1 问题来源

- `spring` 将 `bean` 的初始化和各个 `bean` 之间的关系交给了框架来处理，大大了简化了项目搭建的流程
- 但是在简化的同时也隐藏了 `bean` 之间初始化关系的流程
- 熟悉 `bean` 初始化的流程可以更好的把握项目运行的顺序，也能够更好的把握 `spring` 中的一些扩展点

2 bean 初始化流程

- `spring` 启动的整个过程可以最简化为 **两个步骤**： 1) 获取所有的 `beanDefinition` 2) 遍历所有的 `beanDefinition` ，实例化相应的 `bean` (非懒加载的 `bean`)

```
//AbstractApplicationContext类中的方法
public void refresh() {
    //这一步是：获取注册所有的 beanDefinition
    this.invokeBeanFactoryPostProcessors(beanFactory);
    //这一步是：遍历所有的 beanDefinition，实例化相应的bean(非懒加载的bean)
    this.finishBeanFactoryInitialization(beanFactory);
}
```

2.1 getBean->doGetBean->createBean

- 通过别名获取 `beanName`
- 判断缓存中是否已经存在此实例，如果有，打出相应日志
- 如果没有：按照 `bean` 的类型，分情况处理
- 将父类属性合并到此类中
- 递归初始化依赖类（这里是 `xml` 中定义的 `depends-on` 属性）
- 初始化该类

```
//finishBeanFactoryInitialization最终会遍历beanDefinition，调用getBean方法初始化bean
public Object getBean(String name, Object... args) throws BeansException {
    return this.doGetBean(name, (Class)null, args, false);
}
protected <T> T doGetBean(String name, @Nullable Class<T> requiredType, @Nullable
Object[] args, boolean typeCheckOnly) throws BeansException {
    //这里是通过别名获取beanName（这样就可以注册别名了）
    String beanName = this.transformedBeanName(name);
    //检查缓存中是否已经存在了bean实例
    //在按beanDefinitions顺序实例化时，如果beanA内依赖了beanB，那么将会先去初始化beanB
    //当初始化beanA后，在往下初始化可能会又遍历到beanB，这时beanB就已经存在实例了
    Object sharedInstance = this.getSingleton(beanName);
    Object bean;
    //如果有，判断bean是否正在创建，打相应的日志
    if (sharedInstance != null && args == null) {
        if (this.logger.isTraceEnabled()) {
            if (this.isSingletonCurrentlyInCreation(beanName)) {
                this.logger.trace("Returning eagerly cached instance of singleton bean
'" + beanName + "' that is not fully initialized yet - a consequence of a circular
reference");
            } else {
                this.logger.trace("Returning cached instance of singleton bean '" +
beanName + "'");
            }
        }
        bean = this.getObjectForBeanInstance(sharedInstance, name, beanName,
(RootBeanDefinition)null);
    } else {
        //如果是多例模式，正在创建，那肯定有问题，因为多例是互不相关的，直接报错
        if (this.isPrototypeCurrentlyInCreation(beanName)) {
            throw new BeanCurrentlyInCreationException(beanName);
        }
        //在父类上下文中找该bean，找到了就返回(代码略)
        //获取缓存的BeanDefinition对象并合并其父类和本身的属性
        final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);

        String[] dependsOn = mbd.getDependsOn();//获取依赖类，这个属性是在xml中定义的，获取如
上
        //递归初始化依赖类，注意这里的依赖类，只是说明在实例化本类之前，需要先实例化依赖类
        //xml中的 depends-on属性 主要是用来 指定 各个bean之间的初始化顺序
        //当然这里也会出现循环 dependsOn 的情况，如果出现，会有以下代码检测到，直接报错，不然不就死循
环了嘛
        //if (this.isDependent(beanName, dep)) {
        // throw new BeanCreationException(...)
        //}
```

```
        if (mbd.isSingleton()) {
            //单例初始化
            sharedInstance = createBean();
            bean = this.getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
        } else if (mbd.isPrototype()) {
            //多例初始化，差不多
        } else {
            //其它类型（如Session等）
            String scopeName = mbd.getScope();
            Scope scope = (Scope)this.scopes.get(scopeName);
            //初始化
        }
    }
    if (requiredType != null && !requiredType.isInstance(bean)) {
        //如果!requiredType.isInstance(bean)，要做什么转换吧
        T convertedBean = this.getTypeConverter().convertIfNecessary(bean,
requiredType);
        return convertedBean;
    } else {
        return bean;
    }
}
```

