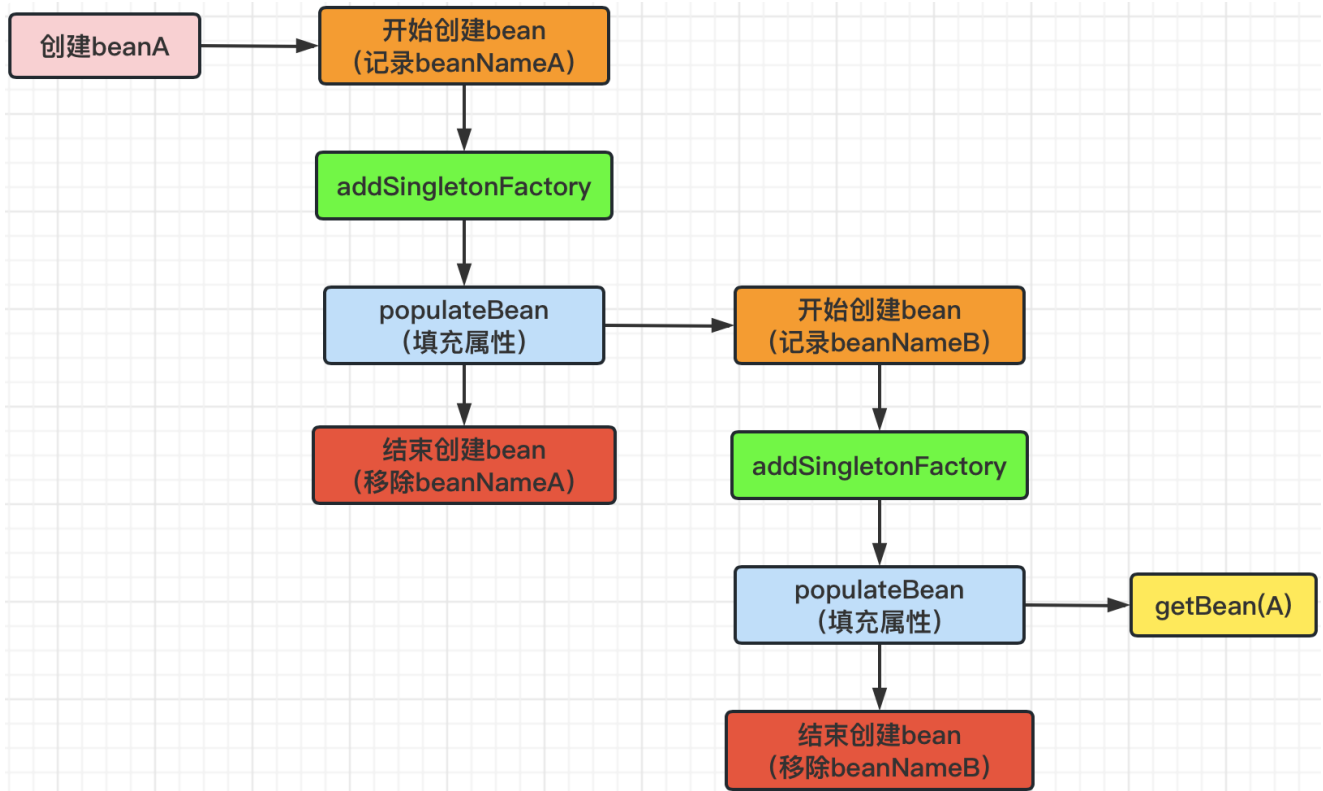
- 对于 `getEarlyBeanReference(beanName, mbd, bean)` 方法，它会调用所有 `SmartInstantiationAwareBeanPostProcessor#getEarlyBeanReference(...)` 方法。

```

AbstractAutowireCapableBeanFactory.java
1  protected Object getEarlyBeanReference(String beanName, RootBeanDefinition mbd, Object bean) {
2      Object exposedObject = bean;
3      if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
4          for (SmartInstantiationAwareBeanPostProcessor bp : getBeanPostProcessorCache().smartInstantiationAware) {
5              exposedObject = bp.getEarlyBeanReference(exposedObject, beanName);
6          }
7      }
8      return exposedObject;
9  }

