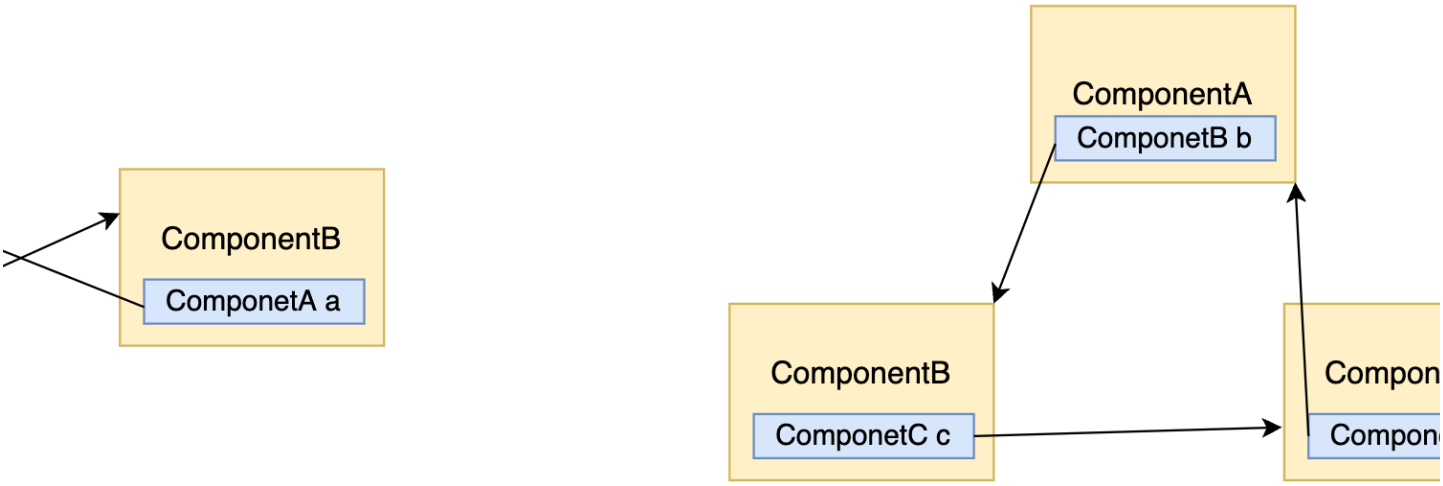
赖与三级缓存

衣赖注入，继续介绍 依赖注入 相关内容，内容包括循环依赖和三级缓存。

ng官方已经不建议循环依赖了，因此尽量**在编码层面避免循环依赖**。

就可能会出现循环依赖问题，如下图所示：



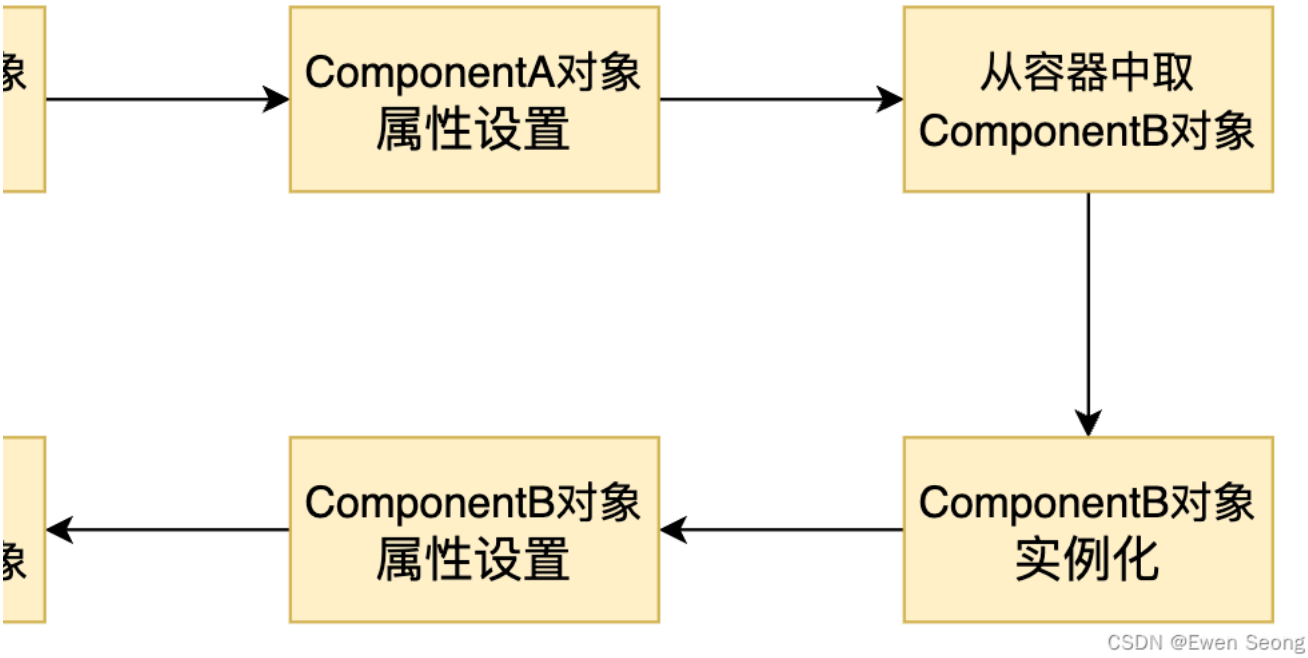
l原理相同，以下以ComponentA和ComponentB为例进行介绍：

onentB;

onentA;

n对象并注入到IOC容器中，而Bean对象的生命周期存在如下流程：准备阶段-> 实例化 ->属性设置->初始化->使用->销毁。

构建流程：



下，Spring创建Bean对象时会陷入无限循环。

例构造一个循环依赖场景：

```
= "com.caltta.ldsdebug.lazy", lazyInit = true)
ion {
```

nponentB设置为懒加载，这样容器启动时就不会将ComponentA和ComponentB注入到IOC容器中。

```
'test begin...");
licationContext applicationContext = new AnnotationConfigApplicationContext(LazyConfiguration.class);
lowCircularReferences方法关闭IOC容器的循环依赖能力,
anFactory)applicationContext.getBeanFactory()).setAllowCircularReferences(false);
A = (ComponentA)applicationContext.getBean("componentA");
'test end!");
```

工厂，并设置禁止循环依赖。

```
actory.UnsatisfiedDependencyException:
e 'componentA': Unsatisfied dependency expressed through field 'componentB';

ingframework.beans.factory.UnsatisfiedDependencyException:Error creating bean with name 'componentB': Unsatisfied dependency expressed throug

ingframework.beans.factory.BeanCurrentlyInCreationException:Error creating bean with name 'componentA': Requested bean is currently in creati
```

etBean("componentA") 执行后，ComponentA会进行完整的Bean生命周期：实例化、属性设置、初始化等流程；在ComponentA对象的属性设置阶段会因需
对象(*关闭循环依赖能力时-不会存储半成本的A对象*)，因此陷入循环。

faultSingletonBeanRegistry中定义了三个属性，作为Spring的三级缓存：

```
cts: bean name to bean instance. */
bject> singletonObjects = new ConcurrentHashMap<>(256);

n objects: bean name to bean instance. */
bject> earlySingletonObjects = new HashMap<>(16);

ories: bean name to ObjectFactory. */
bjectFactory<?>> singletonFactories = new HashMap<>(16);
```

OC容器的单例池，存放创建好的成品对象(完成了实例化、属性设置和初始化等流程，可直接对外使用)；earlySingletonObjects作为二级缓存，存放的是半字: 存放用于构造Bean对象的lambda表达式，本质是Bean对象创建工厂。

