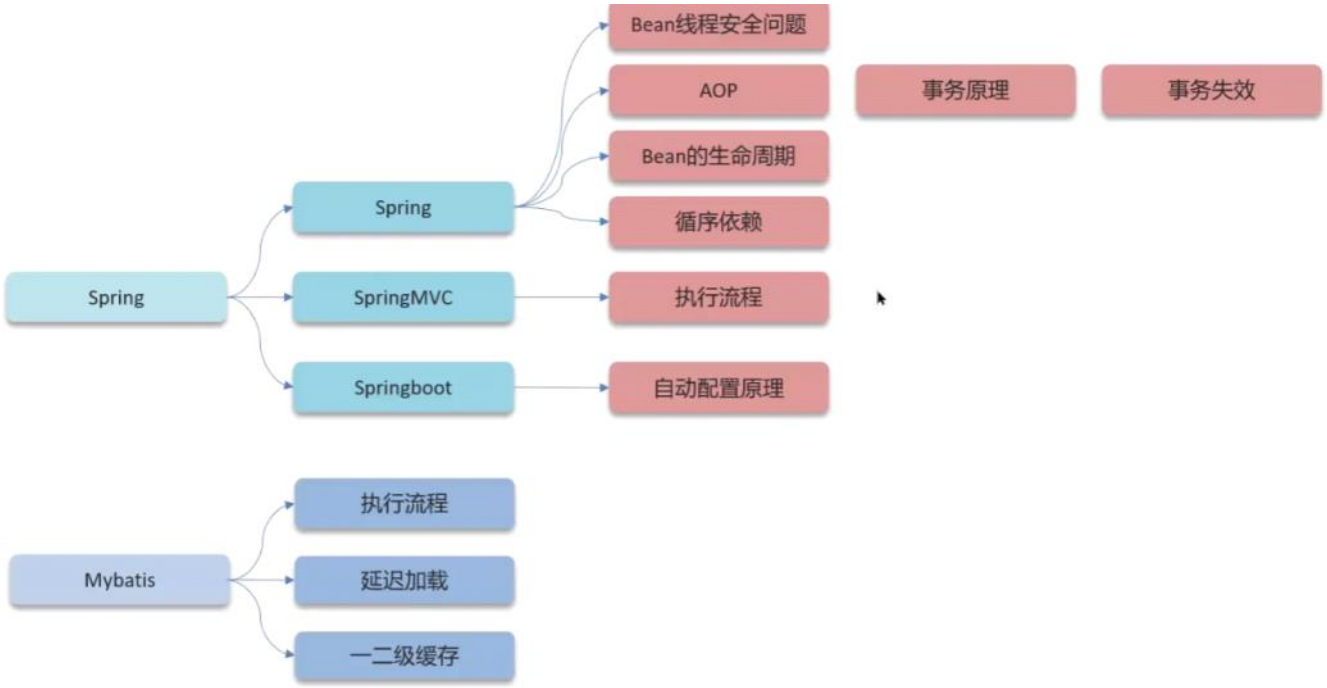
Spring

2024年4月5日 17:48



Spring框架中的单例bean是线程安全的吗?

Spring框架中的bean是单例的吗?

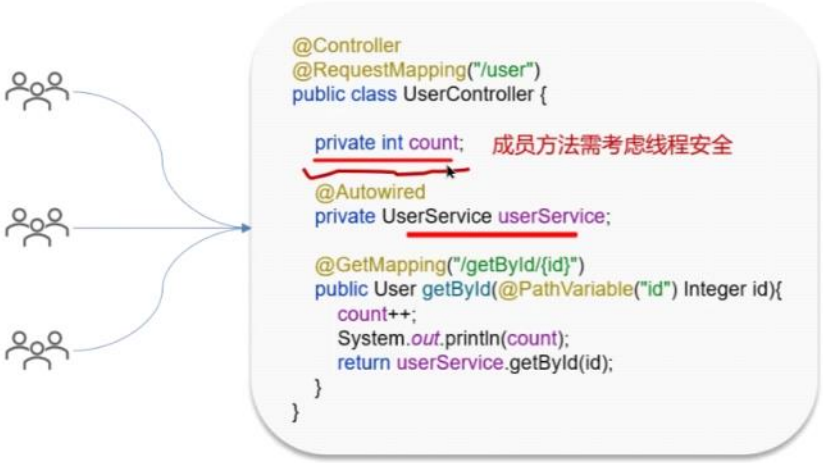
```
@Service
@Scope("singleton")
public class UserServiceImpl implements UserService {


}
```

- singleton : bean在每个Spring IOC容器中只有一个实例。
- prototype：一个bean的定义可以有多个实例。

不是线程安全的

Spring框架中的单例bean是线程安全的吗?