```

- 我们以 **AB 循环依赖** 为例，类 **A** 中含有属性类 **B**，而类 **B** 中又会含有属性类 **A**，那么初始化 beanA 的过程如下图所示：



- 当调用 `getBean(A)` 的时候，并不是直接去实例化A，而是先去检测缓存中是否有已经创建好的bean，或者是否已经存在创建好的ObjectFactory，而此时对于A的ObjectFactory我们早已经创建，所以便不会再去向后执行，而是直接调用 `ObjectFactory#getObject()` 方法去创建A。

8.4> populateBean(...)属性注入

- 针对属性注入的操作，是由`populateBean(...)`方法进行负责的，其相关源码和注释如下图所示：

```
1  protected void populateBean(String beanName, RootBeanDefinition mbd, @
   Nullable BeanWrapper bw) {
2      /** 步骤1: 如果bean的实例bw为null, 但是却定义了bean的属性值, 则抛异常; 否则
       直接return返回 */
3      if (bw == null) {
4          if (mbd.hasPropertyValues()) throw new BeanCreationException(
           ...);
5          else return;
6      }
7
8      /** 步骤2: 针对配置了InstantiationAwareBeanPostProcessor实现类, 那么会调
       用postProcessAfterInstantiation方法 */
9      if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors(
           ))
10         for (InstantiationAwareBeanPostProcessor bp : getBeanPostProce
               ssorCache().instantiationAware)
11             if (!bp.postProcessAfterInstantiation(bw.getWrappedInstanc
                   e(), beanName))
12                 return;
13
14     /** 步骤3: 获得配置的bean属性, 然后根据注入类型 (byName/byType) 执行注入操
       作 */
15     PropertyValues pvs = (mbd.hasPropertyValues() ? mbd.getPropertyVal
           ues() : null);
16     int resolvedAutowireMode = mbd.getResolvedAutowireMode(); // 获得自
       动装配模型AutowireMode
17     if (resolvedAutowireMode == AUTOWIRE_BY_NAME || resolvedAutowireMo
           de == AUTOWIRE_BY_TYPE) {
18         MutablePropertyValues newPvs = new MutablePropertyValues(pvs);
19         if (resolvedAutowireMode == AUTOWIRE_BY_NAME)
20             autowireByName(beanName, mbd, bw, newPvs); // 通过set方法方
               法, 根据name自动注入
21         if (resolvedAutowireMode == AUTOWIRE_BY_TYPE)
22             autowireByType(beanName, mbd, bw, newPvs); // 通过set方法方
               法, 根据type自动注入
23         pvs = newPvs;
24     }
25
26     /** 步骤4: 获取加工处理后的属性pvs */
27     boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors
           ();
28     boolean needsDepCheck = (mbd.getDependencyCheck() != AbstractBeanD
           efinition.DEPENDENCY_CHECK_NONE);
29     PropertyDescriptor[] filteredPds = null;
30
```

```

31     if (hasInstAwareBpps) { // 是否配置了后置处理器InstantiationAwareBean
32 PostProcessor
33         if (pvs == null) pvs = mbd.getPropertyValues();
34         for (InstantiationAwareBeanPostProcessor bp : getBeanPostProce
35 ssorCache().instantiationAware) {
36             // 执行InstantiationAwareBeanPostProcessor#postProcessPrope
37 rties(...)方法
38             PropertyValues pvsToUse = bp.postProcessProperties(pvs, bw
39 .getWrappedInstance(), beanName);
40             if (pvsToUse == null) {
41                 if (filteredPds == null)
42                     // 从给定的BeanWrapper中提取一组经过筛选的PropertyDescr
43 iptor, 排除忽略的依赖关系类型或在忽略的依赖接口上定义的属性。
44                     filteredPds = filterPropertyDescriptorsForDependen
45 cyCheck(bw, mbd.allowCaching);
46             // 执行InstantiationAwareBeanPostProcessor#postProcessP
47 ropertyValues(...)方法
48             pvsToUse = bp.postProcessPropertyValues(pvs, filteredP
49 ds, bw.getWrappedInstance(), beanName);
50             if (pvsToUse == null) return;
51         }
52         pvs = pvsToUse;
53     }
54 }
55
56 /** 步骤5: 执行依赖检查 */
57 if (needsDepCheck) { // 是否需要执行依赖检测操作
58     if (filteredPds == null) filteredPds = filterPropertyDescripto
59 rsForDependencyCheck(bw, mbd.allowCaching);
60     checkDependencies(beanName, mbd, filteredPds, pvs);
61 }
62
63 /** 步骤6: 将属性应用到bean中 */
64 if (pvs != null)
65     applyPropertyValues(beanName, mbd, bw, pvs);
66 }

