Spring 源码初探

> 本文基于 jdk 11
>
> 文章已收录我的仓库：Java学习笔记

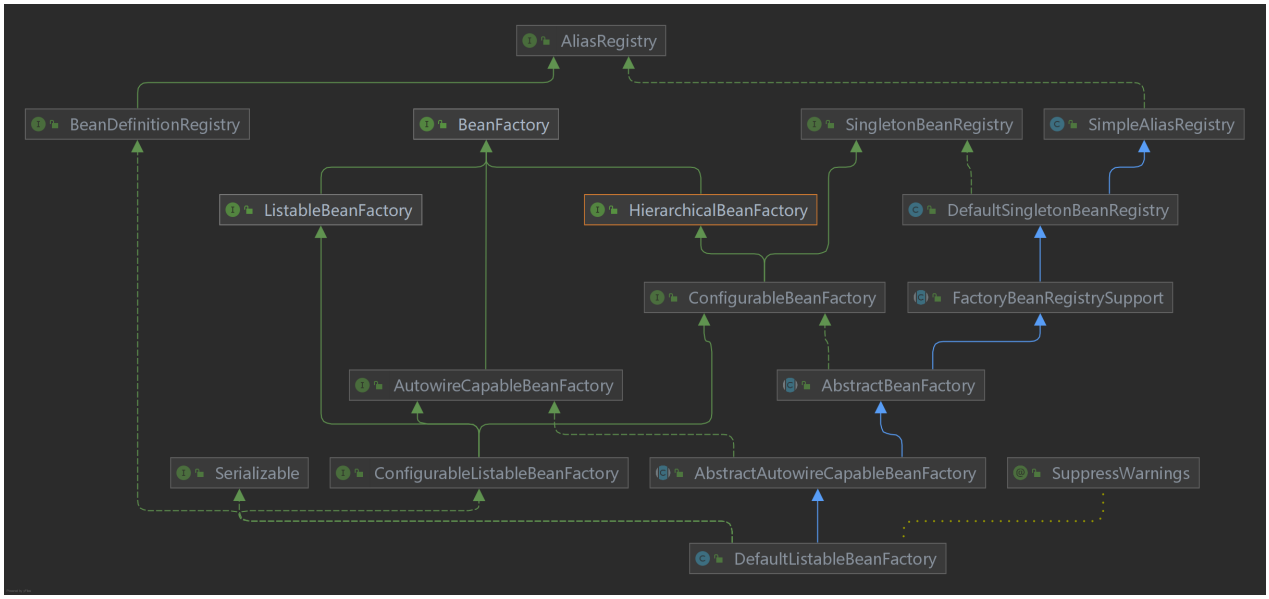
核心类

interface BeanFactory

该接口是访问 Spring bean 容器的根接口，是 bean 容器的基本客户端视图； 其他接口如ListableBeanFactory和ConfigurableBeanFactory可用于扩展一些其他功能。