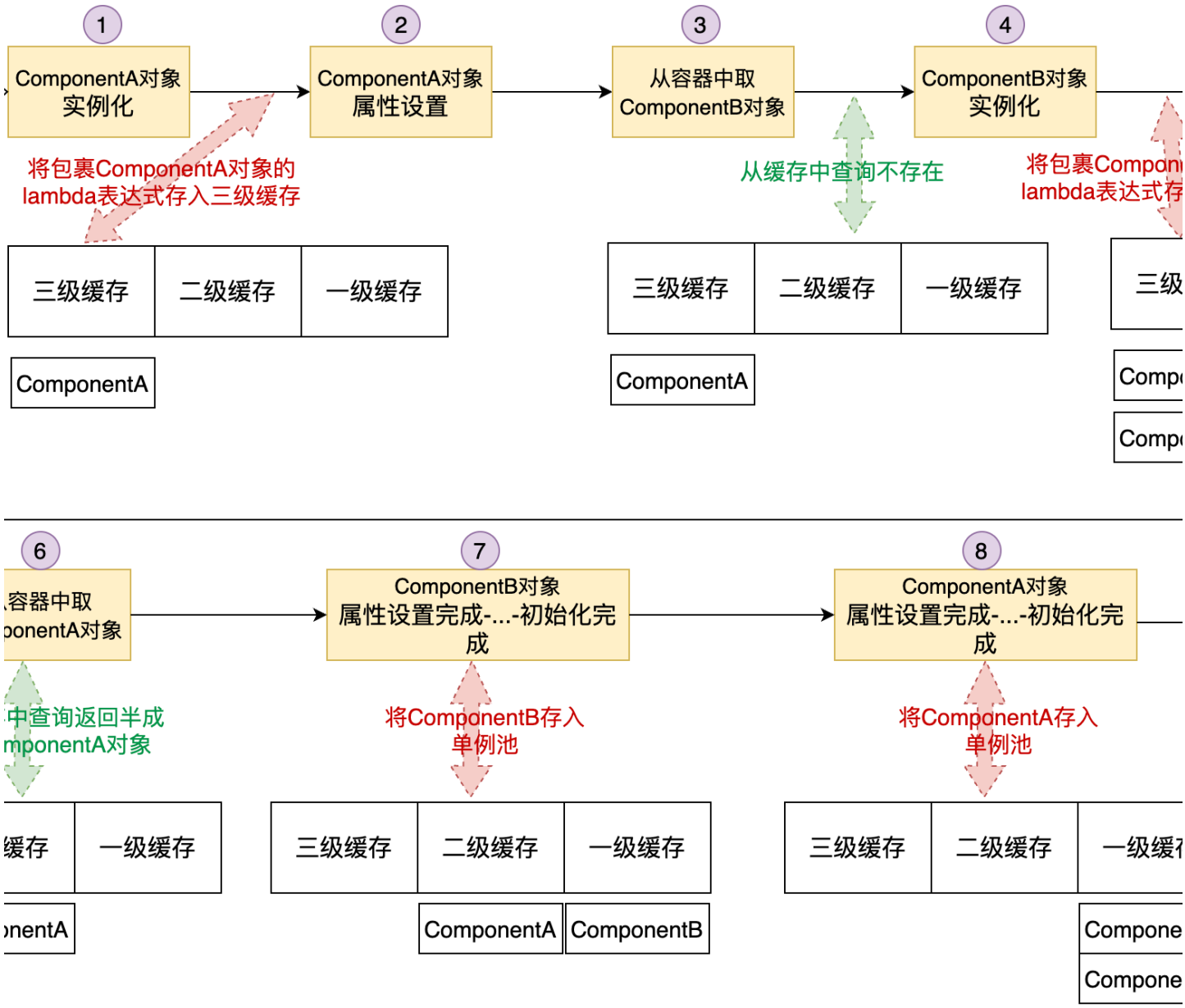
顺序为：一级->二级->三级, 代码如下：

```
String beanName) {
Name, true);


on(String beanName, boolean allowEarlyReference) {
Name获取Bean对象
= this.singletonObjects.get(beanName);
null && isSingletonCurrentlyInCreation(beanName)) {
singletonObjects) {
根据beanName获取Bean对象
= this.earlySingletonObjects.get(beanName);
```