```

8.4.1> autowireByName(...)根据名称进行注入

- 在传入的参数pvs中找出已经加载的bean，然后递归实例化相关bean，最后将其加入到pvs中。源码如下所示：

```
1  protected void autowireByName(String beanName, AbstractBeanDefinition
   mbd, BeanWrapper bw, MutablePropertyValues pvs) {
2      String[] propertyNames = unsatisfiedNonSimpleProperties(mbd, bw);
   // 寻找bw中需要依赖注入的属性
3      for (String propertyName : propertyNames) {
4          if (containsBean(propertyName)) {
5              Object bean = getBean(propertyName); // 递归初始化相关bean
6              pvs.add(propertyName, bean);
7              registerDependentBean(propertyName, beanName); // 注册依赖
8              if (logger.isTraceEnabled()) logger.trace(...);
9          }
10         else if (logger.isTraceEnabled()) logger.trace(...);
11     }
12 }
```

8.4.2> autowireByType(...)根据类型进行注入

- 由于需要根据类型进行注入，所以需要进行类型的解析和对比操作，相关的代码逻辑就变得复杂了，如下是相关源码：


```

1  protected void autowireByType(String beanName, AbstractBeanDefinition
   mbd, BeanWrapper bw, MutablePropertyValues pvs) {
2      TypeConverter converter = getCustomTypeConverter();
3      if (converter == null) converter = bw;
4
5      Set<String> autowiredBeanNames = new LinkedHashSet<>(4);
6      String[] propertyNames = unsatisfiedNonSimpleProperties(mbd, bw);
   // 寻找bw中需要依赖注入的属性
7      for (String propertyName : propertyNames) {
8          try {
9              PropertyDescriptor pd = bw.getPropertyDescriptor(propertyName);
10             if (Object.class != pd.getPropertyType()) {
11                 MethodParameter methodParam = BeanUtils.getWriteMethod
   Parameter(pd); // 探测指定属性的set方法
12                 boolean eager = !(bw.getWrappedInstance() instanceof P
   riorityOrdered);
13                 DependencyDescriptor desc = new AutowireByTypeDependen
   cyDescriptor(methodParam, eager);
14                 /**
15                  * 解析指定beanName的属性所匹配的值，并把解析到的属性名称存储在
   autowiredBeanNames中。
16                  * 当属性存在多个封装bean时 (@Autowired private List<A> aL
   ist) 将会找到所有匹配A类型的bean并将其注入进去
17                  */
18                 Object autowiredArgument = resolveDependency(desc, bea
   nName, autowiredBeanNames, converter);
19                 if (autowiredArgument != null)
20                     pvs.add(propertyName, autowiredArgument);
21                 for (String autowiredBeanName : autowiredBeanNames) {
22                     registerDependentBean(autowiredBeanName, beanName)
   ; // 注册依赖
23                     if (logger.isTraceEnabled()) logger.trace(...);
24                 }
25                 autowiredBeanNames.clear();
26             }
27         }
28         catch (BeansException ex) {throw new UnsatisfiedDependencyExce
   ption(...);}
29     }
30 }