2.2 createBean->doCreateBean

- 设置 `override` 属性
- 调用所有 `InstantiationAwareBeanPostProcessor`，做一些扩展

```
protected Object createBean(String beanName, RootBeanDefinition mbd, @Nullable Object[]
args) throws BeanCreationException {
    RootBeanDefinition mbdToUse = mbd;
    //设置一些mbdToUse属性（代码略）
    //如果有子父类有相同的方法，设置override属性
    mbdToUse.prepareMethodOverrides();
    Object beanInstance;
    //if (targetType != null) {
    //    bean = this.applyBeanPostProcessorsBeforeInstantiation(targetType, beanName);
    //    if (bean != null) {
    //        bean = this.applyBeanPostProcessorsAfterInitialization(bean, beanName);
    //    }
    // }
    //给beanPostProcessor一个返回代理而不是目标bean实例的机会，这里就是一个扩展点
    beanInstance = this.resolveBeforeInstantiation(beanName, mbdToUse);
    //如果这边就已经可以初始化了，那就直接返回了，不会有依赖处理不来的问题（因为之前已经处理依赖了）
    if (beanInstance != null) {
        return beanInstance;
    }
    //4
    beanInstance = this.doCreateBean(beanName, mbdToUse, args);
    return beanInstance;
}
```

2.3 doCreateBean->(createBeanInstance,populateBean,initializeBean,registerDisposableBeanIfNecessary)

- `createBeanInstance`：通过策略创建bean包装实例——这里可能回去递归初始化构造器中参数的类
- 调用所有的 `MergedBeanDefinitionPostProcessors`，一个扩展点
- 提前暴露引用，允许循环引用，一般为 `true`
- `populateBean`：注入依赖类和相关属性——这里递归初始化注解的依赖类
- `initializeBean`：调用初始化方法
- `registerDisposableBeanIfNecessary`：搬到 `bean` 销毁的方法

```
protected Object doCreateBean(String beanName, RootBeanDefinition mbd, @Nullable
Object[] args) throws BeanCreationException {
    BeanWrapper instanceWrapper = null;
    if (mbd.isSingleton()) {
        //相当于pop
        instanceWrapper = (BeanWrapper)this.factoryBeanInstanceCache.remove(beanName);
    }

    if (instanceWrapper == null) {
        //如果没有，就创建，通过反射创建wrapper实例
        instanceWrapper = this.createBeanInstance(beanName, mbd, args);
    }
    //包装类
    Object bean = instanceWrapper.getWrappedInstance();
    Class<?> beanType = instanceWrapper.getWrappedClass();
    if (beanType != NullBean.class) {
        mbd.resolvedTargetType = beanType;
    }

    Object var7 = mbd.postProcessingLock;
    synchronized(mbd.postProcessingLock) {
        //如果还没处理
        if (!mbd.postProcessed) {
            //调用所有的MergedBeanDefinitionPostProcessors，一个扩展点
            this.applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);
            mbd.postProcessed = true;
        }
    }
    //提前暴露引用，允许循环引用，一般为true
    boolean earlySingletonExposure = mbd.isSingleton() && this.allowCircularReferences
&& this.isSingletonCurrentlyInCreation(beanName);
    if (earlySingletonExposure) {
        //打日志代码略
        this.addSingletonFactory(beanName, () -> {
            return this.getEarlyBeanReference(beanName, mbd, bean);
        });
    }
    Object exposedObject = bean;
    //5
```

```
        this.populateBean(beanName, mbd, instanceWrapper);
        //6
        exposedObject = this.initializeBean(beanName, exposedObject, mbd);
        //处理earlySingletonExposure代码略

        //注册（绑定）注解的DisposableBean方法，bean销毁时就可以调用
        this.registerDisposableBeanIfNecessary(beanName, bean, mbd);
        return exposedObject;
    }
```