简单来说，该类就是“容器”接口类，用以存放 bean，其内定义了一系列 getBean 方法，是一个**存粹**的 “bean 生产地”。

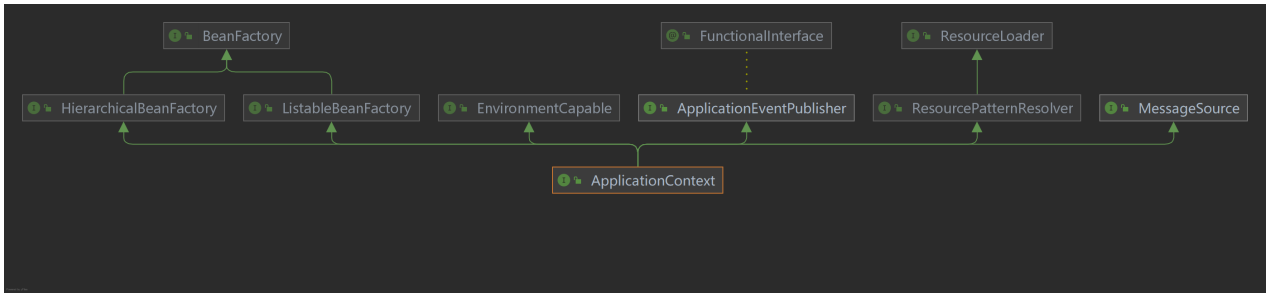
class DefaultListableBeanFactory



如果说 BeanFactory 是一个简单存粹的 “bean 生产地”，那么 DefaultListableBeanFactory 就是一个庞大而复杂的 “bean 生产机器”，DefaultListableBeanFactory 不仅实现了 BeanFactory 接口，还实现了其他与 bean 相关的接口，例如别名相关、BeanDefinition等，**这个类是 spring 默认使用的 bean 工厂**，但它已经不像 BeanFactory 那么存粹了。

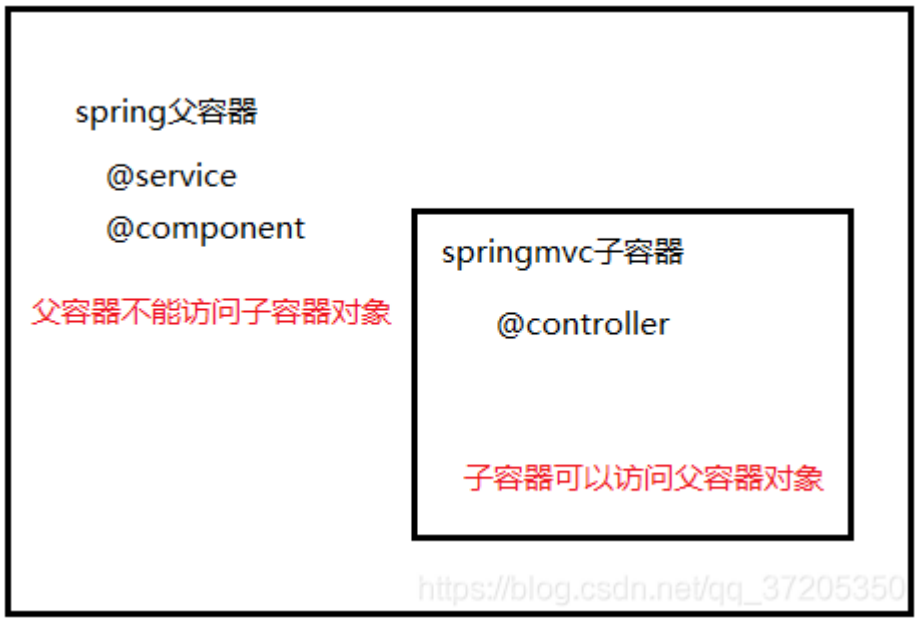
如无特殊说明，下文说所 bean 工厂或 BeanFactory 均是指 DefaultListableBeanFactory。

interface ApplicationContext



该类是 IOC 的中央接口，该接口类继承了 BeanFactory 并且实现了更多的接口，即具备更完善的功能，例如：

- 用于访问应用程序组件的 Bean 工厂方法，继承自 **ListableBeanFactory**，默认使用 **DefaultListableBeanFactory** 工厂 。
- 以通用方式加载文件资源的能力，继承自org.springframework.core.io.ResourceLoader接口。
- 能够将事件发布到注册的侦听器，继承自ApplicationEventPublisher接口。
- 解析消息的能力，支持国际化，继承自MessageSource接口。
- 从父上下文继承，例如，单个父上下文可以被整个 Web 应用程序使用，而每个 servlet 都有自己的子上下文，该子上下文独立于任何其他 servlet 的子上下文，例如 Spring 容器是 SpringMVC 的父容器，SpringMVC 容器可以访问 Spring 中所有 Bean。



interface BeanDefinition

BeanDefinition 描述了一个 bean 实例，它具有 bean 的属性值、构造函数参数值以及其他信息(由具体实现中实现)，简单的将该类就是存放在 bean 的元数据（还未实例化），以便我们后续创建 bean 实例。

interface BeanDefinitionReader

读取 BeanDefinition 信息，例如可以有 XML 形式读取（XmlBeanDefinitionReader）、配置文件读取（PropertiesBeanDefinitionReader）、注解读取（AnnotatedBeanDefinitionReader ）或由配置类读取（ConfigurationClassBeanDefinitionReader）。