```

【解释】Spring中提供了对集合类型注入的支持，如果使用注解的方式，则如下所示：

[@Autowired](#)

```
private List<Test> tests;
```

Spring将会把所有与Test匹配的类型找出来并注入到 `tests` 属性中，正式由于这一原因，所以在 `autowireByType(...)`方法中，新建了局部变量 `autowiredBeanNames`，用于存储所有依赖的bean，如果只是对非集合类的属性注入的话，那么这个属性就没啥用处了。

- 下面我们再来分析一下`resolveDependency(...)`方法，其源码如下所示：

```
▼ DefaultListableBeanFactory.java Java | 复制代码

1 public Object resolveDependency(DependencyDescriptor descriptor,
2                                 String requestingBeanName,
3                                 Set<String> autowiredBeanNames,
4                                 TypeConverter typeConverter) throws BeansException {
5     descriptor.initParameterNameDiscovery(getParameterNameDiscoverer());
6     if (Optional.class == descriptor.getDependencyType()) // Optional类型的特殊处理
7         return createOptionalDependency(descriptor, requestingBeanName);
8     else if (ObjectFactory.class == descriptor.getDependencyType() || // ObjectFactory类型的特殊处理
9             ObjectProvider.class == descriptor.getDependencyType()) // ObjectProvider类型的特殊处理
10        return new DependencyObjectProvider(descriptor, requestingBeanName);
11     else if (javaxInjectProviderClass == descriptor.getDependencyType()) // javaxInjectProviderClass类型的特殊处理
12        return new Jsr330Factory().createDependencyProvider(descriptor, requestingBeanName);
13     else {
14         Object result = getAutowireCandidateResolver().getLazyResolutionProxyIfNecessary(descriptor, requestingBeanName);
15         if (result == null) // 默认的getLazyResolutionProxyIfNecessary(...)方法返回null
16             // 通用处理逻辑
17             result = doResolveDependency(descriptor, requestingBeanName, autowiredBeanNames, typeConverter);
18         return result;
19     }
20 }
```

- 下面我们再来分析一下`doResolveDependency(...)`方法，其源码如下所示：

```
1  public Object doResolveDependency(DependencyDescriptor descriptor,
2                                     String beanName,
3                                     Set<String> autowiredBeanNames,
4                                     TypeConverter typeConverter) throws
    BeansException {
5      InjectionPoint previousInjectionPoint = ConstructorResolver.setCurrentInjectionPoint(descriptor);
6      try {
7          Object shortcut = descriptor.resolveShortcut(this);
8          if (shortcut != null) return shortcut;
9          Class<?> type = descriptor.getDependencyType();
10         /** 步骤1: 针对Spring中@Value注解的获取和解析 */
11         Object value = getAutowireCandidateResolver().getSuggestedValue(descriptor);
12         if (value != null) {
13             if (value instanceof String) {
14                 // 如果实现并注册了StringValueResolver接口的实现, 则调用resolveStringValue方法对value进行处理
15                 String strVal = resolveEmbeddedValue((String) value);
16                 BeanDefinition bd = (beanName != null && containsBean(beanName) ?
17                                     getMergedBeanDefinition(beanName) : null);
18                 // 如果配置了BeanExpressionResolver, 则对value值进行表达式
19                 解析
20                 value = evaluateBeanDefinitionString(strVal, bd);
21             }
22             // 获得类型转换器, 并对value值进行转换处理
23             TypeConverter converter = (typeConverter != null ? typeConverter : getTypeConverter());
24             try {
25                 return converter.convertIfNecessary(value, type, descriptor.getTypeDescriptor());
26             } catch (UnsupportedOperationException ex) {
27                 return (descriptor.getField() != null ?
28                     converter.convertIfNecessary(value, type, descriptor.getField()) :
29                     converter.convertIfNecessary(value, type, descriptor.getMethodParameter()));
30             }
31         }
32
33         /** 步骤2: 如果注入的是StreamDependencyDescriptor、Collection、Map、数组 */
34
```

```

35         Object multipleBeans = resolveMultipleBeans(descriptor, beanName,
36         autowiredBeanNames, typeConverter);
37         if (multipleBeans != null)
38             return multipleBeans;
39
40         Map<String, Object> matchingBeans = findAutowireCandidates(beanName,
41         type, descriptor);
42         if (matchingBeans.isEmpty()) {
43             if (isRequired(descriptor))
44                 raiseNoMatchingBeanFound(type, descriptor.getResolvableType(),
45                 descriptor);
46             return null;
47         }
48         String autowiredBeanName;
49         Object instanceCandidate;
50         if (matchingBeans.size() > 1) {
51             autowiredBeanName = determineAutowireCandidate(matchingBeans,
52             descriptor);
53             if (autowiredBeanName == null) {
54                 if (isRequired(descriptor) || !indicatesMultipleBeans(type))
55                     return descriptor.resolveNotUnique(descriptor.getResolvableType(),
56                     matchingBeans);
57                 else
58                     return null;
59             }
60             instanceCandidate = matchingBeans.get(autowiredBeanName);
61         }
62         else {
63             Map.Entry<String, Object> entry = matchingBeans.entrySet().iterator().next();
64             autowiredBeanName = entry.getKey();
65             instanceCandidate = entry.getValue();
66         }
67         if (autowiredBeanNames != null)
68             autowiredBeanNames.add(autowiredBeanName);
69         if (instanceCandidate instanceof Class)
70             instanceCandidate = descriptor.resolveCandidate(autowiredBeanName, type,
71             this);
72         Object result = instanceCandidate;
73         if (result instanceof NullBean) {
74             if (isRequired(descriptor))
75                 raiseNoMatchingBeanFound(type, descriptor.getResolvableType(),
76                 descriptor);
77             result = null;
78         }

