【死磕 Spring】—— IOC 之循环依赖处理 (http://cmsblogs.com/?p=2887)

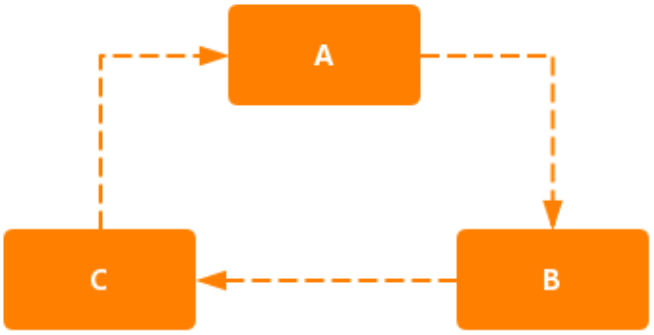
2018-11-05 分类：死磕 Spring (http://cmsblogs.com/?cat=206) 阅读(12283) 评论(0)

> 原文出自：http://cmsblogs.com (http://cmsblogs.com)

这篇分析 `doCreateBean()` 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()` 中处理，其实在整个加载 bean 的过程中都有涉及，所以下篇内容并不仅仅只局限于 `doCreateBean()`，而是从整个 Bean 的加载过程进行分析。

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：



(https://gitee.com/chenssy/blog-home/raw/master/image/201811/201808131001.png) 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 (http://cmsblogs.com/?p=2806) 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，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。

解决循环依赖

我们先从加载 bean 最初始的方法 doGetBean() 开始。在 doGetBean() 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。

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

调用 getSingleton() 方法从单例缓存中获取，如下：

```
protected Object getSingleton(String beanName, boolean allowEarlyReference) {
    Object singletonObject = this.singletonObjects.get(beanName);
    if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {
        synchronized (this.singletonObjects) {
            singletonObject = this.earlySingletonObjects.get(beanName);
            if (singletonObject == null && allowEarlyReference) {
                ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);
                if (singletonFactory != null) {
                    singletonObject = singletonFactory.getObject();
                    this.earlySingletonObjects.put(beanName, singletonObject);
                    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：

```
/** Cache of singleton objects: bean name --> bean instance */
private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>(256);

/** Cache of singleton factories: bean name --> ObjectFactory */
private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap<>(16);

/** Cache of early singleton objects: bean name --> bean instance */
private final Map<String, Object> earlySingletonObjects = new HashMap<>(16);
```

意义如下：

- singletonObjects：单例对象的cache
- singletonFactories： 单例对象工厂的cache
- earlySingletonObjects： 提前暴光的单例对象的Cache

他们就是 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。这里我们可以通过 getSingleton() 看到他们是如何配合的，这分析该方法之前，提下其中的 isSingletonCurrentlyInCreation() 和 allowEarlyReference 。

- `isSingletonCurrentlyInCreation()`：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：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()` 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()` 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，就再从二级缓存中 earlySingletonObjects 获取，如果还是没有获取到且运行 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()` 获取对象，并将其加入到二级缓存 earlySingletonObjects 中，从三级缓存 singletonFactories 删除，如下：

```
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。 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 `doCreateBean()`（AbstractAutowireCapableBeanFactory）中，有这么一段代码：

```
boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences && isSingletonCurrent
lyInCreation(beanName));
if (earlySingletonExposure) {
    if (logger.isDebugEnabled()) {
        logger.debug("Eagerly caching bean '" + beanName +
                     "' to allow for resolving potential circular references");
    }
    addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));
}
```

如果 `earlySingletonExposure == true` 的话，则调用 `addSingletonFactory()` 将他们添加到缓存中，但是一个 bean 要具备如下条件才会添加至缓存中：

- 单例
- 运行提前暴露 bean
- 当前 bean 正在创建中

`addSingletonFactory()` 代码如下：

```
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。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()` 方法，源码如下：

```
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()` 处理不同 scope 时，如果是 singleton，则调用 `getSingleton()`，如下：

```
// Create bean instance.
if (mbd.isSingleton()) {
    sharedInstance = getSingleton(beanName, () -> {
        try {
            return createBean(beanName, mbd, args);
        }
        catch (BeansException ex) {
            // Explicitly remove instance from singleton cache: It might have been put there
            // eagerly by the creation process, to allow for circular reference resolution.
            // Also remove any beans that received a temporary reference to the bean.
            destroySingleton(beanName);
            throw ex;
        }
    });
    bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
}
```

(https://gitee.com/chenssy/blog-home/raw/master/image/201811/15341402420152.jpg) 前面几篇博客已经分析了 `createBean()`，这里就不再阐述了，我们关注方法 `getSingleton()` 代码如下：

```
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;
    }
}
```

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()` 获取了，也就是 `getSingleton()` 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()` 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。 **更多阅读**

- 【死磕 Spring】—— IOC 之开启 bean 的实例化进程 (http://cmsblogs.com/?p=2846)
- 【死磕 Spring】—— IOC 之 Factory 实例化 bean (http://cmsblogs.com/?p=2848)
- 【死磕 Spring】—— IOC 之构造函数实例化 bean (http://cmsblogs.com/?p=2850)
- 【死磕 Spring】----- IOC 之 属性填充 (http://cmsblogs.com/?p=2885)

标签： Spring源码解析 (http://cmsblogs.com/?tag=spring%e6%ba%90%e7%a0%81%e8%a7%a3%e6%9e%90)

死磕Java (http://cmsblogs.com/?tag=%e6%ad%bb%e7%a3%95java)

死磕Spring (http://cmsblogs.com/?tag=%e6%ad%bb%e7%a3%95spring)

循环引用 b站视频
https://www.bilibili.com/video/BV1ET4y1N7Sp?p=2