2.4 createBeanInstance

- 使用适当的实例化策略为指定 `bean` 创建一个新实例：工厂方法、构造函数自动注入或简单的实例化。
- 详细参考链接

```
protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition mbd,
@Nullable Object[] args) {
    // Make sure bean class is actually resolved at this point.
    Class<?> beanClass = resolveBeanClass(mbd, beanName);

    Supplier<?> instanceSupplier = mbd.getInstanceSupplier();
    //
    if (instanceSupplier != null) {
        return obtainFromSupplier(instanceSupplier, beanName);
    }
    //通过工厂方法创建
    if (mbd.getFactoryMethodName() != null) {
        return instantiateUsingFactoryMethod(beanName, mbd, args);
    }

    // Shortcut when re-creating the same bean...
    boolean resolved = false;
    boolean autowireNecessary = false;
    if (args == null) {
        synchronized (mbd.constructorArgumentLock) {
            if (mbd.resolvedConstructorOrFactoryMethod != null) {
                resolved = true;
                autowireNecessary = mbd.constructorArgumentsResolved;
            }
        }
    }
    if (resolved) {
        if (autowireNecessary) {
            //构造函数自动注入
            return autowireConstructor(beanName, mbd, null, null);
        }
        else {
            //使用默认构造器
            return instantiateBean(beanName, mbd);
        }
    }
```

```
    // Candidate constructors for autowiring?
    Constructor<?>[] ctors = determineConstructorsFromBeanPostProcessors(beanClass,
beanName);
    //如果满足
    if (ctors != null || mbd.getResolvedAutowireMode() == AUTOWIRE_CONSTRUCTOR ||
            mbd.hasConstructorArgumentValues() || !ObjectUtils.isEmpty(args)) {
        //使用策略模式，通过构造函数、参数实例化bean
        return autowireConstructor(beanName, mbd, ctors, args);
    }

    // 需要根据参数解析、确定构造函数
    ctors = mbd.getPreferredConstructors();
    if (ctors != null) {
        return autowireConstructor(beanName, mbd, ctors, null);
    }

    //simply use no-arg constructor.
    return instantiateBean(beanName, mbd);
}
```

2.5 populateBean

- 将本类的依赖类注入到类中
- 将所有 `ProtertyValues` 中的属性填充至本类中
- [populateBean实现 依赖注入源码解析]
- [Spring源码：PropertyValues类及属性注入一]

```
protected void populateBean(String beanName, RootBeanDefinition mbd, @Nullable
BeanWrapper bw) {

       boolean continueWithPropertyPopulation = true;
       //遍历InstantiationAwareBeanPostProcessor，代理原始类，这里就可以扩展
       if (!mbd.isSynthetic() && this.hasInstantiationAwareBeanPostProcessors()) {
           Iterator var5 = this.getBeanPostProcessors().iterator();
           while(var5.hasNext()) {
               BeanPostProcessor bp = (BeanPostProcessor)var5.next();
               if (bp instanceof InstantiationAwareBeanPostProcessor) {
                   InstantiationAwareBeanPostProcessor ibp =
(InstantiationAwareBeanPostProcessor)bp;
                   if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(),
beanName)) {
                       continueWithPropertyPopulation = false;
                       break;
                   }
               }
           }
       }

       if (continueWithPropertyPopulation) {
           PropertyValues pvs = mbd.hasPropertyValues() ? mbd.getPropertyValues() :
null;
```

```
            if (mbd.getResolvedAutowireMode() == 1 || mbd.getResolvedAutowireMode() ==
2) {
                //按照Name或者Type方式注入关联类（关联类已经在2.1中实例化了）
                MutablePropertyValues newPvs = new
MutablePropertyValues((PropertyValues)pvs);
                if (mbd.getResolvedAutowireMode() == 1) {
                    this.autowireByName(beanName, mbd, bw, newPvs);
                }
                if (mbd.getResolvedAutowireMode() == 2) {
                    this.autowireByType(beanName, mbd, bw, newPvs);
                }

                pvs = newPvs;
            }

            boolean hasInstAwareBpps = this.hasInstantiationAwareBeanPostProcessors();
            // 是否进行依赖检查
            boolean needsDepCheck = mbd.getDependencyCheck() != 0;
            PropertyDescriptor[] filteredPds = null;
            if (hasInstAwareBpps) {
                if (pvs == null) {
                    pvs = mbd.getPropertyValues();
                }
                //对属性做后置处理
                Iterator var9 = this.getBeanPostProcessors().iterator();
                while(var9.hasNext()) {
                    BeanPostProcessor bp = (BeanPostProcessor)var9.next();
                    if (bp instanceof InstantiationAwareBeanPostProcessor) {
                        InstantiationAwareBeanPostProcessor ibp =
(InstantiationAwareBeanPostProcessor)bp;
                        //这里就会出来@Resource @Auotwrid 注入的依赖类，可以产生循环依赖
                        PropertyValues pvsToUse =
ibp.postProcessProperties((PropertyValues)pvs, bw.getWrappedInstance(), beanName);
                        if (pvsToUse == null) {
                            if (filteredPds == null) {
                                filteredPds =
this.filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);
                            }

                            pvsToUse =
ibp.postProcessPropertyValues((PropertyValues)pvs, filteredPds,
bw.getWrappedInstance(), beanName);
                            if (pvsToUse == null) {
                                return;
                            }
                        }
                        pvs = pvsToUse;
                    }
                }
            }
            if (needsDepCheck) {
                if (filteredPds == null) {
```

```
                    filteredPds = this.filterPropertyDescriptorsForDependencyCheck(bw,
mbd.allowCaching);
                }

                this.checkDependencies(beanName, mbd, filteredPds, (PropertyValues)pvs);
            }
            if (pvs != null) {
                //将所有ProtertyValues中的属性填充至BeanWrapper中
                //https://blog.csdn.net/zhuqiuhui/article/details/82391836，这里简单介绍了
                this.applyPropertyValues(beanName, mbd, bw, (PropertyValues)pvs);
            }
        }
}
```