```

```

73         if (!ClassUtils.isAssignableValue(type, result)) throw new Bea
74         nNotOfRequiredTypeException(...);
75         return result;
76     } finally {
77         ConstructorResolver.setCurrentInjectionPoint(previousInjection
Point);
    }
}

```

【解释】寻找类型的匹配执行顺序是，首先尝试使用解析器进行解析，如果解析器没有成功解析，那么可能是使用默认的解析器没有做任何处理，或者是使用了自定义的解析器，但是对于集合等类型来说并不在解析范围之内，所以再次对不同类型进行不同情况的处理，虽然说对于不同类型处理的方式不一致，但是大致的思路还是相似的。

8.4.3> applyPropertyValues(...)

- 程序运行到这里，已经完成了对所有注入属性的获取，但是**获取的属性是以 `PropertyValues` 形式存在的，并没有应用到已经实例化的bean中**，这项工作是在 `applyPropertyValues(...)` 方法中实现的，具体源码如下所示：

```
1  protected void applyPropertyValues(String beanName, BeanDefinition mbd
   , BeanWrapper bw, PropertyValues pvs) {
2      if (pvs.isEmpty()) return;
3
4      if (System.getSecurityManager() != null && bw instanceof BeanWrapp
   erImpl)
5          ((BeanWrapperImpl) bw).setSecurityContext(getAccessControlCont
   ext());
6
7      MutablePropertyValues mpvs = null;
8      List<PropertyValue> original;
9      if (pvs instanceof MutablePropertyValues) {
10         mpvs = (MutablePropertyValues) pvs;
11         if (mpvs.isConverted()) { // 如果mpvs中的值已经被转换为对应的类型,
   那么可以直接设置到bw中
12             try {
13                 bw.setPropertyValues(mpvs);
14                 return;
15             }
16             catch (BeansException ex) {throw new BeanCreationException
   (...);}
17         }
18         original = mpvs.getPropertyValueList();
19     } else
20         // 如果pvs不是MutablePropertyValues类型, 那么直接使用原始的属性获取方
   法
21         original = Arrays.asList(pvs.getPropertyValues());
22
23     TypeConverter converter = getCustomTypeConverter();
24     if (converter == null) converter = bw;
25
26     // 获取对应的解析器
27     BeanDefinitionValueResolver valueResolver = new BeanDefinitionValu
   eResolver(this, beanName, mbd, converter);
28     List<PropertyValue> deepCopy = new ArrayList<>(original.size());
29     boolean resolveNecessary = false;
30     // 遍历属性, 将其转换为对应类的对应属性类型
31     for (PropertyValue pv : original) {
32         if (pv.isConverted()) deepCopy.add(pv);
33         else {
34             String propertyName = pv.getName();
35             Object originalValue = pv.getValue();
36             if (originalValue == AutowiredPropertyMarker.INSTANCE) {
37                 Method writeMethod = bw.getPropertyDescriptor(propertyName).getWriteMethod();
```

```

38         if (writeMethod == null) throw new IllegalArgumentException(
39             ption(...);
40             originalValue = new DependencyDescriptor(new MethodParameter(
41                 writeMethod, 0), true);
42             }
43             Object resolvedValue = valueResolver.resolveValueIfNecessary(
