芋道源码 —— 知识星球

我是一段不羁的公告！
记得给芳芳这 3 个项目加油，添加一个 STAR 噢。
https://github.com/YunaiV/SpringBoot-Labs
https://github.com/YunaiV/onemall
https://github.com/YunaiV/ruoyi-vue-pro
2019-06-13
Spring

【死磕 Spring】—— IoC 之加载 Bean：创建 Bean（五）之循环依赖处理

本文主要基于 Spring 5.0.6.RELEASE

摘要：原创出处 http://cmsblogs.com/?p=todo 「小明哥」，谢谢！

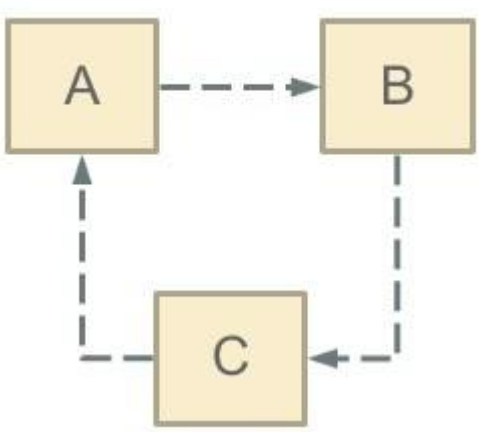
作为「小明哥」的忠实读者，「老芳芳」略作修改，记录在理解过程中，参考的资料。

> 友情提示：本文建议重点阅读，因为 Spring 如何解决循环依赖，是 Spring IOC 相关的面试题中最高频的面试题之一。
>
> 需要搞懂两个点：为什么需要二级缓存？为什么需要三级缓存？

这篇分析 `#doCreateBean(...)` 方法的第三个过程：循环依赖处理。其实，循环依赖并不仅仅只是在 `#doCreateBean(...)` 方法中处理，而是在整个加载 bean 的过程中都有涉及。所以，本文内容并不仅仅只局限于 `#doCreateBean(...)` 方法，而是从整个 Bean 的加载过程进行分析。

1. 什么是循环依赖

循环依赖，其实就是循环引用，就是两个或者两个以上的 bean 互相引用对方，最终形成一个闭环，如 A 依赖 B，B 依赖 C，C 依赖 A。如下图所示：



循环依赖，其实就是一个死循环的过程，在初始化 A 的时候发现引用了 B，这时就会去初始化 B，然后又发现 B 引用 C，跑去初始化 C，初始化 C 的时候发现引用了 A，则又会去初始化 A，依次循环永不退出，除非有终结条件。

Spring 循环依赖的场景有两种：

1. 构造器的循环依赖。
2. field 属性的循环依赖。

对于构造器的循环依赖，Spring 是无法解决的，只能抛出 BeanCurrentlyInCreationException 异常表示循环依赖，所以下面我们分析的都是基于 field 属性的循环依赖。

在博客 《【【死磕 Spring】—— IoC 之开启 Bean 的加载》 中提到，Spring 只解决 scope 为 singleton 的循环依赖。对于scope 为 prototype 的 bean ，Spring 无法解决，直接抛出 BeanCurrentlyInCreationException 异常。

为什么 Spring 不处理 prototype bean 呢？其实如果理解 Spring 是如何解决 singleton bean 的循环依赖就明白了。这里先卖一个关子，我们先来关注 Spring 是如何解决 singleton bean 的循环依赖的。

2. 解决循环依赖

2.1 getSingleton

我们先从加载 bean 最初始的方法 AbstractBeanFactory 的 `#doGetBean(final String name, final Class<T> requiredType, final Object[] args, boolean typeCheckOnly)` 方法开始。

在 `#doGetBean(...)` 方法中，首先会根据 `beanName` 从单例 bean 缓存中获取，如果不为空则直接返回。代码如下：

```
// AbstractBeanFactory.java

Object sharedInstance = getSingleton(beanName);
```

