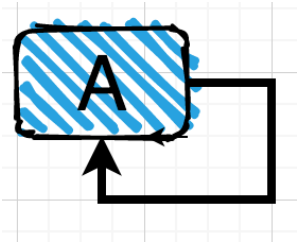
Spring 解决循环依赖的流程分析

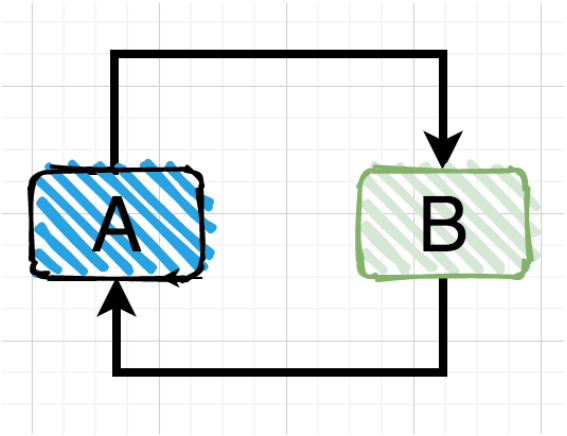
1.什么是循环依赖?

循环依赖：一个或多个对象实例之间存在直接或间接的依赖关系，这种依赖关系构成了构成一个环形调用。

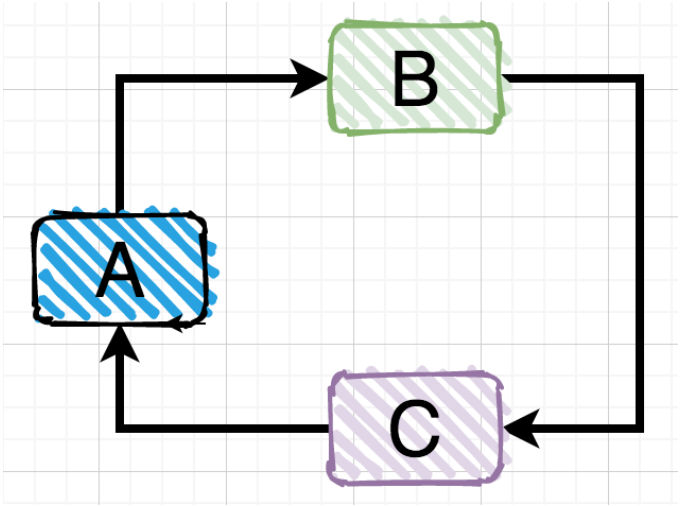
- 第一种情况：自己依赖自己的直接依赖



- 第二种情况：两个对象之间的直接依赖



- 第三种情况：多个对象之间的间接依赖



2. Spring 创建 Bean 主要流程

2.1. 实例化 Bean

```
//
org/springframework/beans/factory/support/AbstractAutowireCapableBeanFactory.java#doCreate
Bean
instanceWrapper = createBeanInstance(beanName, mbd, args);
```

主要是通过反射调用默认构造函数创建 Bean 实例，此时 Bean 的属性都还是默认值 null。被注解 @Bean 标记的方法就是此阶段被调用的。

2.2. 填充 Bean 属性

```
//
org/springframework/beans/factory/support/AbstractAutowireCapableBeanFactory.java#doCreate
Bean
populateBean(beanName, mbd, instanceWrapper);
```

这一步主要是对 Bean 的依赖属性进行填充，对 @Value、@Autowired、@Resource 注解标注的属性注入对象引用。

2.3. 调用 Bean 初始化方法

```
//
org/springframework/beans/factory/support/AbstractAutowireCapableBeanFactory.java#doCreate
Bean
exposedObject = initializeBean(beanName, exposedObject, mbd);
```

调用配置指定中的 init 方法，例如，如果 xml 文件指定 Bean 的 init-method 方法或注解 `@Bean(initMethod = "initMethod")` 指定的方法。

3. BeanPostProcessor 接口拓展点

在 Bean 创建的流程中 Spring 提供了多个 BeanPostProcessor 接口方便开发者对 Bean 进行自定义调整和加工。

有以下几种 BeanPostProcessor 接口比较常用：

- `postProcessMergedBeanDefinition`：可对 BeanDefinition 添加额外的自定义配置；
- `getEarlyBeanReference`：返回早期暴露的 Bean 引用，一个典型的例子是循环依赖时如果有动态代理，需要在此先返回代理实例；
- `postProcessAfterInstantiation`：在 populateBean 前用户可以手动注入一些属性；
- `postProcessProperties`：对属性进行注入，例如配置文件加密信息在此解密后注入；
- `postProcessBeforeInitialization`：属性注入后的一些额外操作；
- `postProcessAfterInitialization`：实例完成创建的最后一步，这里也是一些 BPP 进行 AOP 代理的时机。

