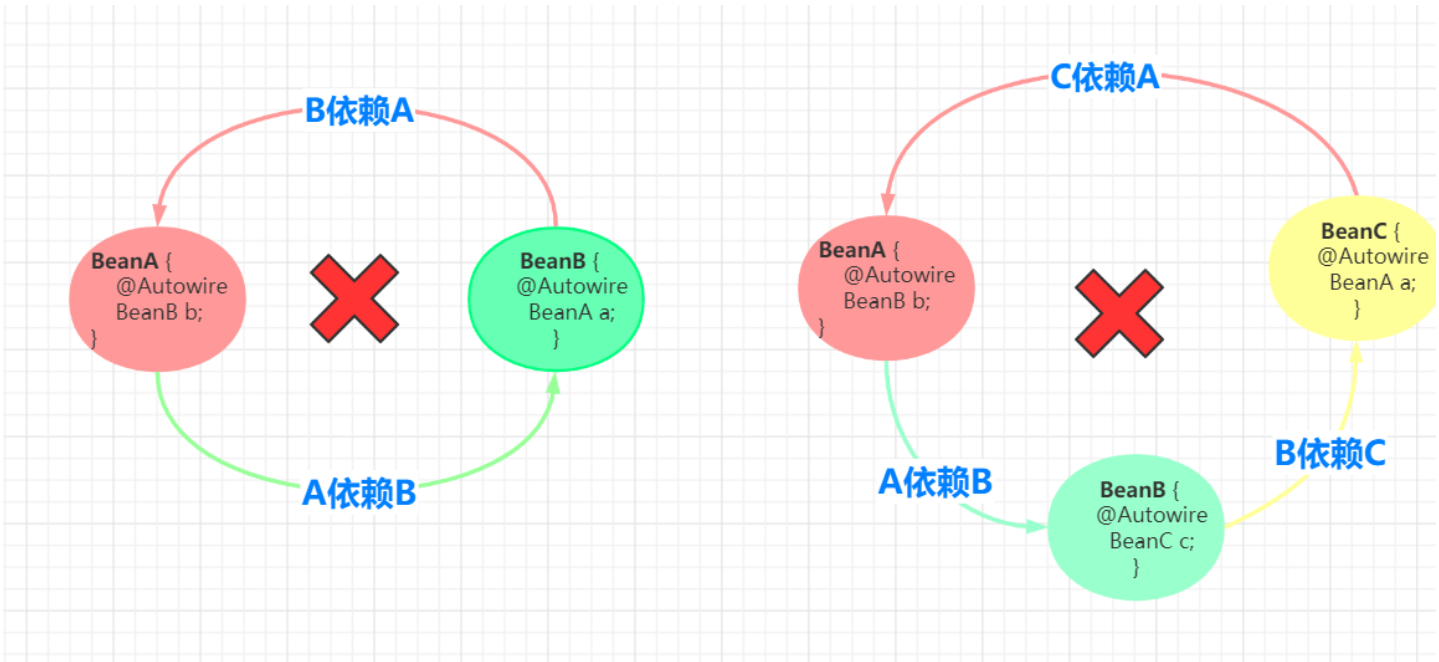
Spring 是如何解决循环依赖的

1.什么是循环依赖?

所谓的循环依赖是指，A 依赖 B，B 又依赖 A，它们之间形成了循环依赖。或者是 A 依赖 B，B 依赖 C，C 又依赖 A。它们之间的依赖关系如下：



2.手写Spring循环依赖:

DEMO:

```
1
2 /***
3  * @Author 徐庶   QQ:1092002729
4  * @Slogan 致敬大师，致敬未来的你
5  *
6  * Spring --循环依赖实例DEMO  :
7  * 帮助您更有效的理解Spring循环依赖源码
```

```
 */
public class MainStart {

    private static Map<String, BeanDefinition> beanDefinitionMap = new ConcurrentHashMap<>(256);

    /**
     * 读取bean定义，当然在spring中肯定是根据配置 动态扫描注册
     */
    public static void loadBeanDefinitions() {
        RootBeanDefinition aBeanDefinition=new RootBeanDefinition(InstanceA.class);
        RootBeanDefinition bBeanDefinition=new RootBeanDefinition(InstanceB.class);
        beanDefinitionMap.put("instanceA",aBeanDefinition);
        beanDefinitionMap.put("instanceB",bBeanDefinition);
    }

    public static void main(String[] args) throws Exception {
        // 加载了BeanDefinition
        loadBeanDefinitions();
        // 注册Bean的后置处理器

        // 循环创建Bean
        for (String key : beanDefinitionMap.keySet()){
            // 先创建A
            getBean(key);
        }
        InstanceA instanceA = (InstanceA) getBean("instanceA");
        instanceA.say();
    }

    // 一级缓存
    public static Map<String,Object> singletonObjects=new ConcurrentHashMap<>();


    // 二级缓存:  为了将 成熟Bean和纯净Bean分离，避免读取到不完整得Bean
    public static Map<String,Object> earlySingletonObjects=new ConcurrentHashMap<>();

    // 三级缓存
    public static Map<String,ObjectFactory> singletonFactories=new ConcurrentHashMap<>();

    // 循环依赖标识
    public  static  Set<String> singletonsCurrennlyInCreation=new HashSet<>();


    // 假设A 使用了Aop @PointCut("execution(* *..InstanceA.*(..))")   要给A创建动态代理
    // 获取Bean
    public  static Object getBean(String beanName) throws Exception {
        Object singleton = getSingleton(beanName);
        if(singleton!=null){
            return singleton;
        }

        // 正在创建
        if(!singletonsCurrennlyInCreation.contains(beanName)){
            singletonsCurrennlyInCreation.add(beanName);
        }
        // createBean