循环依赖场景：

```
) {
'test begin...");
licationContext applicationContext = new AnnotationConfigApplicationContext(LazyConfiguration.class);
:A = (ComponentA)applicationContext.getBean("componentA");
'test end!");
```

会抛出循环依赖异常。因为Spring通过引入三级缓存解决了该类循环依赖问题。案例中Bean对象的实例化和属性设置流程可以表示为：



不再赘述，请参考 Spring系列-2 Bean的生命周期

象时，进入对象ComponentA的Bean生命周期；首先实例化ComponentA对象，此时ComponentA对象未经过属性注入，是一个半成品；然后将包裹Comp

```
an(String beanName, RootBeanDefinition mbd, @Nullable Object[] args) {

oper = createBeanInstance(beanName, mbd, args);
apper.getWrappedInstance();

tons to be able to resolve circular references
ay lifecycle interfaces like BeanFactoryAware
```

entA对象存入包装后存入三级缓存：

```
e, () -> getEarlyBeanReference(beanName, mbd, bean));

actory(String beanName, ObjectFactory<?> singletonFactory) {
etonObjects) {
Objects.containsKey(beanName)) {
actories.put(beanName, singletonFactory);
etonObjects.remove(beanName);
singletons.add(beanName);
```

一个接口)，传入的lambda表达式 `() -> getEarlyBeanReference(beanName, mbd, bean)` 会以匿名类的形式传参；因此从singletonFactories三级缓存中获取
需代理，为代理后的对象)。

介段，对b属性的注入, 堆栈的调用链进入**从容器中获取ComponentB对象**;
列-3 Bean实例化与依赖注入

mponentB的Bean生命周期；完成ComponentB对象的实例化并存入三级缓存，过程同步骤(1)。
介段，进行a属性的注入，尝试从IOC容器获取ComponentA对象；
用三级缓存中的lambda表达式，构建出ComponentA对象(可能是代理后的对象)，然后将该Bean对象存入二级缓存，并从三级缓存中删除。

```
on(String beanName, boolean allowEarlyReference) {
据beanName获取不到Bean对象...

anName获取ObjectFactory并调用getObject获取Bean对象

conFactory = this.singletonFactories.get(beanName);

执行lambda表达式，获取Bean对象
etonFactory.getObject();
```

ne为componentA对应的记录。去除该记录(lambda表达式)执行后，得到一个半成品对象，如果该Bean对象需要被代理，则返回的是代理后的Bean对象。

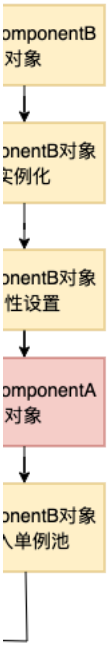
始化操作后，将B对象放入到一级缓存中。

```
an创建完Bean对象后，会调用addSingleton方法
(String beanName, Object singletonObject) {
letonObjects) {

s.put(beanName, singletonObject);

ies.remove(beanName);
Objects.remove(beanName);

letons.add(beanName);
```