服务器中的代码片段

Spring bean并没有可变的状态(比如Service类和DAO类)，所以在某种程度上说Spring的单例bean是线程安全的。

Spring框架中的单例bean是线程安全的吗?

不是线程安全的

Spring框架中有一个@Scope注解，默认的值就是singleton，单例的。

因为一般在spring的bean的中都是注入无状态的对象，没有线程安全问题，如果在bean中定义了可修改的成员变量，是要考虑线程安全问题的，可以使用多例或者加锁来解决

什么是AOP，你们项目中有没有使用到AOP

对AOP的理解

有没有真的用过aop

AOP称为面向切面编程，用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块，这个模块被命名为“切面”（Aspect），减少系统中的重复代码，降低了模块间的耦合度，同时提高了系统的可维护性。

常见的AOP使用场景:

- 记录操作日志
- 缓存处理
- Spring中内置的事务处理

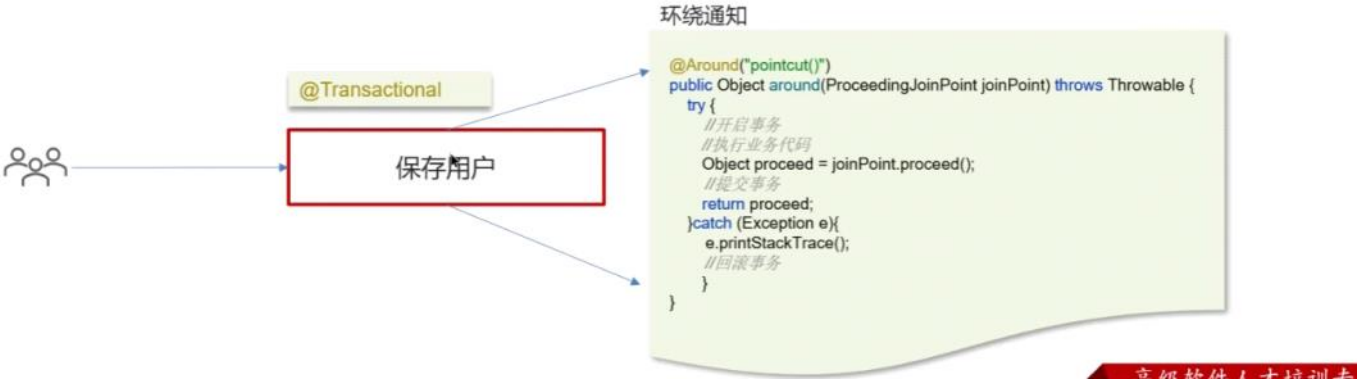
什么是AOP,你们项目中有没有使用到AOP

记录操作日志思路

Spring中的事务是如何实现的

Spring支持编程式事务管理和声明式事务管理两种方式。

- 编程式事务控制：需使用TransactionTemplate来进行实现，对业务代码有侵入性，项目中很少使用
- 声明式事务管理：声明式事务管理建立在AOP之上的。其本质是通过AOP功能，对方法前后进行拦截，将事务处理的功能编织到拦截的方法中，也就是在目标方法开始之前加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。



什么是AOP

面向切面编程，用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取公共模块复用，降低耦合

你们项目中有没有使用到AOP

记录操作日志，缓存，spring实现的事务

核心是：使用aop中的环绕通知+切点表达式（找到要记录日志的方法），通过环绕通知的参数获取请求方法的参数（类、方法、注解、请求方式等），获取到这些参数以后，保存到数据库

Spring中的事务是如何实现的

其本质是通过AOP功能，对方法前后进行拦截，在执行方法之前开启事务，在执行完目标方法之后根据执行情况提交或者回滚事务。

Spring中事务失效的场景有哪些

对spring框架的深入理解、复杂业务的编码经验

- 异常捕获处理
- 抛出检查异常
- 非public方法

Spring中事务失效的场景?