```

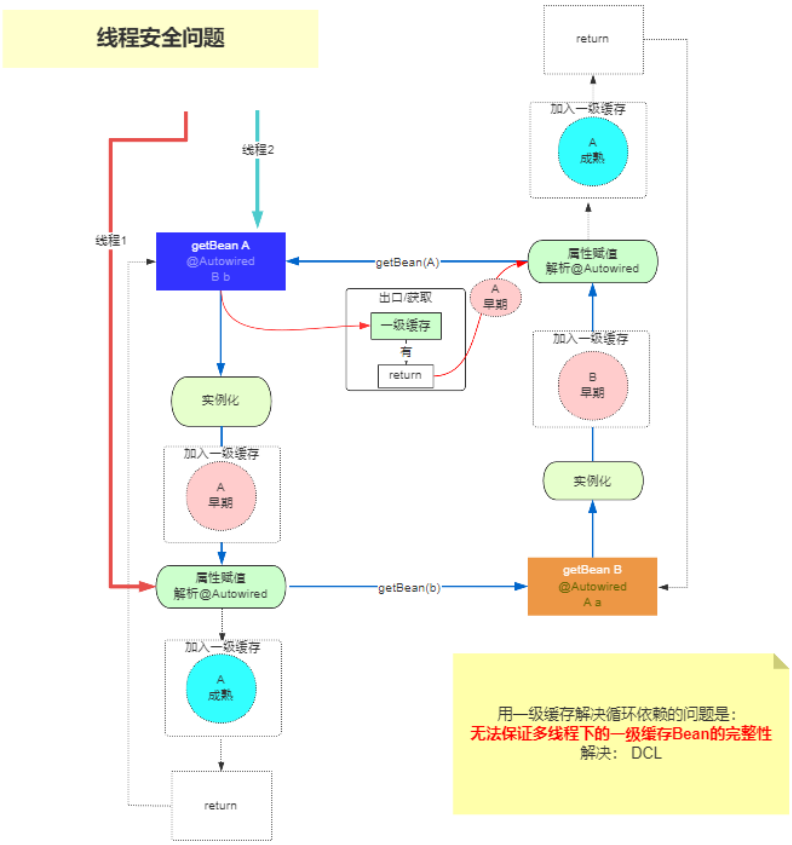
```
        // 实例化
        RootBeanDefinition beanDefinition = (RootBeanDefinition) beanDefinitionMap.get(beanName);
        Class<?> beanClass = beanDefinition.getBeanClass();
        Object instanceBean = beanClass.newInstance();  // 通过无参构造函数

        // 创建动态代理  （耦合 、BeanPostProcessor）    Spring还是希望正常的Bean 还是再初始化后创建
        // 只在循环依赖的情况下在实例化后创建proxy    判断当前是不是循环依赖
        singletonFactories.put(beanName, () -> new JdkProxyBeanPostProcessor().getEarlyBeanReference(earlySingletonObjects.get(beanName

        // 添加到二级缓存
        // earlySingletonObjects.put(beanName,instanceBean);

        // 属性赋值
        Field[] declaredFields = beanClass.getDeclaredFields();
        for (Field declaredField : declaredFields) {
            Autowired annotation = declaredField.getAnnotation(Autowired.class);
            // 说明属性上面有Autowired
            if(annotation!=null){
                 declaredField.setAccessible(true);
                 // byname  bytype  byconstrator
                // instanceB
                String name = declaredField.getName();
                Object fileObject= getBean(name);   //拿到B得Bean
                declaredField.set(instanceBean,fileObject);
            }

        }



        // 初始化   init-mthod
        // 放在这里创建已经完了  B里面的A 不是proxy
        // 正常情况下会再 初始化之后创建proxy




        // 由于递归完后A 还是原实例，，  所以要从二级缓存中拿到proxy 。
        if(earlySingletonObjects.containsKey(beanName)){
            instanceBean=earlySingletonObjects.get(beanName);
        }

        // 添加到一级缓存   A
        singletonObjects.put(beanName,instanceBean);


        // remove 二级缓存和三级缓存
        return instanceBean;
    }


    public  static Object getSingleton(String beanName){
        // 先从一级缓存中拿
        Object bean = singletonObjects.get(beanName);

        // 说明是循环依赖
        if(bean==null && singletonsCurrennlyInCreation.contains(beanName)){
            bean=earlySingletonObjects.get(beanName);
            // 如果二级缓存没有就从三级缓存中拿
            if(bean==null) {
```

```
                // 从三级缓存中拿
                ObjectFactory factory = singletonFactories.get(beanName);
                if (factory != null) {
                    bean=factory.getObject(); // 拿到动态代理
                    earlySingletonObjects.put(beanName, bean);
                }
            }


        }

        return bean;

    }

}
```