设，将从单例池中获取的ComponentB对象注入到b属性中；A完成属性注入和后续的初始化等流程后，将A对象存入一级缓存，并删除二级缓存中的A对象；

前暴露引用，因此需要引入缓存；由于一级缓存是作为单例池存在，因此提前暴露的对象不能放在一级缓存而需要另外开辟缓存。如2.1中例子，本质上如

期如下所示：



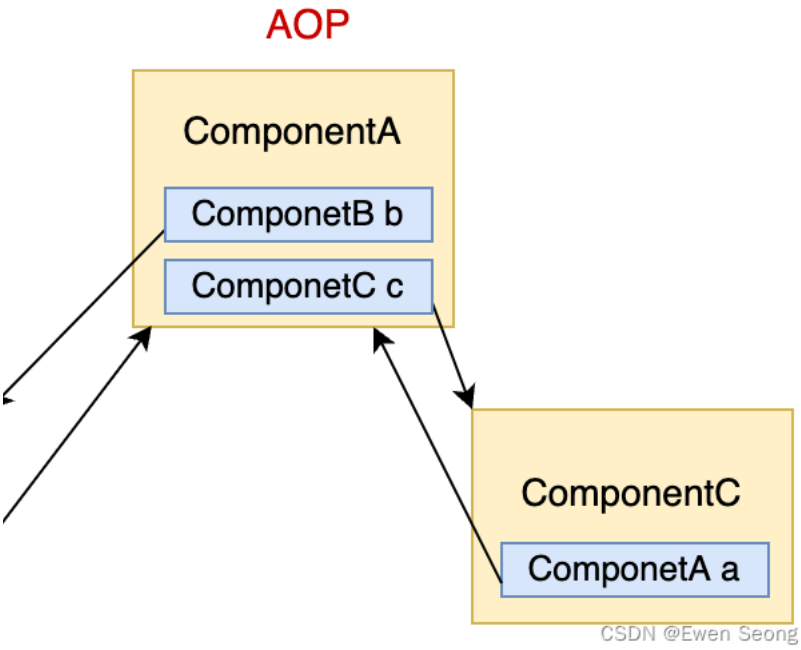
wen Seong

始化阶段的BPP(BeanPostProcessor)执行；因此，上述ComponentB在属性注入阶段得到的ComponentA对象应是原始Bean对象(ComponentA对象的生命
级缓存，该缓存中存放lambda表达式；当执行该表达式时，如果ComponentA需要代理，则可以将ComponentA对象在初始化阶段的AOP操作提前至Com

一个标志(`this.earlyProxyReferences`)使得ComponentA的Bean对象在初始化阶段的AOP过程可以得到已被代理的信息：

```
eference(Object bean, String beanName) {
getCacheKey(bean.getClass(), beanName);
es.put(cacheKey, bean);
操作
sary(bean, beanName, cacheKey);


erInitialization(@Nullable Object bean, String beanName) {

is.getCacheKey(bean.getClass(), beanName);
```

只需要第一级缓存和第三级缓存就可以解决ComponentA和ComponentB的循环依赖，即使ComponentA或ComponentB需要被代理。



omponentB和ComponentC对象存在循环依赖问题；ComponentB和ComponentC对象在属性设置阶段获取ComponentA对象时，两次都会从第三级缓存中
onentA单例的原则。
级缓存配合，共同解决上述问题：第一次通过lambda表达式得到代理对象，然后将代理对象存入二级缓存并删除三级缓存；第二次直接从二级缓存中查询

与依赖注入一文中遗留的两个方法。

```
eanName, pvs)方法(省略try-catch异常):

: bean, @Nullable String beanName, @Nullable PropertyValues pvs) throws Throwable {
is.member;


edArgument(beanName, this.cachedFieldValue);

Value(field, bean, beanName);
```

1) 根据beanName等信息获取待注入的值; (2)通过反射完成属性注入。

```
gument(beanName, this.cachedFieldValue);

ue(field, bean, beanName);
```

```
ue = resolveFieldValue(field, bean, beanName).
 beanName):

/alue(Field field, Object bean, @Nullable String beanName) {
否必须required、bean的class对象构造DependencyDescriptor对象
sc = new DependencyDescriptor(field, this.required);
(bean.getClass());
```

```
ory.resolveDependency方法获取注入的值value【 ⚠主线】
.resolveDependency(desc, beanName, new LinkedHashSet<>(1), beanFactory.getTypeConverter());

) {
```