调用 `#getSingleton(String beanName, boolean allowEarlyReference)` 方法，从单例缓存中获取。代码如下：

```
// DefaultSingletonBeanRegistry.java

@Nullable
protected Object getSingleton(String beanName, boolean allowEarlyReference) {
    // 从单例缓冲中加载 bean
    Object singletonObject = this.singletonObjects.get(beanName);
    // 缓存中的 bean 为空，且当前 bean 正在创建
    if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {
        // 加锁
        synchronized (this.singletonObjects) {
            // 从 earlySingletonObjects 获取
            singletonObject = this.earlySingletonObjects.get(beanName);
            // earlySingletonObjects 中没有，且允许提前创建
            if (singletonObject == null && allowEarlyReference) {
                // 从 singletonFactories 中获取对应的 ObjectFactory
```

```
                ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);
                if (singletonFactory != null) {
                    // 获得 bean
                    singletonObject = singletonFactory.getObject();
                    // 添加 bean 到 earlySingletonObjects 中
                    this.earlySingletonObjects.put(beanName, singletonObject);
                    // 从 singletonFactories 中移除对应的 ObjectFactory
                    this.singletonFactories.remove(beanName);
                }
            }
        }
    }
    return singletonObject;
}
```

- 这个方法主要是从三个缓存中获取，分别是：singletonObjects、earlySingletonObjects、singletonFactories 。三者定义如下：

```
// DefaultSingletonBeanRegistry.java

/**
 * Cache of singleton objects: bean name to bean instance.
 *
 * 一级缓存，存放的是单例 bean 的映射。
 *
 * 注意，这里的 bean 是已经创建完成的。
 *
 * 对应关系为 bean name --> bean instance
 */
private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>(256);

/**
 * Cache of early singleton objects: bean name to bean instance.
 *
 * 二级缓存，存放的是早期半成品（未初始化完）的 bean，对应关系也是 bean name --> bean instance。
 *
 * 它与 {@link #singletonObjects} 区别在于， 它自己存放的 bean 不一定是完整。
 *
 * 这个 Map 也是【循环依赖】的关键所在。
 */
private final Map<String, Object> earlySingletonObjects = new HashMap<>(16);

/**
 * Cache of singleton factories: bean name to ObjectFactory.
 *
 * 三级缓存，存放的是 ObjectFactory，可以理解为创建早期半成品（未初始化完）的 bean 的 factory ，最终添加
 *
 * 对应关系是 bean name --> ObjectFactory
 *
 * 这个 Map 也是【循环依赖】的关键所在。
 */
private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap<>(16);
```

  - singletonObjects ： 单例对象的 Cache 。
  - earlySingletonObjects ： 提前曝光的单例对象的 Cache 。
  - singletonFactories ： 单例对象工厂的 Cache 。

它们三，就是 Spring 解决 singleton bean 的关键因素所在，我称他们为三级缓存：

第一级为 singletonObjects
第二级为 earlySingletonObjects
第三级为 singletonFactories

这里，我们已经通过 #getSingleton(String beanName, boolean allowEarlyReference) 方法，看到他们是如何配合的。详细分析该方法之前，提下其中的 #isSingletonCurrentlyInCreation(String beanName) 方法和 allowEarlyReference 变量：

#isSingletonCurrentlyInCreation(String beanName) 方法：判断当前 singleton bean 是否处于创建中。bean 处于创建中，也就是说 bean 在初始化但是没有完成初始化，有一个这样的过程其实和 Spring 解决 bean 循环依赖的理念相辅相成。因为 Spring 解决 singleton bean 的核心就在于提前曝光 bean 。
allowEarlyReference 变量：从字面意思上面理解就是允许提前拿到引用。其实真正的意思是，是否允许从 singletonFactories 缓存中通过 #getObject() 方法，拿到对象。为什么会有这样一个字段呢？原因就在于 singletonFactories 才是 Spring 解决 singleton bean 的诀窍所在，这个我们后续分析。

#getSingleton(String beanName, boolean allowEarlyReference) 方法，整个过程如下：

首先，从一级缓存 singletonObjects 获取。
如果，没有且当前指定的 beanName 正在创建，就再从二级缓存 earlySingletonObjects 中获取。

如果，还是没有获取到且允许 singletonFactories 通过 #getObject() 获取，则从三级缓存 singletonFactories 获取。如果获取到，则通过其 #getObject() 方法，获取对象，并将其加入到二级缓存 earlySingletonObjects 中，并从三级缓存 singletonFactories 删除。代码如下：

```
// DefaultSingletonBeanRegistry.java

singletonObject = singletonFactory.getObject();
this.earlySingletonObjects.put(beanName, singletonObject);
this.singletonFactories.remove(beanName);
```

- 这样，就从三级缓存升级到二级缓存了。
- 所以，二级缓存存在的意义，就是缓存三级缓存中的 ObjectFactory 的 #getObject() 方法的执行结果，提早曝光的单例 Bean 对象。