> 下面观点完全由徐庶老师看完源码个人观点总结，如有雷同纯属英雄所见略同， 如有误解欢迎讨论！

为什么需要二级缓存?

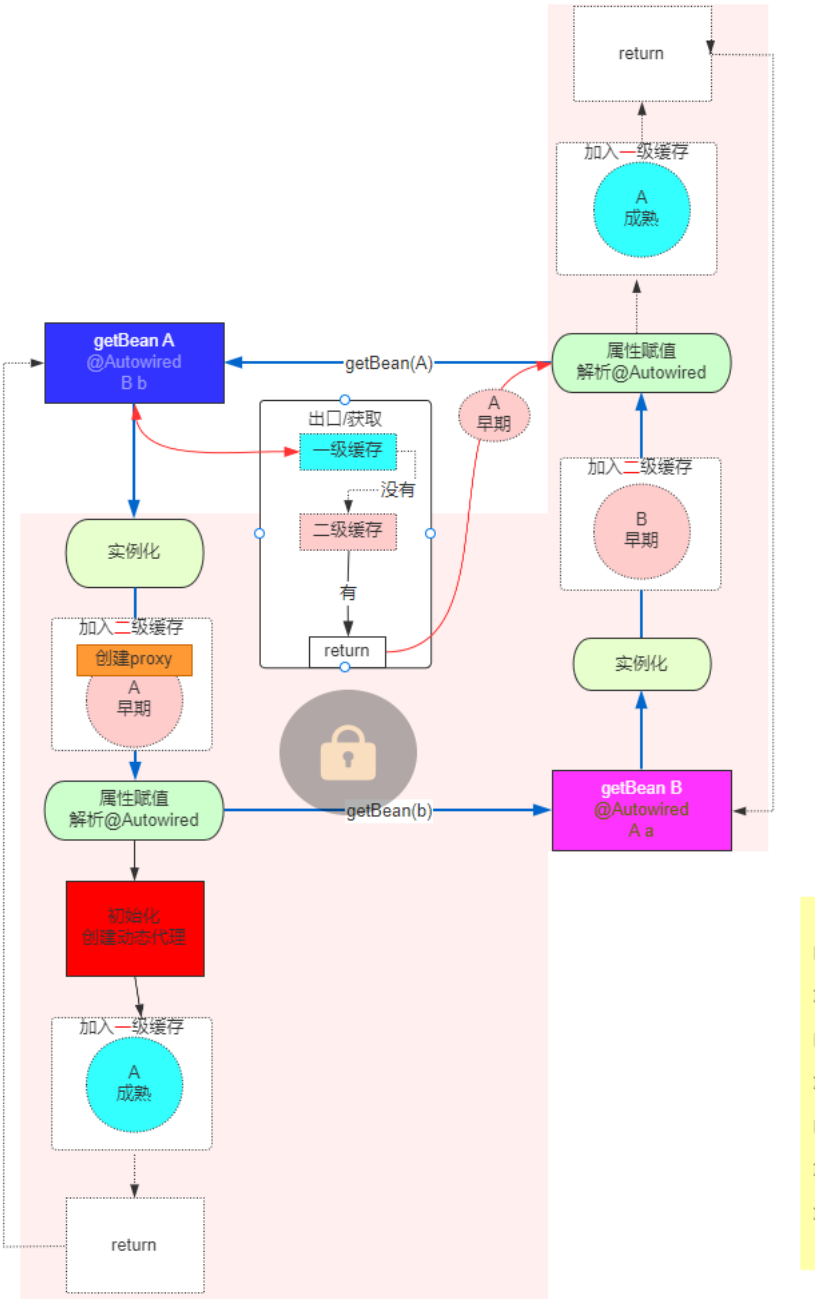
- 作用：存储创建bean时不完整的早期bean；
  - 和完整bean分离存储，保证并发线程安全。提升性能；
- 解决问题：
  - 如果只有一级缓存：多线程getBean获取不完整的bean：