> 在 BeanDefinitionLoader 类中的 load 方法有具体的判断

```
private int load(Object source) {
   Assert.notNull(source, "Source must not be null");
   if (source instanceof Class<?>) { // 这是一个 Class<>？
      return load((Class<?>) source);
   }
   if (source instanceof Resource) { // 这是一个资源？
      return load((Resource) source);
   }
   if (source instanceof Package) { // 这是一个包？
      return load((Package) source);
   }
   if (source instanceof CharSequence) { // 一个字符串？例如 XML 文件名
      return load((CharSequence) source);
   }
   throw new IllegalArgumentException("Invalid source type " +
source.getClass());
}
```

interface Aware

一个标记超级接口，指示 bean 有资格通过回调样式的方法由特定框架对象的 Spring 容器通知。

简单的来讲，aware 就是 bean 的额外的一些属性，例如你想知道某个 bean 的 name(id)，则你可以实现让这个 bean BeanNameAware 接口，该接口只有一个 setBeanName 方法，spring 在初始化 bean 的时候会判断该 bean 是否实现了某个具体的 Aware 接口（例如，通过 instanceof），如果是的话则调用 set 方法注入属性。

```
@Component("我是 beanName")
public class Bean implements BeanNameAware {
    public String name;

    @Override
    public void setBeanName(String name) {
        this.name = name;
    }
}
```

Test 代码：

```
@Autowired
Bean bean;
@Test
void contextLoads() {
    System.out.println(bean);
}

// 输出：Bean{name='我是 beanName'}
```

interface BeanFactoryPostProcessor

当所有的 BeanDefinitionReader 加载完 BeanDefinition 到 BeanFactory 后，Spring 执行每一个注册的 BeanFactoryPostProcessor 的 **`postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)`方法**，该方法被认为是一个**增强器**，允许对 BeanDefinition 信息做额外的修改或注册 BeanDefinition。

例如由 `@Component、@Bean` 等标记的 bean 就是在这一阶段被增强器扫描并注册 BeanDefinition 加载到 BeanFactory 中。

> 详细请参考 SpringBoot 源码总结

例如，我们可以自己实现一个简易的注解`myBean`，使得被该注解标记的类都能成为 Spring Bean：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface myBean {
    String beanName() default "";
}
```

添加注解：

```
@myBean(beanName = "abc")
public class Bean {
}
```

注册 BeanFactoryPostProcessor，扫描所有被 @myBean 标记的类，并注册 BeanDefinition：

```
@Component
public class MyBeanFactoryPostProcessor implements
BeanFactoryPostProcessor {
    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory
beanFactory) throws BeansException {
        Enumeration<URL> resources = null;
        String basePackage = "com.happysnaker";
        try {
            resources =
Thread.currentThread().getContextClassLoader().getResources(basePackage.r
eplaceAll("\\.", "/"));
        } catch (IOException e) {
            e.printStackTrace();
        }
        while (resources.hasMoreElements()) {
            URL resource = resources.nextElement();
```

```
            String protocol = resource.getProtocol();
            if ("file".equals(protocol)) {
                String filePath = null;
                try {
                    filePath = URLDecoder.decode(resource.getFile(),
"UTF-8");
                } catch (UnsupportedEncodingException e) {
                    e.printStackTrace();
                }
                try {
                    // 扫描 com.happysnaker 包下的所有类
                    List<Class> classes = getAllClass(new File(filePath),
basePackage);
                    for (Class aClass : classes) {
                        doWork(beanFactory, aClass);
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }

    }

    private void doWork(ConfigurableListableBeanFactory beanFactory,
Class c) {
        // 如果是被 @myBean 标记的话
        if (c.isAnnotationPresent(myBean.class)) {
            myBean annotation = (myBean) c.getAnnotation(myBean.class);
            BeanDefinitionBuilder builder =
BeanDefinitionBuilder.genericBeanDefinition(c);
            try {
                BeanDefinitionRegistry registry =
(BeanDefinitionRegistry) beanFactory;
                // 注册 BeanDefinition
                registry.registerBeanDefinition(annotation.beanName(),
builder.getBeanDefinition());
            } catch (Exception e) {
                e.printStackTrace();
            }

        }
    }

    private List<Class> getAllClass(File file, String path) throws
IOException {
        List<Class> ans = new ArrayList<>();
        if (file.isDirectory()) {
            File[] files = file.listFiles();
            for (File file1 : files) {
                List<Class> classList = getAllClass(file1, path + "." +
file1.getName());
                if (classList != null) {
                    ans.addAll(classList);
                }
```

```
            }
        } else {
            if (file.getName().indexOf(".class") != -1) {
                path = path.substring(0, path.indexOf(".class"));
                try {
                    Class c = Class.forName(path);
                    ans.add(c);
                } catch (ClassNotFoundException e) {
                    e.printStackTrace();
                }
            }
        }
        return ans;
    }
}
```

测试：

```
@Qualifier("abc")
@Autowired
Bean bean;
@Test
void contextLoads() {
    System.out.println(bean);
}

// 输出：com.happysnaker.bean.Bean@12fcc71f
```

interface BeanDefinitionRegistryPostProcessor

该方法与 BeanFactoryPostProcessor 类一个意思，其本身就实现了 BeanFactoryPostProcessor，这个类下有一个方法 postProcessBeanDefinitionRegistry，实现了这个方法的类会与 postProcessBeanFactory 方法在同一个函数内执行。

事实上由于 BeanDefinitionRegistryPostProcessor 继承了 BeanFactoryPostProcessor，我们会认为 BeanDefinitionRegistryPostProcessor 就是一个 BeanFactoryPostProcessor，下文中以及后续文章将不会再出现 BeanDefinitionRegistryPostProcessor 这个名词。

interface BeanPostProcessor

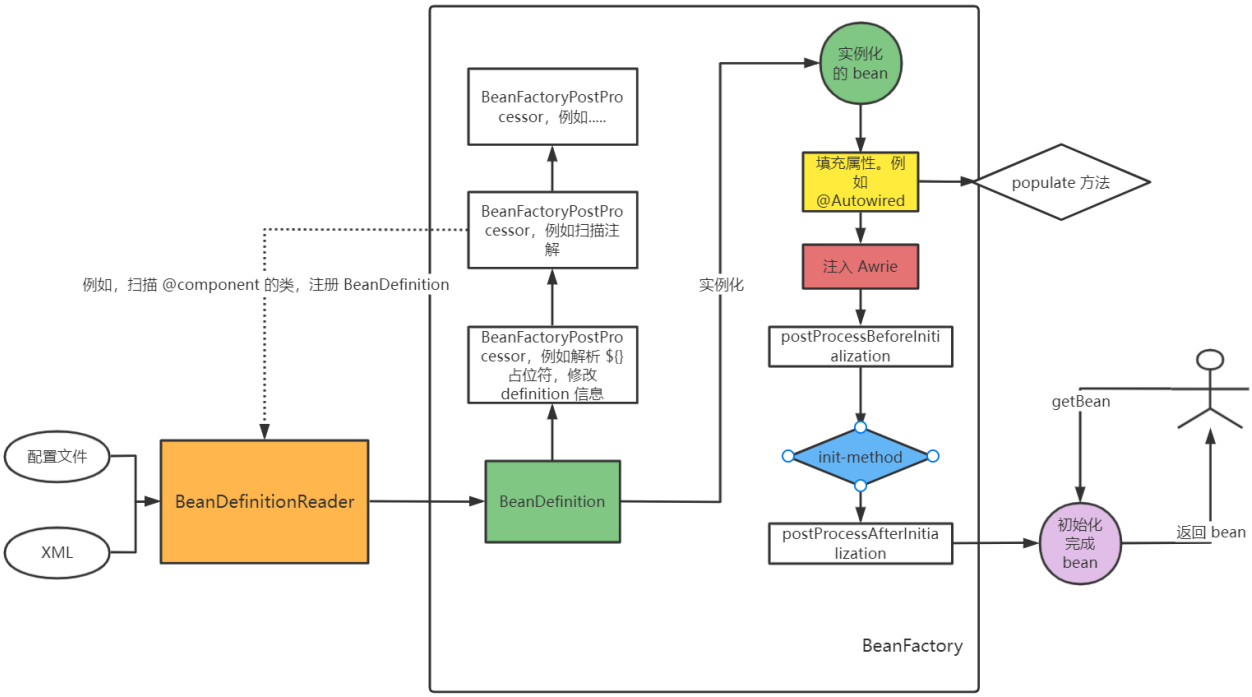
该接口中有两个方法：