cy方法：

```
ency(DependencyDescriptor descriptor, @Nullable String requestingBeanName,
g> autowiredBeanNames, @Nullable TypeConverter typeConverter) throws BeansException {


azy注解，则生成一个代理对象返回，否则返回null;
依赖的方案
wireCandidateResolver().getLazyResolutionProxyIfNecessary(
uestingBeanName);


ependency(descriptor, requestingBeanName, autowiredBeanNames, typeConverter);
```

```
ndency(DependencyDescriptor descriptor, @Nullable String beanName,
g> autowiredBeanNames, @Nullable TypeConverter typeConverter) throws BeansException {


息(@Qualifier等)获取Bean集合
chingBeans = findAutowireCandidates(beanName, type, descriptor);

为空场景
cy()) {
riptor)) {
BeanFound(type, descriptor.getResolvableType(), descriptor);
```


findAutowireCandidates(beanName, type, descriptor) 方法获取条件匹配的bean对象，形成集合matchingBeans：通过@Qualifier注解指定了name时，需

行处理：

equired(@Autowired注解时指定的属性)为false, 则返回空；否则抛出NoSuchBeanDefinitionException异常。

据优先级进行筛选：

```
utowireCandidate(Map<String, Object> candidates, DependencyDescriptor descriptor) {
 descriptor.getDependencyType();
使用该Bean对象
 = determinePrimaryCandidate(candidates, requiredType);
 null) {
date;

高的Bean对象（value值越小优先级越高）
 = determineHighestPriorityCandidate(candidates, requiredType);
 null) {
```

idate;

tionException异常

@Primary注解的Bean对象作为优先级最高的对象被返回；否则，找出候选Bean集合的标注了@Priority注解且属性值最小的Bean对象返回；否则，抛出No
主入。

可进一步学习相关知识

系统学习中

本人水平有限，从网上找的资料整合之后做的，请辩证的看待其中内容。

意，也可能有错误，欢迎您指出。 （2）如果您不了解Spring Bean的声明周期，那么您可以看一下文章（https://blog.csdn.net/qq_37171353/article/details/103165108）或者

问题_spring三级缓存如何解决循环依 ...
还没有完全初始化的bean,这样后续其它的bean 在注入Aservice 的bean 时 会发现 Aservice 的bean 已经有了,所以就可以直接使用,不需要在额外创建,这里spring 使用 map (二

解决循环依赖问题?
应的三级缓存如下所示: // 单实例对象注册器publicclassDefaultSingletonBeanRegistryextendsSimpleAliasRegistryimplementsSingletonBeanRegistry{privatestaticfinalintSUPP

只有10%的人才算“真的懂”
个很重要的话题，一方面是因为源码中为了解决循环依赖做了很多处理，另外一方面是因为面试的时候，如果问到Spring中比较高阶的问题，那么循环依赖必定逃不掉。如果你

收藏】
活：循环引用的基本概念、Spring的Bean创建流程、三级缓存解决循环依赖问题、源码调试分析及流程归纳、循环依赖是否一定能被三级缓存解决，最后还给了个特殊的例子并

spring循环依赖和三级缓存
ean的实例,他们的存在主要是为了解决循环依赖问题(即,Spring容器在创建Bean过程中,可能存在BeanA依赖BeanB,同时,B有依赖A的情况(即,循环依赖),为了确保在初始化过程中

循环依赖_spring三级缓存代理对象-C...
循环依赖。 Spring 的 解决方法:三级缓存。 适用场景:使用 Setter 方法的依赖注入。 三级缓存: HashMap<String, Objects> singtletonObjects = new ConcurrentHashMap<>():一

依赖BService，BService依赖AService。 如果不考虑Spring，循环依赖并不是问题，因为对象之间相互依赖是很正常的事情。 但是在spring中，循环依赖就是一个问题了，因为

ring三级缓存如何解决循环依赖-CSDN...
在对象之间存在相互依赖关系,形成一个闭环,导致无法准确地完成对象的创建和初始化;当两个或多个对象彼此之间相互引用,而这种相互引用形成一个循环时,就可能出现循环依