44                 pv, originalValue); // 执行类型转换
45             Object convertedValue = resolvedValue;
46             boolean convertible = bw.isWritableProperty(propertyName)
47             &&
48                 !PropertyAccessorUtils.isNestedOrIndexedProperty(propertyName);
49             if (convertible)
50                 convertedValue = convertForProperty(resolvedValue, propertyName, bw, converter);
51             if (resolvedValue == originalValue) {
52                 if (convertible) pv.setConvertedValue(convertedValue);
53                 deepCopy.add(pv);
54             }
55             else if (convertible && originalValue instanceof TypedStringValue &&
56                 !((TypedStringValue) originalValue).isDynamic() &&
57                 !(convertedValue instanceof Collection || ObjectUtils.isArray(convertedValue))) {
58                 pv.setConvertedValue(convertedValue);
59                 deepCopy.add(pv);
60             }
61             else {
62                 resolveNecessary = true;
63                 deepCopy.add(new PropertyValue(pv, convertedValue));
64             }
65         }
66         if (mpvs != null && !resolveNecessary)
67             mpvs.setConverted();
68         try {
69             bw.setPropertyValues(new MutablePropertyValues(deepCopy));
70         } catch (BeansException ex) {throw new BeanCreationException(...);
71     }
72 }

```

8.5> initializeBean(...)初始化bean

- 这个方法主要是针对我们配置的 `init-method` 属性，当Spring中程序已经执行过bean的实例化，并且进行了属性的填充，而就在这时将会调用用户设定的初始化方法。具体源码如下所示：

```
AbstractAutowireCapableBeanFactory.java  Java | 复制代码

1  protected Object initializeBean(String beanName, Object bean, @Nullable
    RootBeanDefinition mbd) {
2      if (System.getSecurityManager() != null) {
3          AccessController.doPrivileged((PrivilegedAction<Object>) () -
        > {
4              invokeAwareMethods(beanName, bean);
5              return null;
6          }, getAccessControlContext());
7      }
8      else invokeAwareMethods(beanName, bean);
9
10     Object wrappedBean = bean;
11     if (mbd == null || !mbd.isSynthetic())
12         // 调用配置的所有BeanPostProcessor#postProcessBeforeInitializati
        on方法
13         wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrap
        pedBean, beanName);
14
15     try {
16         invokeInitMethods(beanName, wrappedBean, mbd); // 激活用户自定义
        的init方法
17     } catch (Throwable ex) {throw new BeanCreationException(...);}
18
19     if (mbd == null || !mbd.isSynthetic())
20         // 调用配置的所有BeanPostProcessor#postProcessAfterInitializatio
        n方法
21         wrappedBean = applyBeanPostProcessorsAfterInitialization(wrapp
        edBean, beanName);
22
23     return wrappedBean;
24 }
```