```
Object postProcessBeforeInitialization(Object bean, String beanName);
Object postProcessAfterInitialization(Object bean, String beanName);
```

与 BeanFactoryPostProcessor 不同的是，BeanFactoryPostProcessor 在 bean 实例化之前被调用，而 BeanPostProcessor 在 bean 实例化之后、**初始化前后调用**，注入 @Value、@AutoWired、AOP 均依赖该类实现。

生命周期

Spring 生命周期通常被认为包括 **实例化 和 初始化两个步骤**，实例化通常是通过反射 newInstance 构建对象，而初始化包括设置对象属性、注入 val（例如 AutoWired）等，执行 init-method 方法，设置代理类等操作，你可以简单的认为实例化只是简单的**创建类**，而初始化是**包装类**。



刷新生命周期与实例化

打开断点调试，进入 AbstractApplicationContext 抽象类的 **refresh()** 方法，该方法是 Spring 启动的核心步骤，由 12 个方法构成：

prepareRefresh()

容器刷新前的准备，设置上下文状态为激活状态，开始启动计时，获取属性，验证必要的属性等。

obtainFreshBeanFactory()

跟踪源码发现，最终调用了 **AbstractRefreshableApplicationContext 类的 refreshBeanFactory 方法**，该方法销毁原有 beanFactory，获取新的 beanFactory，通过断点调试确定 beanFactory 是 DefaultListableBeanFactory，同

时还会 loadBeanDefinitions，跟踪该方法发现调用了 XmlBeanDefinitionReader 类解析 XML 文件，生成 BeanDefinition。

prepareBeanFactory(beanFactory)

配置上下文的 ClassLoader，设置 SpEL 表达式解析器，添加忽略注入的接口，添加存在的 bean 以及 BeanFactoryPostProcessors 等。

postProcessBeanFactory(beanFactory);

允许子类继承 AbstractApplicationContext 并扩展该方法。

invokeBeanFactoryPostProcessors(beanFactory)

实例化并调用所有注册的 beanFactory 后置处理器，跟踪源码发现调用了 **PostProcessorRegistrationDelegate 类的 invokeBeanFactoryPostProcessors 方法**，该方法获取所有实现了 BeanFactoryPostProcessor 接口的 bean，然后调用增强器方法，首先会调用被 @PriorityOrdered 标记的方法 ，再调用被 @Ordered 标记的方法，最后调用普通方法。

我们前面已经提到了这一步将会解析一些注解标注的 bean，事实上 ConfigurationClassPostProcessor 增强器的增强方法中会调用这样一种方法：`enhanceConfigurationClasses(beanFactory);`，该方法会**将所有用@Configuration 注解修饰的类用 cglib 技术代理加强，这样做的目的是为了解决单例问题**，例如一个 Configuration 配置类下面可能会调用两次 @Bean 方法，这违反了单例原则，因此通过增强代理来避免这种情况发送。

registerBeanPostProcessors(beanFactory)

实例化和注册 beanFactory 中扩展了 `BeanPostProcessor` 的 bean，但**并不执行**，而是等到初始化时执行。

initMessageSource()

初始化国际化工具类 MessageSource。

initApplicationEventMulticaster()

初始化事件广播器。

onRefresh()

模板方法，在容器刷新的时候可以自定义逻辑，不同的Spring容器做不同的事情，在 SpringBoot 中，容器为 `AnnotationConfigServletWebServerApplicationContext`

（继承自 `ServletWebServerApplicationContext`），这个容器刷新上下文时创建并启动了 `tomcat` 。

registerListeners()

注册监听器，监听 early application events。

finishBeanFactoryInitialization(beanFactory)

实例化和初始化所有剩余的（非懒加载）单例类，比如 invokeBeanFactoryPostProcessors 方法中根据各种注解解析出来的类，在这个时候都会被实例化和初始化。

跟踪源码，发现最终调用：