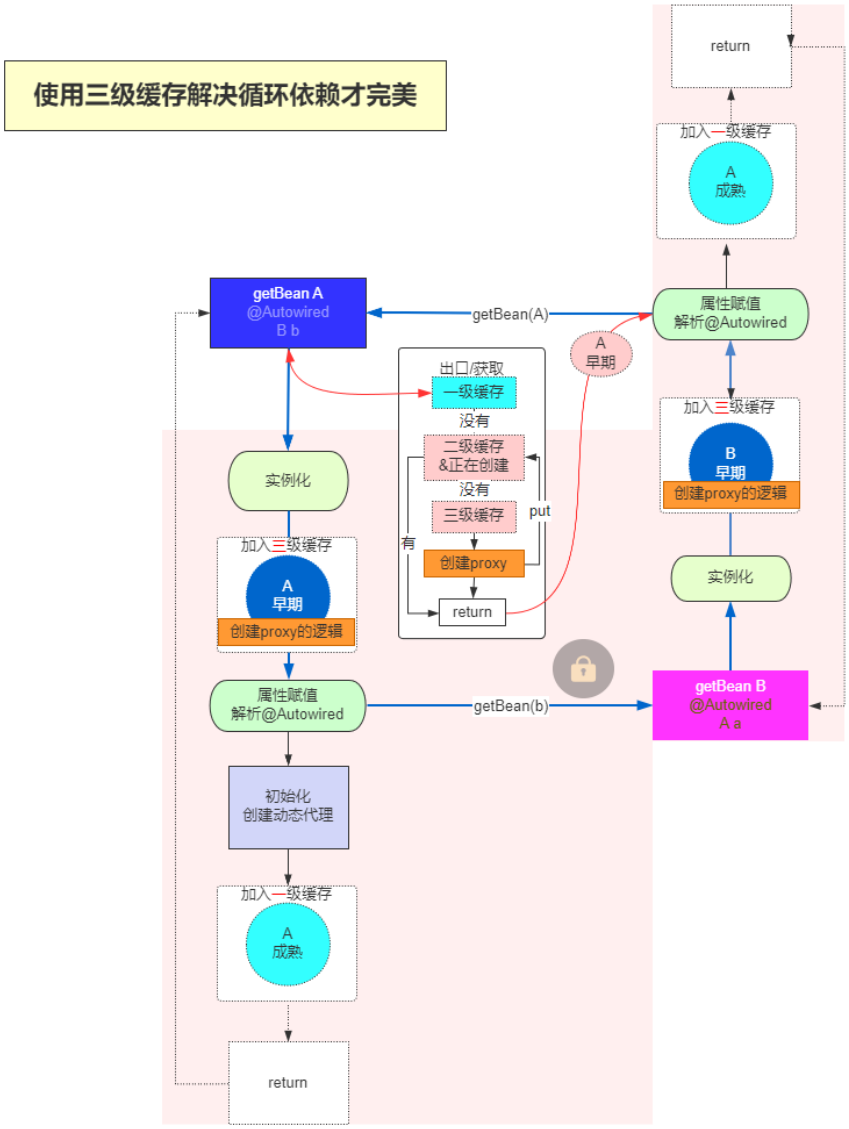
  - 要解决这个问题就要锁住整个getBean方法， 锁粒度太大。已经创建好的bean也需要等待。
  - 通过加入二级缓存可以降低锁的粒度，**提升性能**，将bean进行分离责任分明。

为什么需要三级缓存?

- 作用：解决了循环依赖的死循环。
  - 存储函数接口，提升bean创建过程扩展性，保证规范，代码职责单一性，。
- 解决问题：



- 如果只有二级缓存：aop动态代理需要放在实例化后创建
  - 破坏了bean的生命周期的规范性
  - 循环依赖多次创建多次动态代理?
    - 需要只在循环依赖才创建AOP，普通bean依然初始化创建 以保证规范性。
    - 通过函数接口提升bean在创建过程的扩展性（现在虽然只看到解决循环依赖，在spring6.2并发创建bean有奇效）
    - 通过函数接口保证代码职责单一性， 以为循环依赖只能在getSingleton才能判断， 而beanPostProcessor 是在bean生命周期中调用。



为什么Spring不能解决构造器的循环依赖?

从流程图应该不难看出来，在Bean调用构造器实例化之前，一二三级缓存并没有Bean的任何相关信息，在实例化之后才放入三级缓存中，因此当getBean的时候缓存并没有命中，这样就抛出了循环依赖的异常了。

为什么多例Bean不能解决循环依赖?