8.5.1> invokeAwareMethods(...)激活Aware方法

- Spring中提供了一些Aware接口实现，比如：`BeanFactoryAware`、`ApplicationContextAware`、`ResourceLoaderAware`、`ServletContextAware`等，实现这些Aware接口的bean在被初始化之后，可以取得一些相对的资源。我们可以通过示例来了解一下Aware的用法。


```
▼ Hello.java Java | 复制代码

1 public class Hello {
2     public void say() {
3         System.out.println("hello");
4     }
5 }
```

```
▼ Test.java Java | 复制代码

1 public class Test implements BeanFactoryAware {
2     private BeanFactory beanFactory;
3
4     // 声明bean的时候，Spring会自动注入BeanFactory实例
5     @Override
6     public void setBeanFactory(BeanFactory beanFactory) throws BeansEx
7         ception {
8         this.beanFactory = beanFactory;
9     }
10
11     public void testAware() {
12         // 通过hello这个bean，从BeanFactory中获得实例
13         Hello hello = (Hello) beanFactory.getBean("hello");
14         hello.say();
15     }
16 }
```

```
16 @Slf4j
17 @SpringBootApplication
18 public class SpringbootDemoApplication {
19     public static void main(String[] args) throws Throwable {
20         XmlBeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("oldbean.xml"));
21         Test test = (Test) beanFactory.getBean("test");
22         test.testAware();
23     }
24 }
```

Console Endpoints

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...
21:35:03.876 [main] DEBUG org.springframework.beans.factory.xml.XmlBeanDefinitionReader - Loaded 6 bean defini
21:35:03.881 [main] DEBUG org.springframework.beans.factory.xml.XmlBeanFactory - Creating shared instance of
21:35:03.888 [main] DEBUG org.springframework.beans.factory.xml.XmlBeanFactory - Creating shared instance of
hello
Process finished with exit code 0
```

- 按照上面的方法我们可以获取到Spring中的BeanFactory，并且可以根据BeanFactory获取所有的bean，以及进行相关设置。当然还有其他Aware的使用方法也都是大同小异的，此时，我们再来看一下invokeAwareMethods(...)的源码实现：

```
1 private void invokeAwareMethods(String beanName, Object bean) {
2     if (bean instanceof Aware) {
3         if (bean instanceof BeanNameAware)
4             ((BeanNameAware) bean).setBeanName(beanName); // 向BeanNameAware中注入beanName
5
6         if (bean instanceof BeanClassLoaderAware) {
7             ClassLoader bcl = getBeanClassLoader();
8             if (bcl != null)
9                 ((BeanClassLoaderAware) bean).setBeanClassLoader(bcl);
10            // 向BeanClassLoaderAware中注入classLoader
11        }
12        if (bean instanceof BeanFactoryAware) // 向BeanFactoryAware中注入beanFactory
13            ((BeanFactoryAware) bean).setBeanFactory(AbstractAutowireCapableBeanFactory.this);
14    }
15 }
```

8.5.2> invokeInitMethods(...)激活自定义的init方法

- 客户定制的初始化方法除了我们熟知的使用配置 `init-method` 外，还有使自定义的bean实现 `InitializingBean` 接口，并在 `afterPropertiesSet()` 方法中实现自己的初始化业务逻辑。其中，`InitializingBean`的`afterPropertiesSet()`方法先被执行，而`init-method`后执行。下面是相关源码实现：

```

1  protected void invokeInitMethods(String beanName, Object bean, @Nullable RootBeanDefinition mbd) {
2      boolean isInitializingBean = (bean instanceof InitializingBean);
3      if (isInitializingBean && (mbd == null || !mbd.isExternallyManagedInitMethod("afterPropertiesSet"))) {
4          if (logger.isTraceEnabled()) logger.trace(...);
5
6          if (System.getSecurityManager() != null) {
7              try {
8                  AccessController.doPrivileged((PrivilegedExceptionAction<Object>) () -> {
9                      ((InitializingBean) bean).afterPropertiesSet();
10                     // 属性初始化后的处理
11                     return null;
12                 }, getAccessControlContext());
13             } catch (PrivilegedActionException pae) {throw pae.getException();}
14         }
15         // 属性初始化后的处理, 调用InitializingBean#afterPropertiesSet()方法
16         else ((InitializingBean) bean).afterPropertiesSet();
17     }
18     if (mbd != null && bean.getClass() != NullBean.class) {
19         String initMethodName = mbd.getInitMethodName();
20         if (StringUtils.hasLength(initMethodName) &&
21             !(isInitializingBean &&
22               "afterPropertiesSet".equals(initMethodName)) &&
23             !mbd.isExternallyManagedInitMethod(initMethodName)) {
24             // 调用自定义的init-method方法
25             invokeCustomInitMethod(beanName, bean, mbd);
26         }
27     }
28 }

```

- 调用自定义的 `init-method` 方法，源码如下所示：