```
// Instantiate all remaining (non-lazy-init) singletons.
beanFactory.preInstantiateSingletons();
```

继续跟踪源码，preInstantiateSingletons 方法中进入如下语句块：

```
// 早期对象
if (isEagerInit) {
    getBean(beanName);
}
```

点进去，来到 doGetBean 方法，最终进入到：

```
if (mbd.isSingleton()) {
    sharedInstance = getSingleton(beanName, () -> {
        try {
            return createBean(beanName, mbd, args);
        }
    });
}
```

点进 createBean 方法，一步步跟踪来到 AbstractAutowireCapableBeanFactory 类的 doCreateBean 方法，然后进入到 instantiateBean 方法，然后进入到 instantiate 方法，该方法中执行：

```
constructorToUse = clazz.getDeclaredConstructor();
```

然后构造器传入到 BeanUtils.instantiateClass 方法，该方法中直接实例化对象：

```
return ctor.newInstance(argsWithDefaultValues);
```

当然，上述只是无参构造器的流程，如果需要 `@Autowired`，Spring 会执行 `determineConstructorsFromBeanPostProcessors(beanClass, beanName);` 方法会拿到构造函数列表，这些构造函数所需的参数都在 Bean 工厂中，随后会 `ctors = mbd.getPreferredConstructors();` 决出最好的构造函数，默认是全长的构造函数，然后进行实例化。

实例被初始化，总算解开了我一直以来的疑惑。

finishRefresh()

refresh做完之后需要做的一些事情。例如，清除上下文资源缓存（如扫描中的ASM元数据），发布ContextRefreshedEvent 事件告知对应的 ApplicationListener 进行响应的操作。

初始化

循环依赖问题

即 A 依赖与 B，同时 B 依赖于 A：

```
@Component
public class B {
    @Autowired
    A a;

    public B(A a) {
        this.a = a;
    }
}

@Component
public class A {
    @Autowired
    B b;

    public A(B b) {
        this.b = b;
    }
}
```

那么想初始化 A，就要填充 B，而 B 未被创建，就会去递归的创建 B，然后初始化 B，想要初始化 B，就要填充 A，而 A 未被创建，就会去递归的创建 A，那么...

其实解决的办法也很简单，如果 A、B 提供了 set 方法的话：

```
A a = new A();
B b = new B();
```

```
b.setA(a);
a.setB(b);
```

这种思想叫**提前暴露对象**，例如 b 注入了一个不完整的 a，Spring 也是基于这种思想解决循环依赖的。

循环依赖源码跟踪

Spring 是基于三级缓存解决循环依赖。

一级缓存 Map<String, Object>，key 是 beanName，val 是**已创建完成**的单例 bean 对象。

二级缓存 Map<String, Object>，key 是 beanName，val 是**未创建完成**的单例 bean 对象，即已实例化但未初始化的对象，例如，上面示例代码中的 a。

二级缓存 Map<String, ObjectFactory>，key 是 beanName，val 是一个函数式接口，其中调用 getObject 方法获取对象。

我们模拟 A、B 循环以来问题。

这里先初始化 A 对象。

初始化的逻辑也在 **finishBeanFactoryInitialization(beanFactory)** 中，我们按照上面步骤同样来到 doGetBean 方法，此时我们注意代码：