我们自己手写了解决循环依赖的代码，可以看到，核心是利用一个map，来解决这个问题的，这个map就相当于缓存。

为什么可以这么做，因为我们的bean是单例的，而且是字段注入（setter注入）的，单例意味着只需要创建一次对象，后面就可以从缓存中取出来，字段注入，意味着我们无需调用构造方法进行注入。

- 如果是原型bean，那么就意味着每次都要去创建对象，无法利用缓存;
- 如果是构造方法注入，那么就意味着需要调用构造方法注入，也无法利用缓存。

3 看源码

2.2 哪三级缓存?

```
DefaultSingletonBeanRegistry类的三个成员变量命名如下：
/** 一级缓存 这个就是我们大名鼎鼎的单例缓存池 用于保存我们所有的单实例bean */
private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>(256);

/** 三级缓存 该map用户缓存 key为 beanName  value 为ObjectFactory(包装为早期对象) */
private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap<>(16);

/** 二级缓存 ，用户缓存我们的key为beanName value是我们的早期对象(对象属性还没有来得及进行赋值) */
private final Map<String, Object> earlySingletonObjects = new HashMap<>(16);
```

以 BeanA 和 BeanB 两个类相互依赖为例

2.1. 创建原始 bean 对象

也就是老师所说的纯洁态Bean

```
instanceWrapper = createBeanInstance(beanName, mbd, args);
final Object bean = (instanceWrapper != null ? instanceWrapper.getWrappedInstance() : null);
```

假设 beanA 先被创建，创建后的原始对象为BeanA@1234，上面代码中的 bean 变量指向就是这个对象。

2.2. 暴露早期引用

该方法用于把早期对象包装成一个ObjectFactory 暴露到三级缓存中 用于将解决循环依赖...

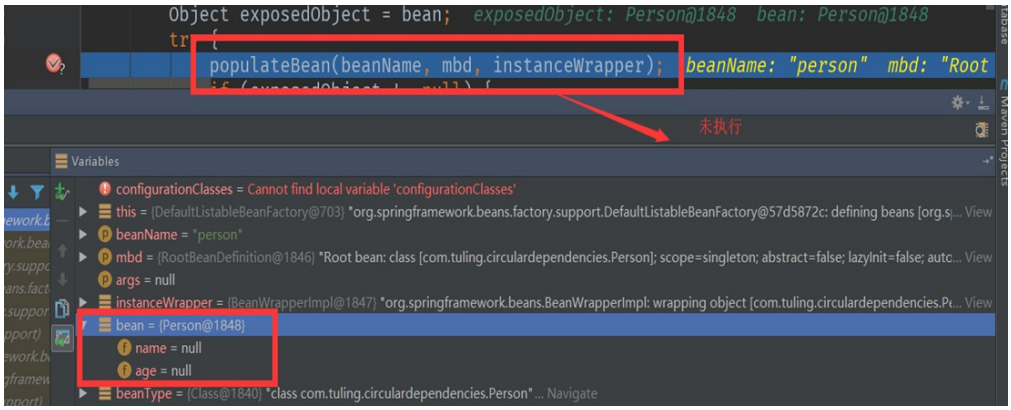
```

protected void addSingletonFactory(String beanName, ObjectFactory<?> singletonFactory) {
...
        //加入到三级缓存中，，，，，暴露早期对象用于解决循环依赖
        this.singletonFactories.put(beanName, singletonFactory);
 ...
}
```

beanA 指向的原始对象创建好后，就开始把指向原始对象的引用通过 ObjectFactory 暴露出去。getEarlyBeanReference 方法的第三个参数 bean 指向的正是 createBeanInstance 方法创建出原始 bean 对象 BeanA@1234。

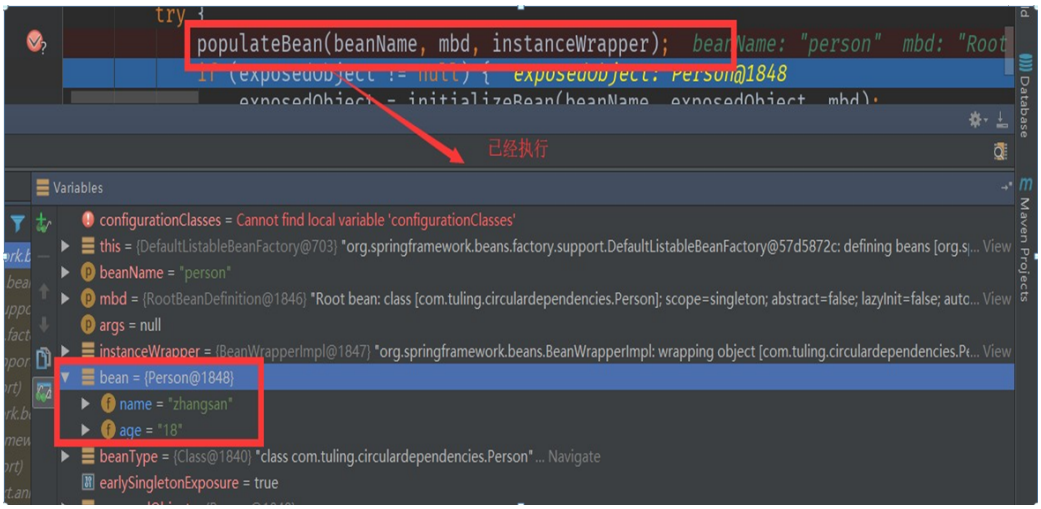
2.3. 解析依赖

```
populateBean(beanName, mbd, instanceWrapper);
```