2.6 initializeBean

- 统一调用 `bean.setxxxAware`;如果该类实现了某个 `aware接口` 的话
- 初始化之前调用，遍历所有的 `beanPostProcessorsBefore`
- 调用初始化方法：`afterPropertiesSet()` 和 `initMethod()`
- 调用 `BeanPostProcessorsAfter`

```
protected Object initializeBean(String beanName, Object bean, @Nullable
RootBeanDefinition mbd) {
    if (System.getSecurityManager() != null) {
        AccessController.doPrivileged(() -> {
            this.invokeAwareMethods(beanName, bean);
            return null;
        }, this.getAccessControlContext());
    } else {
        //就3中AwareMethods（BeanNameAware，BeanClassLoaderAware，BeanFactoryAware）
        //统一调用bean.setxxxAware;如果该类实现了某个aware接口的话
        this.invokeAwareMethods(beanName, bean);
    }
    Object wrappedBean = bean;
    if (mbd == null || !mbd.isSynthetic()) {
        //初始化之前调用，遍历所有的beanPostProcessors，这里调用了@PostConstruct注解的初始化方
法！通过CommonAnnotationPostprocessor
        wrappedBean = this.applyBeanPostProcessorsBeforeInitialization(bean, beanName);
    }

    //调用初始化方法，继承了InitializingBean（调用它的afterPropertiesSet方法）或指定了
InitMethod（通过注解等）先调用afterPropertiesSet()，后调用
initMethod(@Bean(initMethod="..."))
    this.invokeInitMethods(beanName, wrappedBean, mbd);

    if (mbd == null || !mbd.isSynthetic()) {
        //初始化之后调用
        wrappedBean = this.applyBeanPostProcessorsAfterInitialization(wrappedBean,
beanName);
    }
    return wrappedBean;
}
```

2.7 registerDisposableBeanIfNecessary

- 关键在 `DisposableBeanAdapter` 中解析(绑定)了类的 `destroy` 方法
- 这样 `bean` 销毁时，就会调用相应的方法，对应于 `initializeBean`

```
//关键在DisposableBeanAdapter中解析(绑定)了类的destroy方法，不具体分析了
protected void registerDisposableBeanIfNecessary(String beanName, Object bean,
RootBeanDefinition mbd) {
   AccessControlContext acc = (System.getSecurityManager() != null ?
getAccessControlContext() : null);
   if (!mbd.isPrototype() && requiresDestruction(bean, mbd)) {
      if (mbd.isSingleton()) {
         // Register a DisposableBean implementation that performs all destruction
         // work for the given bean: DestructionAwareBeanPostProcessors,
         // DisposableBean interface, custom destroy method.
         registerDisposableBean(beanName,
               new DisposableBeanAdapter(bean, beanName, mbd, getBeanPostProcessors(),
acc));
      }
      else {
         // A bean with a custom scope...
         Scope scope = this.scopes.get(mbd.getScope());
         if (scope == null) {
            throw new IllegalStateException("No Scope registered for scope name '" +
mbd.getScope() + "'");
         }
         scope.registerDestructionCallback(beanName,
               new DisposableBeanAdapter(bean, beanName, mbd, getBeanPostProcessors(),
acc));
      }
   }
}
```