```
// Eagerly check singleton cache for manually registered singletons.
Object sharedInstance = getSingleton(beanName);
```

spirng 会尝试从缓存中获取对象，getSingleton 会一级一级的去判断是否有缓存(先判断一级)，当然这里 A 对象肯定不在缓存，因此会先实例化 A 对象。

```
// singletonObject 一级；  earlySingletonObjects 二级；singletonFactories
三级；
protected Object getSingleton(String beanName, boolean
allowEarlyReference) {
    Object singletonObject = this.singletonObjects.get(beanName);
    if (singletonObject == null &&
isSingletonCurrentlyInCreation(beanName)) {
        singletonObject = this.earlySingletonObjects.get(beanName);
        if (singletonObject == null && allowEarlyReference) {
            synchronized (this.singletonObjects) {
               singletonObject = this.singletonObjects.get(beanName);
                if (singletonObject == null) {
                    singletonObject =
this.earlySingletonObjects.get(beanName);
                    if (singletonObject == null) {
                        ObjectFactory<?> singletonFactory =
this.singletonFactories.get(beanName);
```

```
                        if (singletonFactory != null) {
                            singletonObject =
singletonFactory.getObject();
                            this.earlySingletonObjects.put(beanName,
singletonObject);
                            this.singletonFactories.remove(beanName);
                        }
                    }
                }
            }
        }
        return singletonObject;
    }
```

注意还发现如果三级缓存取出对象，会添加至二级缓存，同时移除三级缓存。

让我们套用 **finishBeanFactoryInitialization(beanFactory)** 讲解，当我们拿到了反射创建的 a 实例之后，回到 doCreateBean 中继续执行：

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

注意看 addSingletonFactory 方法，该方法将 a 的对象工厂(函数式接口)添加到三级缓存中，也就是说**调用三级缓存工厂的 getObject 方法实际上会调用 getEarlyBeanReference 方法**：

```
protected Object getEarlyBeanReference(String beanName,
RootBeanDefinition mbd, Object bean) {
    Object exposedObject = bean;
    // 是否有 beanPostProcessors 增强处理器，如果有，则使用增强处理器返回的对象
    if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors())
{
        for (BeanPostProcessor bp : getBeanPostProcessors()) {
            if (bp instanceof SmartInstantiationAwareBeanPostProcessor) {
                SmartInstantiationAwareBeanPostProcessor ibp =
(SmartInstantiationAwareBeanPostProcessor) bp;
                exposedObject = ibp.getEarlyBeanReference(exposedObject,
beanName);
            }
        }
    }
    return exposedObject;
}
```

getEarlyBeanReference 方法事实上就是返回了 bean 对象，如果需要加强，就返回加强后的结果，**即返回最终版本的 bean**，如果有代理的话，这里会返回代理作为 bean。

回到 doCreateBean:

```
Object exposedObject = bean;
try {
    populateBean(beanName, mbd, instanceWrapper);
    exposedObject = initializeBean(beanName, exposedObject, mbd);
}
```

populateBean 是填充属性的方法，**A 对象需要填充 B 对象**。进入之后跟踪代码，最终调用了 `applyPropertyValues(beanName, mbd, bw, pvs)` 方法，然后调用 BeanDefinitionValueResolver 类的 resolveValueIfNecessary 方法，进而进入了 resolveReference 方法：