还没有进行属性装配，自动注入的属性都是null



初始化好的Bean



populateBean 用于向 beanA 这个原始对象中填充属性，当它检测到 beanA 依赖于 beanB 时，会首先去实例化 beanB。

beanB 在此方法处也会解析自己的依赖，当它检测到 beanA 这个依赖，于是调用 BeanFactroy.getBean("beanA") 这个方法，从容器中获取 beanA。

2.4. 获取早期引用

```java
protected Object getSingleton(String beanName, boolean allowEarlyReference) {
   /**
    * 第一步:我们尝试去一级缓存(单例缓存池中去获取对象,一般情况从该map中获取的对象是直接可以使用的)
    * IOC容器初始化加载单实例bean的时候第一次进来的时候 该map中一般返回空
    */
   Object singletonObject = this.singletonObjects.get(beanName);
   /**
    * 若在第一级缓存中没有获取到对象,并且singletonsCurrentlyInCreation这个list包含该beanName
    * IOC容器初始化加载单实例bean的时候第一次进来的时候 该list中一般返回空,但是循环依赖的时候可以满足该条件
    */
   if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {
      synchronized (this.singletonObjects) {
         /**
          * 尝试去二级缓存中获取对象(二级缓存中的对象是一个早期对象)
          * 何为早期对象:就是bean刚刚调用了构造方法，还来不及给bean的属性进行赋值的对象(纯净态)
          * 就是早期对象
          */
         singletonObject = this.earlySingletonObjects.get(beanName);
         /**
          * 二级缓存中也没有获取到对象,allowEarlyReference为true(参数是有上一个方法传递进来的true)
          */
         if (singletonObject == null && allowEarlyReference) {
            /**
             * 直接从三级缓存中获取 ObjectFactory对象 这个对接就是用来解决循环依赖的关键所在
             * 在ioc后期的过程中,当bean调用了构造方法的时候,把早期对象包裹成一个ObjectFactory
             * 暴露到三级缓存中
             */
            ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);
            //从三级缓存中获取到对象不为空
            if (singletonFactory != null) {
               /**
                * 在这里通过暴露的ObjectFactory 包装对象中,通过调用他的getObject()来获取我们的早期对象
                * 在这个环节中会调用到 getEarlyBeanReference()来进行后置处理
                */
               singletonObject = singletonFactory.getObject();
               //把早期对象放置在二级缓存,
               this.earlySingletonObjects.put(beanName, singletonObject);
               //ObjectFactory 包装对象从三级缓存中删除掉
               this.singletonFactories.remove(beanName);
            }
         }
      }
   }
   return singletonObject;
}
```

接着上面的步骤讲:

1.populateBean 调用 BeanFactroy.getBean("beanA") 以获取 beanB 的依赖。

2.getBean("beanB") 会先调用 getSingleton("beanA")，尝试从缓存中获取 beanA。此时由于 beanA 还没完全实例化好

3.于是 this.singletonObjects.get("beanA") 返回 null。

4.接着 this.earlySingletonObjects.get("beanA") 也返回空，因为 beanA 早期引用还没放入到这个缓存中。

5.最后调用 singletonFactory.getObject() 返回 singletonObject，此时 singletonObject != null。singletonObject 指向 BeanA@1234，也就是 createBeanInstance 创建的原始对象。此时 beanB 获取到了这个原始对象的引用，beanB 就能顺利完成实例化。beanB 完成实例化后，beanA 就能获取到 beanB 所指向的实例，beanA 随之也完成了实例化工作。由于 beanB.beanA 和 beanA 指向的是同一个对象 BeanA@1234，所以 beanB 中的 beanA 此时也处于可用状态了。

关于5.3.10版本的改动

```
protected Object getSingleton(String beanName, boolean allowEarlyReference) {
    // Quick check for existing instance without full singleton lock
    Object singletonObject = this.singletonObjects.get(beanName);
    if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {
       singletonObject = this.earlySingletonObjects.get(beanName);
       if (singletonObject == null && allowEarlyReference) {
          synchronized (this.singletonObjects) {
             // Consistent creation of early reference within full singleton lock
             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;
}