4. 三级缓存

```
// org/springframework/beans/factory/support/DefaultSingletonBeanRegistry.java
public class DefaultSingletonBeanRegistry extends SimpleAliasRegistry implements
SingletonBeanRegistry {
    // ...
    /** Cache of singleton objects: bean name to bean instance. */
    private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>(256);

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

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

三级缓存：

- **singletonObjects**：一级缓存

  主要存放的是已经完成实例化、属性填充和初始化所有步骤的单例 Bean 实例，这样的 Bean 能够直接提供给用户使用，我们称之为终态 Bean 或叫成熟 Bean。

- **earlySingletonObjects**：二级缓存

  主要存放的已经完成初始化，但属性还没自动赋值的 Bean，这些 Bean 还不能提供用户使用，只是用于提前暴露的 Bean 实例，我们把这样的 Bean 称之为临时 Bean 或早期的 Bean（半成品 Bean）。

- **singletonFactories**：三级缓存

  存放的是 ObjectFactory 的匿名内部类实例，调用 ObjectFactory.getObject() 最终会调用 getEarlyBeanReference 方法，该方法可以获取提前暴露的单例 Bean 引用。

5. Spring解决循环依赖流程分析

Spring 通过三级缓存解决了循环依赖，其中，一级缓存为单例池（singletonObjects），二级缓存为早期曝光对象 earlySingletonObjects，三级缓存为早期曝光对象工厂（singletonFactories）。

当 A、B 两个类发生循环引用时，在 A 完成实例化后，就使用实例化后的对象去创建一个对象工厂，并添加到三级缓存中，如果 A 被 AOP 代理，那么，通过这个工厂获取到的就是 A 代理后的对象，如果 A 没有被 AOP 代理，那么，这个工厂获取到的就是 A 实例化的对象。

当 A 进行属性注入时，会去创建 B，同时，B 又依赖了 A，所以，创建 B 的同时又会去调用 getBean(a) 来获取需要的依赖，此时的 getBean(a) 会从缓存中获取，第一步，先获取到三级缓存中的工厂；第二步，调用对象工工厂的getObject方法来获取到对应的对象，得到这个对象后将其注入到 B 中。紧接着 B 会走完它的生命周期流程，包括初始化、后置处理器等。当 B 创建完后，会将 B 再注入到 A 中，此时， A 再完成它的整个生命周期。

通过上述流程，可以看出 bean 都是需要先可以被实例化才可以的，所以，这也就是为什么构造器依赖可能会失败的原因。

例如，Bean A 的构造器依赖 B，而实例化 A 需要先调用 A 的构造函数，发现依赖 B，那么，需要再去初始化 B，但是，B 也依赖 A，不管 B 是通过构造器注入还是 setter 注入，此时，由于 A 没有被实例化，没有放入三级缓存，所以， B 无法被初始化，所以，spring 会直接报错。反之，如果 A 通过 setter 注入的话，那么，则可以通过构造函数先实例化，放入缓存，然后再填充属性，这样的话不管 B 是通过 setter 还是构造器注入 A，都能在缓存中获取到，于是可以初始化。

6.Spring 如何解决循环依赖

在 Spring 中，只有同时满足以下两点才能解决循环依赖的问题：

- 依赖的 Bean 必须都是单例
- 依赖注入的方式，必须不全是构造器注入，且 beanName 字母序在前的不能是构造器注入

  spring 的 bean 加载顺序：默认情况下，是按照文件完整路径递归查找的，按路径 + 文件名排序，排在前面的先加载。

6.1 为什么必须是单例

如果循环依赖的 Bean 是原型模式，会直接抛错，其源码如下：

```
// org/springframework/beans/factory/support/AbstractBeanFactory.java
protected <T> T doGetBean(String name, @Nullable Class<T> requiredType, @Nullable Object[]
args, boolean typeCheckOnly) throws BeansException {
    // ...