3 各种 PostProcessor 解析

- 在以上分析初始化流程时，用到了各种后置处理器，统计为以下几种：

3.1 InstantiationAwareBeanPostProcessor

- 这个处理器接口有3个方法：

```
    //在2.2 createBean 位置，createBeanInstance()实例化bean之前调用这个beanPostProcessor
    //返回的可能是一个代理。如果返回的不是null，则不会继续执行默认的创建流程，而直接执行
postProcessAfterInitialization，完成该bean的初始化
    default Object postProcessBeforeInstantiation(Class<?> beanClass, String beanName)
{}
    //在bean实例化后，设置自定义属性（populateBean#postProcessProperties）之前调用，
    //通过这个可以自定义设置属性。返回true时，后面同样的后置处理器将不会调用了
    default boolean postProcessAfterInstantiation(Object bean, String beanName){}
    //在postProcessAfterInstantiation之后，设置默认设置属性
(populateBean#applyPropertyValues)之前调用
    //自定义设置属性，如果返回的不为null，就不进行默认的属性设置了，主要完成其他定制的一些依赖注入和依
赖检查等，
    //如AutowiredAnnotationBeanPostProcessor执行@Autowired注解注入，
CommonAnnotationBeanPostProcessor执行@Resource等注解的注入，
PersistenceAnnotationBeanPostProcessor执行@ PersistenceContext等JPA注解的注入，
RequiredAnnotationBeanPostProcessor执行@ Required注解的检查等等
    default PropertyValues postProcessProperties(PropertyValues pvs, Object bean,
String beanName){}
```

3.2 MergedBeanDefinitionPostProcessors

- 这个处理器接口有2个方法：

```
//实例化bean（createBeanInstance()）之后，注入bean属性(populateBean())之前调用
//使bean的定义更明确（没有理解），查看了一些实现类，主要做一些检查处理
void postProcessMergedBeanDefinition(RootBeanDefinition var1, Class<?> var2, String
var3);
//指定名称的bean定义已重置的通知，还有主要是执行各种清除，暂时不理解
default void resetBeanDefinition(String beanName) {}
```

3.3 SmartInstantiationAwareBeanPostProcessor

- 这个处理器接口有3个方法：

```
//在许多地方被调用，用来获取bean的类型
default Class<?> predictBeanType(Class<?> beanClass, String beanName){}
//在createBeanInstance内被调用，用来智能获取最终初始化bean所需的构造方法（构造方法可能有好几个，选取
一个）
default Constructor<?>[] determineCandidateConstructors(Class<?> beanClass, String
beanName){}
//获取对指定bean的早期访问的引用，通常是为了解决循环依赖问题（A->B->C->A），C初始化的时候需要A，而A正
在初始化，就可以提早暴露引用，避免报错
//在createBeanInstance之后，populateBean之前被调用
default Object getEarlyBeanReference(Object bean, String beanName){}
```

3.4 BeanPostProcessor

- 这个处理器接口有2个方法：

```
//在initializeBean()方法中调用，此时类的属性已经注入
//完成一些定制的初始化任务，如@Valid验证，@PostConstruct调用
default Object postProcessBeforeInitialization(Object bean, String beanName){}
//实例化、依赖注入、所有初始化方法完毕后调用，主要做一些AOP代理
default Object postProcessAfterInitialization(Object bean, String beanName){}
```

- 在 `DataSourceAutoConfiguration` 中就注册了一个 `DataSourceInitializerPostProcessor`

```
//这个类在postProcessAfterInitialization 中 初始化了 DataSourceInitializerInvoker类，用于自动运行sql文件
class DataSourceInitializerPostProcessor implements BeanPostProcessor, Ordered {
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        if (bean instanceof DataSource) {
            this.beanFactory.getBean(DataSourceInitializerInvoker.class);
        }
        return bean;
    }
}
```