2.2 addSingletonFactory

上面是从缓存中获取，但是缓存中的数据从哪里添加进来的呢？一直往下跟会发现在 AbstractAutowireCapableBeanFactory 的 #doCreateBean(final String beanName, final RootBeanDefinition mbd, final Object[] args) 方法中，有这么一段代码：

```
// AbstractAutowireCapableBeanFactory.java

boolean earlySingletonExposure = (mbd.isSingleton() // 单例模式
        && this.allowCircularReferences // 运行循环依赖
        && isSingletonCurrentlyInCreation(beanName)); // 当前单例 bean 是否正在被创建
if (earlySingletonExposure) {
    if (logger.isTraceEnabled()) {
```

```
            logger.trace("Eagerly caching bean '" + beanName +
                    "' to allow for resolving potential circular references");
        }
        // 提前将创建的 bean 实例加入到 singletonFactories 中
        // <X> 这里是为了后期避免循环依赖
        addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));
    }
```

当一个 Bean 满足三个条件时，则调用 #addSingletonFactory(...) 方法，将它添加到缓存中。三个条件如下：

- 单例
- 运行提前暴露 bean
- 当前 bean 正在创建中

#addSingletonFactory(String beanName, ObjectFactory<?> singletonFactory) 方法，代码如下：

```
// DefaultSingletonBeanRegistry.java

protected void addSingletonFactory(String beanName, ObjectFactory<?> singletonFactory) {
    Assert.notNull(singletonFactory, "Singleton factory must not be null");
 synchronized (this.singletonObjects) {
     if (!this.singletonObjects.containsKey(beanName)) {
         this.singletonFactories.put(beanName, singletonFactory);
         this.earlySingletonObjects.remove(beanName);
         this.registeredSingletons.add(beanName);
        }
    }
}
```

- 从这段代码我们可以看出，singletonFactories 这个三级缓存才是解决 Spring Bean 循环依赖的诀窍所在。同时这段代码发生在 #createBeanInstance(...) 方法之后，也就是说这个 bean 其实已经被创建出来了，但是它还不是很完美（没有进行属性填充和初始化），但是对于其他依赖它的对象而言已经足够了（可以根据对象引用定位到堆中对象），能够被认出来了。所以 Spring 在这个时候，选择将该对象提前曝光出来让大家认识认识。

另外，<X> 处的 #getEarlyBeanReference(String beanName, RootBeanDefinition mbd, Object bean) 方法也非常重要，这里会创建早期初始化 Bean 可能存在的 AOP 代理等等。代码如下：

```
// AbstractAutowireCapableBeanFactory.java

/**
 * 对创建的早期半成品（未初始化）的 Bean 处理引用
 *
 * 例如说，AOP 就是在这里动态织入，创建其代理 Bean 返回
 *
 * Obtain a reference for early access to the specified bean,
 * typically for the purpose of resolving a circular reference.
 * @param beanName the name of the bean (for error handling purposes)
 * @param mbd the merged bean definition for the bean
 * @param bean the raw bean instance
 * @return the object to expose as bean reference
 */
protected Object getEarlyBeanReference(String beanName, RootBeanDefinition mbd, Object bean) {
    Object exposedObject = bean;
```

```
 if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
     for (BeanPostProcessor bp : getBeanPostProcessors()) {
         if (bp instanceof SmartInstantiationAwareBeanPostProcessor) {
                SmartInstantiationAwareBeanPostProcessor ibp = (SmartInstantiationAwareBeanPostProcessor) bp;
                exposedObject = ibp.getEarlyBeanReference(exposedObject, beanName);
            }
        }
    }
 return exposedObject;
}
```

这也是为什么 Spring 需要额外增加 `singletonFactories` 三级缓存的原因，解决 Spring 循环依赖情况下的 Bean 存在动态代理等情况，不然循环注入到别人的 Bean 就是原始的，而不是经过动态代理的！
另外，这里在推荐一篇[《Spring循环依赖三级缓存是否可以减少为二级缓存？》]文章，解释的也非常不错。