    // Fail if we're already creating this bean instance:
    // We're assumably within a circular reference.
    if (isPrototypeCurrentlyInCreation(beanName)) {
        throw new BeanCurrentlyInCreationException(beanName);
    }

    // ...
}
```

6.2 为什么无法支持原型对象

因为原型模式都需要创建新的对象，不能用以前的对象。

如果 Bean A 和 Bean B 都是原型模式的话，那么，

- 创建 A1 需要创建一个 B1；
- 创建 B1 的时候要创建一个 A2；
- 创建 A2 又要创建一个 B2；
- 创建 B2 又要创建一个 A3；
- 创建 A3 又要创建一个 B3
- ...

就会陷入死循环。

如果是单例的话，创建 A 需要创建 B，而创建的 B 需要的还是之前的个 A。

6.3 为什么不能全是构造器注入

在 Spring 中创建 Bean 分三步:

- 实例化：createBeanInstance，就是 new 了一个对象
- 属性注入：populateBean， 就是 set 了一些属性值
- 初始化：initializeBean，执行一些 aware 接口中的方法，initMethod、AOP 代理等

如果全是构造器注入，例如，A的构造器 A(B b)，那就表明在 new 的时候，就需要得到 B，此时需要 new B(A a)。

但是，B 也是要在构造的时候注入 A ，即 B(A a)，这时候， B 需要在一个 map 中找到不完整的 A ，就会发现找不到，从而导致 Bean 创建失败。

6.4 为什么循环依赖需要三级缓存，二级不够吗

> 思考：如果在实例化 Bean A 之后，在二级 map 里面保存这个 A，然后继续属性注入。发现 A 依赖 B，所以要创建 Bean B，这时候， B 就能从二级缓存得到 A ，完成 B 的建立之后， Bean A 似乎也能完成实例化。

很明显，如果仅仅只是为了解决循环依赖，二级缓存够了，根本就不必要三级缓存。但是，如果我们希望对添加到三级缓存中的实例对象进行增强（例如，AOP），直接用实例对象是行不通的。

但是我们都知道对象如果有代理的话，那么，我们希望直接拿到的是代理对象。

也就是说如果 A 需要被代理，那么， B 依赖的 A 是已经被代理的 A，所以，我们不能返回 A 给 B，而是返回 A 的代理对象给 B。

其核心的代码实现如下：

```
// org/springframework/beans/factory/support/AbstractBeanFactory.java
protected <T> T doGetBean(String name, @Nullable Class<T> requiredType, @Nullable Object[]
args, boolean typeCheckOnly) throws BeansException {
    // ...

    // Eagerly cache singletons to be able to resolve circular references
    // even when triggered by lifecycle interfaces like BeanFactoryAware.
    boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences &&
            isSingletonCurrentlyInCreation(beanName));
    if (earlySingletonExposure) {
        addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));
    }

    // ...
}

protected Object getEarlyBeanReference(String beanName, RootBeanDefinition mbd, Object
bean) {
    Object exposedObject = bean;
    // 判断是否有后置处理器
    if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
        // Bean需要被AOP代理
        for (SmartInstantiationAwareBeanPostProcessor bp :
getBeanPostProcessorCache().smartInstantiationAware) {
            exposedObject = bp.getEarlyBeanReference(exposedObject, beanName);
        }
    }
    return exposedObject;
}

protected <T> T doGetBean(String name, @Nullable Class<T> requiredType, @Nullable Object[]
args, boolean typeCheckOnly) throws BeansException {
```

```
    // ...
    if (mbd.isSingleton()) {
        sharedInstance = getSingleton(beanName, () -> {
            try {
                return createBean(beanName, mbd, args); // 创建单例Bean
            }
            catch (BeansException ex) {
                destroySingleton(beanName);
                throw ex;
            }
        });
        beanInstance = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
    }
    // ...
}
```

三级工厂的作用就是判断这个对象是否需要代理，如果否则直接返回，如果是则返回代理对象。

6.5 为什么代理对象没有放在二级缓存中

通常代理对象的生成是基于后置处理器，是在被代理的对象初始化后期调用生成的，所以，如果我们提早代理了其实是违背了 Bean 定义的生命周期。所以， Spring 先在一个三级缓存放置一个工厂，如果产生循环依赖，那么，就调用这个工厂提早得到代理对象。

如果没产生依赖，这个工厂根本不会被调用，所以，Bean 的生命周期就是对的。