```

5.3.10版本略有改动 这个细节可以跳过。好奇宝宝有兴趣可以接着看：

会发现5.3.10版本将锁粒度降低了， 把二级缓存获取放在锁外了。

为什么要这么改：

避免 DefaultSingletonBeanRegistry.getSingleton(beanName, false) 的单例池锁定

- 弊端： 超极端情况会获取二级缓存的早期bean
- 好处： 降低了锁的粒度， 降低了死锁可能性
  - **粒度**： allowEarlyReference=false 不用阻塞，直接出货。 之前版本无论true或false都要阻塞
  - **死锁**：

举个例子： 如果线程1 锁了一个**共享资源**

此时 线程2 getBean，锁住了单例池，在创建过程获取那个**共享资源**--->由于被线程1锁住了，阻塞...

此时线程1 getBeansOfType---->getSingleton(bean,**false**)，想获取单例池锁（由于被线程2持有）。 形成了死锁

而5.3.10版本这样做如果allowEarlyReference=false则不会进入锁

2.5 初始化

正常给bean进行初始化回调、aware、beanpostprocessor等... 这里没什么好说的

2.6 初始化后

初始化后还有个地方跟循环依赖又关系：

如果A是动态代理对象， 通过三级缓存循环依赖后， B.A=A的代理。 但是A本身还是一个普通对象。所以要把二级缓存的代理对象重新赋给beanA:

```java
if (earlySingletonExposure) {
    // 1. 从二级缓存中拿到代理对象
    Object earlySingletonReference = getSingleton(beanName, false);
    if (earlySingletonReference != null) {
        // 2.赋给当前对象
       if (exposedObject == bean) {
          exposedObject = earlySingletonReference;
       }
       else if (!this.allowRawInjectionDespiteWrapping && hasDependentBean(beanName)) {
          // beanName被哪些bean依赖了，现在发现beanName所对应的bean对象发生了改变，那么则会报错
          String[] dependentBeans = getDependentBeans(beanName);
          Set<String> actualDependentBeans = new LinkedHashSet<>(dependentBeans.length);
          for (String dependentBean : dependentBeans) {
             if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {
                actualDependentBeans.add(dependentBean);
             }
          }
          if (!actualDependentBeans.isEmpty()) {
             throw new BeanCurrentlyInCreationException(beanName,
                   "Bean with name '" + beanName + "' has been injected into other beans [" +
                   StringUtils.collectionToCommaDelimitedString(actualDependentBeans) +
                   "] in its raw version as part of a circular reference, but has eventually been " +
                   "wrapped. This means that said other beans do not use the final version of the " +
                   "bean. This is often the result of over-eager type matching - consider using " +
                   "'getBeanNamesForType' with the 'allowEagerInit' flag turned off, for example.");
          }
       }
    }
}
```

循环依赖可以关闭吗

可以，Spring提供了这个功能，我们需要这么写,SpringBoot2.6.0+就是这么干的：

```java
public class Main {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext applicationContext = new AnnotationConfigApplicationContext();
        applicationContext.setAllowCircularReferences(false);
        applicationContext.register(AppConfig.class);
        applicationContext.refresh();
    }
}

```

关闭了如何解决循环依赖?

1. @Lazy延迟注入

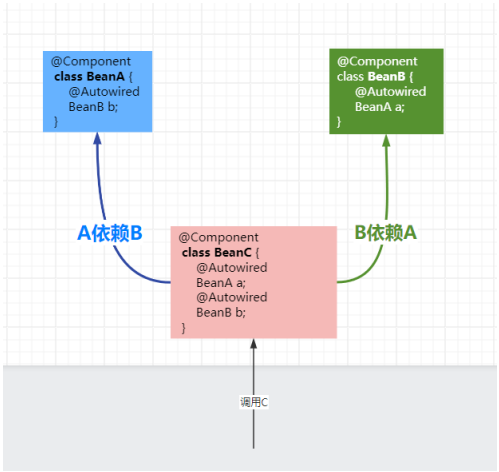
```java
@Lazy
   private IInstanceB instanceB;
```

@Lazy加在属性上， 代表不在spring容器加载时注入， **会临时注入一个代理对象**， 等真正用到再通过代理对象调用最终bean的方法

2. 解除循环依赖

2种方式思路：

1. 把依赖的方法， 直接写在本类中，断绝其中一方依赖关系
2. 添加一个中间类， 中间类去依赖A\B， 然后让中间类去组织他们的依赖方法



3. SpringBoot开启循环依赖

不建议哈，其实循环依赖本身是一种代码不规范的设计，Spring给我们解决了。Spring作者都已表示可能会在后续版本去掉循环依赖支持。 除非你是把Spring代码移植到SpringBoot ，可以考虑开启循环依赖已保证之前代码正常性。

```
spring.main.allow-circular-references=true
```