3.5 有很多后置处理器，都是主要作用在类实例化完成之后

- 类中用注解声明的属性，比如@PostConstruct、@Resource、@Autowired、@Value、@Required、@AspectJ、@Valid、@Scheduled、@ Async等等注解，都是通过注册后置处理器来扩展的。

4 几个 Aware 接口

- 在 `initializeBean()` 方法最开始，就调用了 `invokeAwareMethods`，用来处理类是否实现了几个 `Aware` 接口

```
private void invokeAwareMethods(final String beanName, final Object bean) {
    if (bean instanceof Aware) {
        //如果类实现了BeanNameAware接口，调用setBeanName，设置类的beanName属性
        if (bean instanceof BeanNameAware) {
            ((BeanNameAware) bean).setBeanName(beanName);
        }
        //如果类实现了BeanClassLoaderAware，设置类的BeanClassLoader属性
        if (bean instanceof BeanClassLoaderAware) {
            ClassLoader bcl = getBeanClassLoader();
            if (bcl != null) {
                ((BeanClassLoaderAware) bean).setBeanClassLoader(bcl);
            }
        }
        //如果类实现了BeanFactoryAware，这个类就是工厂bean，因为这个类中有工厂属性
        if (bean instanceof BeanFactoryAware) {
            ((BeanFactoryAware)
bean).setBeanFactory(AbstractAutowireCapableBeanFactory.this);
        }
```

```
    }
}
```

5 循环依赖产生原因

- 循环依赖测试源代码

5.1 有两种情况可以产生循环依赖：构造函数法、其它方法(注解注入、xml 中的 depends-on)