```
bean = parent.getBean(String.valueOf(doEvaluate(ref.getBeanName())));
```

这就是递归的步骤了，这里回到了最开始 getBean - doGetBean - createBean - doCreateBean...这里递归的去实例化并初始化 B 对象。

注意，`@Autowired` 逻辑也是在填充属性这里被解析的。

最终 B 对象同样需要填充 A 对象，于是又递归的去调用 getBean 生产 A 对象，然后...

我们说了 spirng 会尝试从缓存中获取对象，此时三级缓存中已经有 A 对象了，因此会直接取出 A 对象，取出 A 对象之后就不会走我们之前的分支了，此时会直接返回 A 对象！

于是乎 B 对象成功填充 A，于是返回 B，于是 A 对象成功填充 B！

填充完成后，spring 会**删除二级缓存和三级缓存，并填充至一级缓存**，对象成功创建。

缓存的一些问题

为啥要三级，一个缓存不行吗？

- 一个缓存是不可以的，因为都是以 beanName 作为 key，如果只有一个缓存将分不清哪个是完全构造完成的实例，哪个是半完成的实例。

> 当然，如果给 key 多一点信息标识也是可行的。

那为啥要三级，直接两级不好吗？

- 阅读源码发现三级缓存中会构造 bean 的最终版本，也就是说可能会返回 bean 的代理而不是 bean 本身。

那为啥要三级，直接两级存代理不好吗？

- 在填充属性阶段不应该过早的直接执行增强器，否则将违背 Spring 的标准，故除非迫不得已才会提前创建最终版本的 bean。

其他细节

继续我们的源码征程，当填充属性完成后，将执行 initializeBean 方法：

```
try {
    populateBean(beanName, mbd, instanceWrapper);
    exposedObject = initializeBean(beanName, exposedObject, mbd);
}
```

这个方法将执行前置增强器、初始化方法和后置增强器：

```
Object wrappedBean = bean;
if (mbd == null || !mbd.isSynthetic()) {
    wrappedBean =
applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);
}
try {
    invokeInitMethods(beanName, wrappedBean, mbd);
}
catch (Throwable ex) {

}
if (mbd == null || !mbd.isSynthetic()) {
    wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean,
beanName);
}
```

注入 AOP，@AutoWired 都是使用增强器完成的。

这符合我们流程图的步骤。

再接下来 bean 就被注册到 bean 工厂中，可以正常使用了。

AOP 源码

AOP 是增强器 `AbstractAutoProxyCreator` 实现的。

AOP 采用后置增强器：

```
@Override
public Object postProcessAfterInitialization(@Nullable Object bean,
String beanName) {
    if (bean != null) {
        Object cacheKey = getCacheKey(bean.getClass(), beanName);
        if (this.earlyProxyReferences.remove(cacheKey) != bean) {
            return wrapIfNecessary(bean, beanName, cacheKey);
        }
    }
    return bean;
}
```

进入 wrapIfNecessary 方法，然后进入 createProxy 方法：

```
protected Object createProxy(Class<?> beanClass, @Nullable String
beanName,
            @Nullable Object[] specificInterceptors, TargetSource
targetSource) {
    // s
    return proxyFactory.getProxy(getProxyClassLoader());
}
```

跟进源码：

```
public Object getProxy(@Nullable ClassLoader classLoader) {
    return createAopProxy().getProxy(classLoader);
}
```

持续跟进 createAopProxy 源码，最终发现这个方法返回了 DefaultAopProxyFactory 类，跟进 DefaultAopProxyFactory.getProxy 方法：

```
public AopProxy createAopProxy(AdvisedSupport config) throws
AopConfigException {
    if (config.isOptimize() || config.isProxyTargetClass() ||
hasNoUserSuppliedProxyInterfaces(config)) {
        Class<?> targetClass = config.getTargetClass();
        if (targetClass == null) {
            throw new Exception();
        }
        if (targetClass.isInterface() || Proxy.isProxyClass(targetClass))
{
            return new JdkDynamicAopProxy(config);
        }
        return new ObjenesisCglibAopProxy(config);
    }
    else {
        return new JdkDynamicAopProxy(config);
    }
}
```

config 封装了一系列代理信息，例如代理类和被代理类（正式点说就是切点、连接点啥的），targetClass 即时被代理类（代理目标类），**createAopProxy 方法判断代理目标类是否是接口或者是否已经是 JDK Proxy，是的话就采用 JdkDynamicAopProxy，即 JDK 动态代理，否则使用 Cglib 动态代理。**