环依赖小结 “三级缓存”意义 Spring 解决循环依赖原理分析 Spring 容器的“三级缓存” 源码解析 常见问题 循环依赖 循环依赖:就是 N 个类循环(嵌套)使用。 简单来说,就是多个 Bea

方式
pring 框架中，Bean 的循环依赖是...Spring Bean 的循环依赖是一个复杂的问题，但通过三级缓存机制，Spring 能够很好地解决这个问题，使得单例对象可以被正确地创建和初

问题：构造器参数循环依赖、setter方法循环依赖和使用三级缓存解决循环依赖。 第一种：构造器参数循环依赖 构造器参数循环依赖是指通过构造器注入构成的循环依赖。这种

ng三级缓存如何解决循环依赖 c...
一个专门的名字,就叫做 earlySingletonObjects,这是 Spring 三级缓存中的二级缓存,这里保存的是刚刚通过反射创建出来的 Bean,这些 Bean 还没有经历过完整生命周期,Bean 的

_spring bean三级缓存
etBean("a"):从各级缓存里找是否存在,第一次创建没有 createBean("a")、doCreateBean("a")、createBeanInstance("a"):实例化bean,将bean放入三级缓存,("a",() -> getEarlyBea

略 **最新发布**
通过三级缓存和AOP代理机制，实现了对循环依赖的高效处理，保证了应用的稳定性和性能。这个解决方案不仅体现了Spring的灵活性，也为Java开发者提供了宝贵的设计和架

存3.1、为什么不用二级缓存而使用三级缓存3.2、为什么不直接提前生成代理对象直接存入二级缓存中，这样不就不需要三级缓存了吗？ 1、循环依赖 这很简单，如下，A类中

环依赖的三级缓存
三级缓存。 Spring启动过程大致如下: 1.创建beanFactory,加载配置文件 2.解析配置文件转化beanDefination,获取到b

就是填充属性。而填充属性之前会判 属性对象是否被当前对象循环依赖，当发现属性对象被循环依赖的时候会进行aop并且生成属性对象的代理对象。 循环依赖是如何形成的 当

依赖为Spring 初始化bean时可能遇到的问题 如：A依赖A；A依赖B，B依赖A等 主要场景 /** 属性填充时产生的 循环依赖 spring能够自动解决 **/ @Component public class ASer

(源码分析））
nObjects）-256：单例，用于保存我们创建完成的bean对象。 二级缓存（earlysingletonObjects)-16：单例，用于保存我们的实例化 但是未进行属性注入及初始化的对象 三级缓

所以， Spring 没有直接在 createBeanInstance 之后直接生成 bean 的早期引用，而是将 bean 的原始对象包装成了一个 ObjectFactory 放到了三级缓存singletonFactories。2、

源码解析 Spring IOC源码解析(1) Spring IOC源码解析(2) Spring拓展点及调用顺序总结 目录1. 图例解释说明1. 循环依赖流程图2. 举例说明上图2. 源码解析1. AbstractBeanFacto

2. 填充属性。3. 初始化。 第一级缓存：singletonObjects，存放经过初始化后的bean。当通过名字获取bean的时候，如果这个名字对应的bean在第一级缓存中，则直接从第一

存中的创建流程 【三级缓存 循环依赖】
叫单例池，存放已经经历了完整生命周期的Bean对象。 第二级缓存：存放早期暴露出来的Bean对象，实例化以后，就把对象放到这个Map中。（Bean可能只经过实例化，属性

引用A, 这样就构成了循环依赖 class A{ B b; } class B{ A a; } 未使用Spring的情况下: 用构造方法的方式无法解决循环依赖, 但是使用set()可以 A a = new A(new B(new A(new B(