5.2 其它方法产生原因代码跟踪

- 如项目中所示：`ClassA` 关联 `ClassB`，`ClassB` 关联 `ClassC`，`ClassC` 关联 `ClassA`
- 这样初始化任何一个类的时候，都会递归去初始化关联类，导致循环依赖
- **总结：** 关联依赖，能够成功初始化的原因在于，在递归初始化关联类之前，已经把本类的引用给暴露出去了，这样关联类完全可以只先指向它的引用，而先不用关心这个引用到底有没有执行完全

```
//按照这3个类在项目的顺序，首先初始化的是ClassA，通过getBean('classA')调用，开始分析
getBean(beanName){}
doGetBean(beanName){
   //这里在 3级缓存中去找，找到了返回实例，否则返回 null
   //1 singletonObjects 2 earlySingletonObjects 3 singletonFactories
   //用到了：isSingletonCurrentlyInCreation map
   //如果在3级中找到了，就将3级中的这个移除到2级中去，为以后调用此类提高效率
   //ClassA肯定没找到
   Object sharedInstance = this.getSingleton(beanName);
   if (!typeCheckOnly) {//typeCheckOnly一般为flase,只有一个地方为true：
getTypeForFactoryBean
       this.markBeanAsCreated(beanName);//这里标记此类已经创建，put 到 alreadyCreated 中
   }
    //if (ele.hasAttribute(DEPENDS_ON_ATTRIBUTE)) {
    //  String dependsOn = ele.getAttribute(DEPENDS_ON_ATTRIBUTE);
    //  bd.setDependsOn(StringUtils.tokenizeToStringArray(dependsOn,
MULTI_VALUE_ATTRIBUTE_DELIMITERS));
    //}
   String[] dependsOn = mbd.getDependsOn();//获取依赖类，这个属性是在xml中定义的，获取如上
   //递归初始化依赖类，注意这里的依赖类，只是说明在实例化本类之前，需要先实例化依赖类
   //xml中的 depends-on属性 主要是用来 指定 各个bean之间的初始化顺序
   //当然这里也会出现循环 dependsOn 的情况，如果出现，会有以下代码检测到，直接报错，不然不就死循环了嘛
   //if (this.isDependent(beanName, dep)) {
   // throw new BeanCreationException(...)
   //}

   //在getSingleton中 将 beanName加入 isSingletonCurrentlyInCreation----------关键
   //之后，其中第二个lamda参数，在getSingleton中 传给了 singletonFactory.getObject();//这里开
始获取bean，走createBean()方法了
   sharedInstance = this.getSingleton(beanName, () -> {
       try {
```

```
            return this.createBean(beanName, mbd, args);
        } catch (BeansException var5) {
            this.destroySingleton(beanName);
            throw var5;
        }
    });
}
//----一直到创建实例结束 this.createBeanInstance(beanName, mbd, args)，1、2、3级缓存中都没有任何这3个类
//在往后走，this.singletonFactories.put(beanName, singletonFactory);将这个lamda表达式put到3级缓存中去了
//至此，3级缓存中有 classA了
//-------------------------------------------------------用于回调
this.addSingletonFactory(beanName, () -> {
    return this.getEarlyBeanReference(beanName, mbd, bean);
});

//---一直走到populate中的
if (bp instanceof InstantiationAwareBeanPostProcessor) {
    //其中有个CommonAnnotationBeanPostProcessor实现了 @Resouce 的注入，调用其中的
    ibp.postProcessProperties();//开始初始化 classB
}

//-----------------关键--------------------
//---同理 classB 又走到了 ibp.postProcessProperties(); 初始化 classC
//---同理 classC 又走到了 ibp.postProcessProperties(); 初始化 classA
//-----------------关键--------------------
//-----------------关键--------------------
//---这时开始 初始化 classA ，调用 getBean()方法，走到
doGetBean#this.getSingleton(beanName);
 protected Object getSingleton(String beanName, boolean allowEarlyReference) {
     //这里没找到
     Object singletonObject = this.singletonObjects.get(beanName);
     //isSingletonCurrentlyInCreation("classA")在第一次初始化的时候，已经放进去了，返回true
     if (singletonObject == null && this.isSingletonCurrentlyInCreation(beanName)) {
         Map var4 = this.singletonObjects;
         synchronized(this.singletonObjects) {
             //没找到
             singletonObject = this.earlySingletonObjects.get(beanName);
             if (singletonObject == null && allowEarlyReference) {
                //-------------------------------------------------------用在这里回调，得到3级缓存
                 ObjectFactory<?> singletonFactory =
(ObjectFactory)this.singletonFactories.get(beanName);
                 if (singletonFactory != null) {
                     singletonObject = singletonFactory.getObject();
                     //放入2级缓存，提交性能
                     this.earlySingletonObjects.put(beanName, singletonObject);
                     //移除3级缓存
                     this.singletonFactories.remove(beanName);
                 }
             }
         }
```

```
        }
        return singletonObject;
    }
    //------------------------这次 ClassC 得到了 ClassA，开始往下走，一直初始化完ClassC
    //------------------------ClassC 初始化完了，ClassB又继续往下走，一直初始化完ClassB
    //------------------------ClassB 初始化完了，ClassA又继续往下走，一直初始化完ClassA

    //-----------至此，3个类，都顺利的初始化完了
```

- **总结：** 关联依赖，能够成功初始化的原因在于，在递归初始化关联类之前，已经把本类的引用给暴露出去了，这样关联类完全可以只先指向它的引用，而先不用关心这个引用到底有没有执行完全

5.3 构造函数循环依赖

- 构造函数循环依赖肯定是直接报错的
- 因为解决循环依赖的关键在于，提前把本类的引用暴露出去
- 而提前暴露出去的前提是：已经实例化了（`new`了某个构造器了）
- 而构造函数循环依赖，是在实例化本类之前，要得到依赖类，所以本类不可能有引用

5.4 总结

- 任何形成循环依赖的配置方法，如果是在实例化本类之后，在递归实例化依赖类，就不会报错，因为本类的引用已经暴露了，如：注解依赖
- 如果在实例化本类之前，递归实例化依赖类，则会报错，如：构造函数依赖，`xml depends-on` 依赖等

6 总结

- `bean` 初始化的过程细节很多，初期的时候把握一个大概流程即可，不然很有可能看蒙
- 把握整体之后，如果有兴趣或者项目需要，可以慢慢深入

参考

- 深入理解spring生命周期与BeanPostProcessor的实现原理
- 依赖注入之Bean实例化前的准备
- BeanPostProcessor接口
- populateBean实现 依赖注入源码解析
- Spring源码：bean创建（三）:createBeanInstance
- Springboot 源码分析—— 总纲
- Springboot 源码分析—— prepareEnvironment() 解析
- Springboot 源码分析—— refreshContext() 解析
- Springboot 2.1.1.RELEASE