2.3 addSingleton

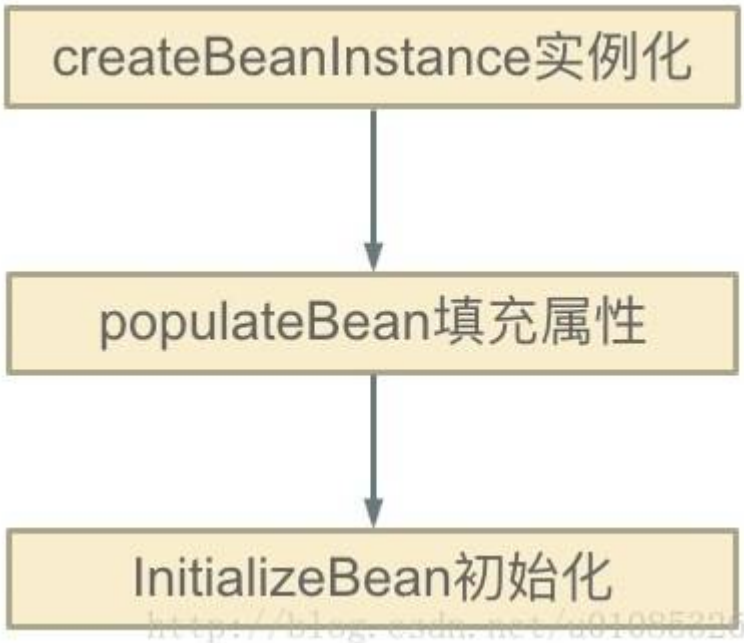
介绍到这里我们发现三级缓存 `singletonFactories` 和 二级缓存 `earlySingletonObjects` 中的值都有出处了，那一级缓存在哪里设置的呢？在类 DefaultSingletonBeanRegistry 中，可以发现这个 `#addSingleton(String beanName, Object singletonObject)` 方法，代码如下：

```
// DefaultSingletonBeanRegistry.java

protected void addSingleton(String beanName, Object singletonObject) {
 synchronized (this.singletonObjects) {
     this.singletonObjects.put(beanName, singletonObject);
     this.singletonFactories.remove(beanName);
     this.earlySingletonObjects.remove(beanName);
     this.registeredSingletons.add(beanName);
    }
}
```

添加至一级缓存，同时从二级、三级缓存中删除。

这个方法在我们创建 bean 的链路中有哪个地方引用呢？其实在前面博客 LZ 已经提到过了，在 `#doGetBean(...)` 方法中，处理不同 scope 时，如果是 singleton，则调用 `#getSingleton(...)` 方法，如下图所示：



前面几篇博客已经分析了 `#createBean(...)` 方法，这里就不再阐述了，我们关注 `#getSingleton(String beanName, ObjectFactory<?> singletonFactory)` 方法，代码如下：

```
// AbstractBeanFactory.java

public Object getSingleton(String beanName, ObjectFactory<?> singletonFactory) {
    Assert.notNull(beanName, "Bean name must not be null");
    synchronized (this.singletonObjects) {
        Object singletonObject = this.singletonObjects.get(beanName);
        if (singletonObject == null) {
            //....
            try {
                singletonObject = singletonFactory.getObject();
                newSingleton = true;
            }
            //.....
            if (newSingleton) {
                addSingleton(beanName, singletonObject);
            }
        }
        return singletonObject;
    }
}
```

- 注意，此处的 `#getSingleton(String beanName, ObjectFactory<?> singletonFactory)` 方法，在 AbstractBeanFactory 类中实现，和 「2.1 getSingleton」 不同。

3. 小结

至此，Spring 关于 singleton bean 循环依赖已经分析完毕了。所以我们基本上可以确定 Spring 解决循环依赖的方案了：

> Spring 在创建 bean 的时候并不是等它完全完成，而是在创建过程中将创建中的 bean 的 ObjectFactory 提前曝光（即加入到 `singletonFactories` 缓存中）。

这样，一旦下一个 bean 创建的时候需要依赖 bean ，则直接使用 ObjectFactory 的 #getObject() 方法来获取了，也就是 「2.1 getSingleton」 小结中的方法中的代码片段了。

