1. 不使用SingletonObjects作为一级缓存解决循环依赖的原因：若使用SingletonObjects解决循环依赖，则需要在实例化完成后立即放入缓存中。则未填充属性且未初始化的Bean可能被提前获取到。
2. 三级缓存：SingletonObjects、EarlySingletonObjects、SingletonFactories为3级缓存。分别存放成品、只完成实例化但未填充属性的Bean、工厂对象。
3. getBean过程：
   1. getSingleton：依次从三级缓存中查找对象。一、二级缓存的对象会直接返回。而对于三级缓存，会先把ObjectFactory对象从三级缓存中取出，然后将ObjectFactory生成的对象放入二级缓存。
   2. createBean：
   3. getObjectForBeanInstance：对于FactoryBean尝试factoryBeanObjectCache中获取。获取不到则使用FactoryBean的GetObject方法获取Bean，并把它放入factoryBeanObjectCache中。
4. Bean创建过程
   1. 首先尝试对Bean进行实例化前预解析。预解析使用InstantiationAwarePostProcessors进行实例化前置处理，若处理过程中产生了Bean，则使用BeanPsotProcessors进行初始化后置处理，最后直接返回。
   2. 实例化时会根据参数类型选择合适的构造器，并使用该构造器结合实例化策略进行实例化。实例化策略有JDK实例化和Cglib实例化两种。
   3. 实例化完成的单例Bean会被包装成ObjectFactory并加入3级缓存SingletonFactories中。
   4. 加入3级缓存后，会使用InstantiationAwarePostProcessors进行实例化后置处理。处理过程中如果处理器表示不需要填充属性，则会直接返回Bean。
   5. 填充属性前会使用InstantiationAwarePostProcessors的PostProcessPropertyValues来处理填充属性。@Autowire和@Value的属性填充就是发生在这一步骤中。
   6. 属性填充，基本值直接填充，引用属性BeanReference则根据BeanName去获取Bean。
   7. Bean初始化的步骤是：为感知类注入可感知的属性->BeanPostProcesser对Bean进行初始化前置处理，ApplicationContextAwareProcessor会在这个步骤为ApplicationContextAware类注入ApplicationContext->Initializing的Bean执行afterPropertiesSet方法->初始化方法(BeanDefinition.getInitMethodName)调用 -> BeanPostProcesser对Bean进行初始化后置处理。如果earlyProxyReferences里不存在BeanName，则AOP会在此时生成代理对象。
   8. 注册DisposableBean
   9. 从三级缓存中取出ObjectFactory并使用它的getObject方法来获取经过AOP处理的对象，并将该对象放入二级缓存。
   10. 注册单例，清空二、三级缓存，并把Bean放入一级缓存。
5. ObjectFactory的GetObject：
   1. 使用InstantiationAwarePostProcessors的getEarlyBeanReference方法。该方法将BeanName加入到earlyProxyReferences中表示已完成AOP处理，并对Bean进行AOP处理。
6. 循环依赖解决过程：A、B相互依赖，先创建A。
   1. A实例化完成，A工厂加入三级缓存。A需要注入B，创建B。
   2. B实例化完成，B工厂加入三级缓存。
   3. B需要A，A工厂从三级缓存移除并被标记到earlyProxyReference中，并将生产出的代理产品A加入二级缓存。
   4. B从二级缓存中得到A’的半成品，初始化完成后使用B工厂完成代理并被标记到earlyProxyReference中，最后移入二级缓存。
   5. B单例注册到一级缓存，同时清空二三级缓存。
   6. 返回A属性填充处，A得到B的成品。
   7. A完成初始化，A调用getSingleton方法从二级缓存中得到A
   8. 成品A被注册到一级缓存，同时清空二三级缓存。