AOP下的循环依赖注意事项详解

问题：

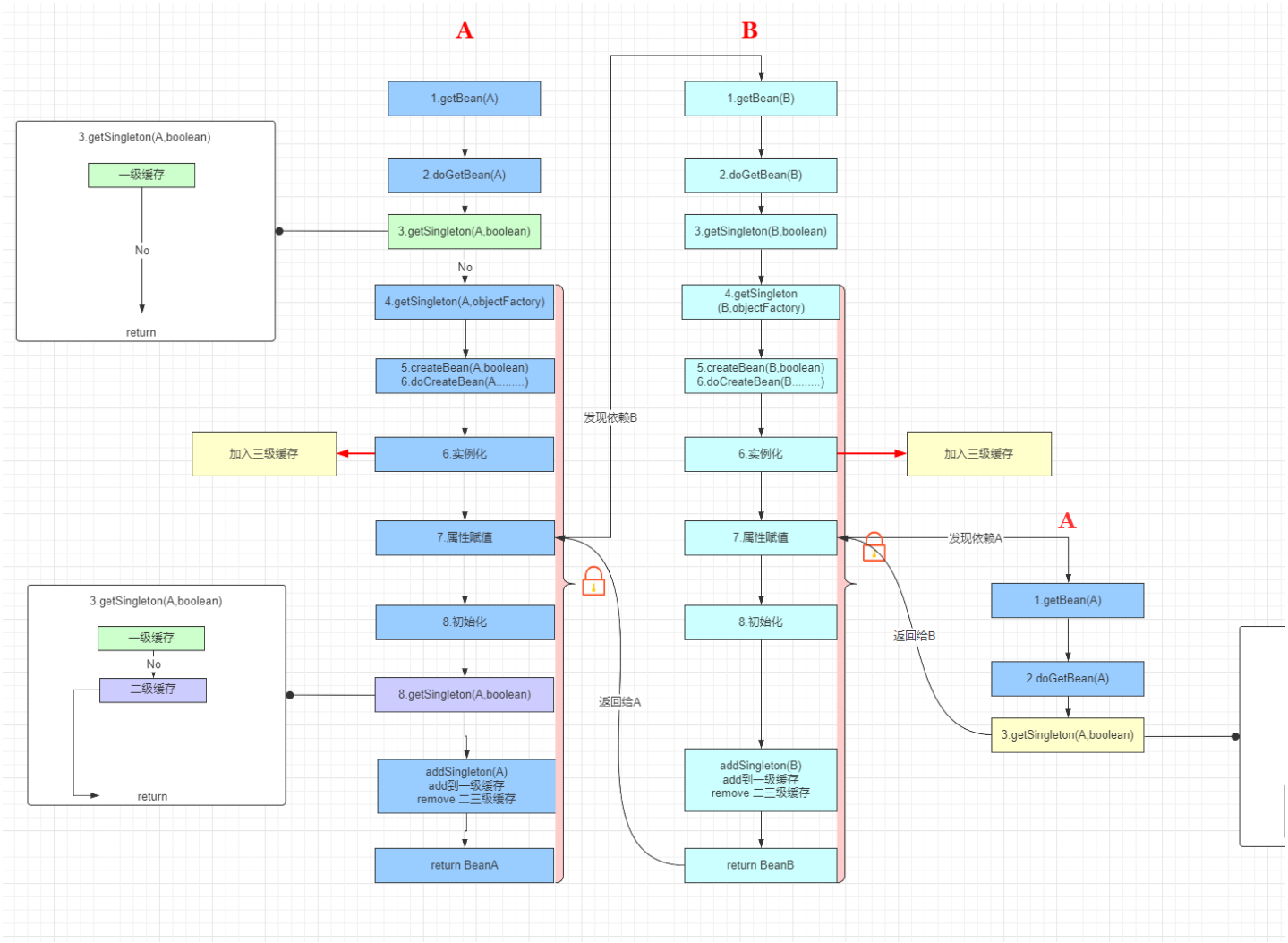
1. 什么情况会抛出throw new BeanCurrentlyInCreationException异常?

要满足3个条件： 1. 是循环依赖 2. AOP了 3. 初始化后改变了当前bean对象

- @Async 会在初始化创建动态代理，导致2级缓存的代理和当前对象不一致，故会抛出throw new BeanCurrentlyInCreationException异常
- @Transactional也是动态代理？为什么不会报错， 因为事务的动态代理和AOP实际上是一个代理对象， 后面学了事务源码就知道了

以上的过程对应下面的流程图：

https://www.processon.com/view/link/5f1fb2cf1e08533a628a7b4c



有道云链接：https://note.youdao.com/s/YG3B67I2