到这里，关于 Spring 解决 bean 循环依赖就已经分析完毕了。最后来描述下就上面那个循环依赖 Spring 解决的过程：

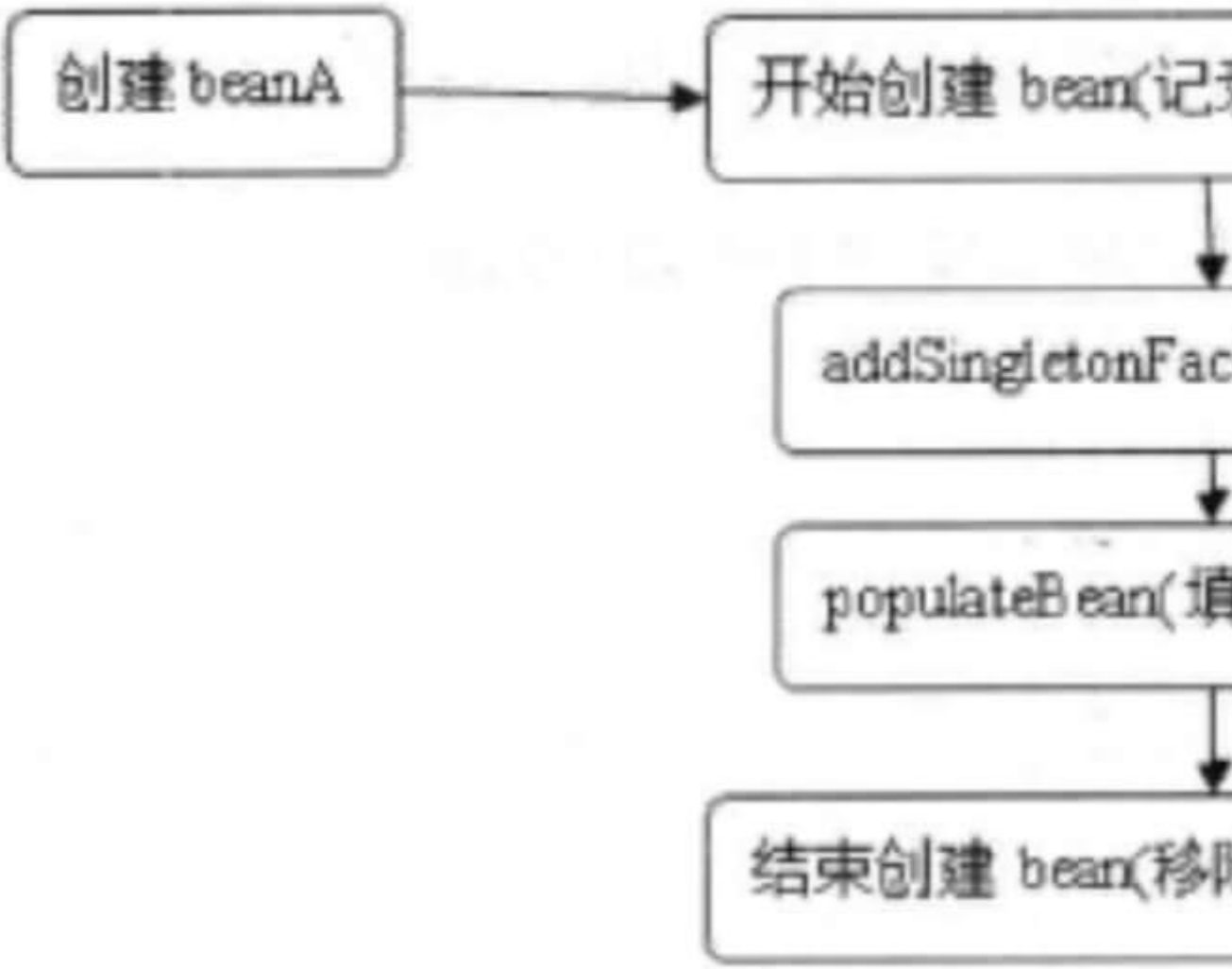
首先 A 完成初始化第一步并将自己提前曝光出来（通过 ObjectFactory 将自己提前曝光），在初始化的时候，发现自己依赖对象 B，此时就会去尝试 get(B)，这个时候发现 B 还没有被创建出来
然后 B 就走创建流程，在 B 初始化的时候，同样发现自己依赖 C，C 也没有被创建出来
这个时候 C 又开始初始化进程，但是在初始化的过程中发现自己依赖 A，于是尝试 get(A)，这个时候由于 A 已经添加至缓存中（一般都是添加至三级缓存 singletonFactories），通过 ObjectFactory 提前曝光，所以可以通过 ObjectFactory#getObject() 方法来拿到 A 对象，C 拿到 A 对象后顺利完成初始化，然后将自己添加到一级缓存中
回到 B ，B 也可以拿到 C 对象，完成初始化，A 可以顺利拿到 B 完成初始化。到这里整个链路就已经完成了初始化过程了

老艿艿的建议

可能逻辑干看比较绕，胖友可以拿出一个草稿纸，画一画上面提到的 A、B、C 初始化的过程。

相信，胖友会很快明白了。

如下是《Spring 源码深度解析》P114 页的一张图，非常有助于理解。



文章目录

回到首页