```
1  protected void invokeCustomInitMethod(String beanName, Object bean, RootBeanDefinition mbd) throws Throwable {
2      String initMethodName = mbd.getInitMethodName(); // 获得init-method方法
3      Assert.state(initMethodName != null, "No init method set");
4
5      /** 步骤1: 获得init-method对应的Method实例对象 */
6      Method initMethod = (mbd.isNonPublicAccessAllowed() ?
7          BeanUtils.findMethod(bean.getClass(), initMethodName) :
8          ClassUtils.getMethodIfAvailable(bean.getClass(), initMethodName));
9      if (initMethod == null)
10         if (mbd.isEnforceInitMethod()) throw new BeanDefinitionValidationException(...);
11         else return;
12      Method methodToInvoke = ClassUtils.getInterfaceMethodIfPossible(initMethod);
13
14      /** 步骤2: 通过反射, 执行init-method的方法调用 */
15      if (System.getSecurityManager() != null) {
16          AccessController.doPrivileged((PrivilegedAction<Object>) () -
17              > {
18                  ReflectionUtils.makeAccessible(methodToInvoke);
19                  return null;
20              });
21          try {
22              AccessController.doPrivileged((PrivilegedExceptionAction<Object>)
23                  () -> methodToInvoke.invoke(bean), getAccessControllerContext());
24          } catch (PrivilegedActionException pae) {...}
25      } else {
26          try {
27              ReflectionUtils.makeAccessible(methodToInvoke);
28              methodToInvoke.invoke(bean); // 通过反射, 执行init-method的方法调用
29          }
30          catch (InvocationTargetException ex) {throw ex.getTargetException();}
31      }
32  }
```

8.6> registerDisposableBeanIfNecessary(...)注册DisposableBean

- Spring同时也提供了销毁方法的扩展入口，对于销毁方法的扩展，除了我们熟知的配置属性`destroy-method`方法外，用户还可以注册后处理器`DestructionAwareBeanPostProcessor`来统一处理bean的销毁方法，具体源码如下所示：

```
AbstractBeanFactory.java      Java | 复制代码

1  protected void registerDisposableBeanIfNecessary(String beanName, Object bean, RootBeanDefinition mbd) {
2      AccessControlContext acc = (System.getSecurityManager() != null ?
3      getAccessControlContext() : null);
4      if (!mbd.isPrototype() && requiresDestruction(bean, mbd)) {
5          /**
6           * 单例模式下注册需要销毁的bean，此方法中会处理实现DisposableBean的bean，
7           * 并且对所有的bean使用DestructionAwareBeanPostProcessor处理
8           */
9          if (mbd.isSingleton())
10             registerDisposableBean(beanName, new DisposableBeanAdapter
11             (bean,
12             beanName,
13             mbd,
14             getBeanPostProcessorCache().destructionAware,
15             acc));
16         else { // 自定义scope的处理
17             Scope scope = this.scopes.get(mbd.getScope());
18             if (scope == null) throw new IllegalStateException(...);
19             scope.registerDestructionCallback(beanName, new Disposable
20             BeanAdapter(bean,
21             beanName,
22             mbd,
23             getBeanPostProcessorCache().destructionAware,
24             acc));
25         }
26     }
27 }
```

吾尝终日而思矣，不如须臾之所学也；

吾尝跂而望矣，不如登高之博见也。

登高而招，臂非加长也，而见者远；

顺风而呼，声非加疾也，而闻者彰。

假舆马者，非利足也，而致千里；

假舟楫者，非能水也，而绝江河。

君子生非异也，善假于物也。

----- 摘自《劝学》

愿本文可以成为大家的“山”、“风”、“马”、“舟”，助大家在技术之路上乘风破浪，大展宏图~~

同时，也欢迎大家关注我的公众号“[爪哇缪斯](#)”~\(^o^)/~ 「干货分享，每天更新」



微信搜一搜



爪哇缪斯