情况一：异常捕获处理

```
@Transactional
public void update(Integer from, Integer to, Double money) {
    try {
        //转账的用户不能为空
        Account fromAccount = accountDao.selectById(from);
        //判断用户的钱是否够转账
        if (fromAccount.getMoney() - money >= 0) {
        fromAccount.setMoney(fromAccount.getMoney() - money);
        accountDao.updateById(fromAccount);

        //异常
        int a = 1/0;

        //被转账的用户
        Account toAccount = accountDao.selectById(to);
        toAccount.setMoney(toAccount.getMoney() + money);
        accountDao.updateById(toAccount);
      }
    } catch (Exception e) {
       e.printStackTrace();
    }
}
```

原因

事务通知只有捉到了目标抛出的异常，才能进行后续的回滚处理，如果目标自己处理掉异常，事务通知无法知悉

解决

在catch块添加throw new RuntimeException(e)抛出

Spring中事务失效的场景?

情况二：抛出检查异常

```
@Transactional
public void update(Integer from, Integer to, Double money) throws FileNotFoundException {
      //转账的用户不能为空
      Account fromAccount = accountDao.selectById(from);
      //判断用户的钱是否够转账
      if (fromAccount.getMoney() - money >= 0) {
       fromAccount.setMoney(fromAccount.getMoney() - money);
       accountDao.updateById(fromAccount);
      //读取文件
      new FileInputStream("dddd");
      //被转账的用户
       Account toAccount = accountDao.selectById(to);
      toAccount.setMoney(toAccount.getMoney() + money);
      accountDao.updateById(toAccount);
    }
}
```

原因

Spring 默认只会回滚非检查异常

解决

配置rollbackFor属性

@Transactional(rollbackFor=Exception.class)

Spring中事务失效的场景?

情况三：非public方法导致的事务失效

```
@Transactional(rollbackFor = Exception.class)
void update(Integer from, Integer to, Double money) throws FileNotFoundException {
      //转账的用户不能为空
      Account fromAccount = accountDao.selectById(from);
      //判断用户的钱是否够转账
      if (fromAccount.getMoney() - money >= 0) {
       fromAccount.setMoney(fromAccount.getMoney() - money);
       accountDao.updateById(fromAccount);

      //读取文件
      new FileInputStream("dddd");

      //被转账的用户
      Account toAccount = accountDao.selectById(to);
       toAccount.setMoney(toAccount.getMoney() + money);
       accountDao.updateById(toAccount);
   }
}
```

原因

Spring 为方法创建代理、添加事务通知、前提条件都是该方法是 public 的

解决

改为 public 方法

Spring中事务失效的场景有哪些

① 异常捕获处理，自己处理了异常，没有抛出，解决：手动抛出

② 抛出检查异常，配置rollbackFor属性为Exception

③ 非public方法导致的事务失效，改为public

Spring的bean的生命周期

Spring容器是如何管理和创建bean实例

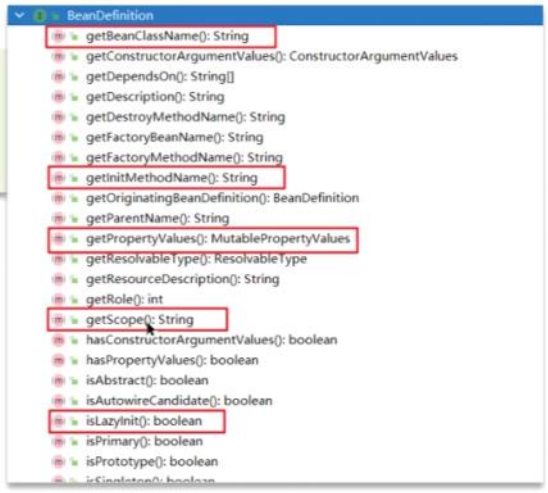
方便调试和解决问题

Bean的流程 → 代码验证

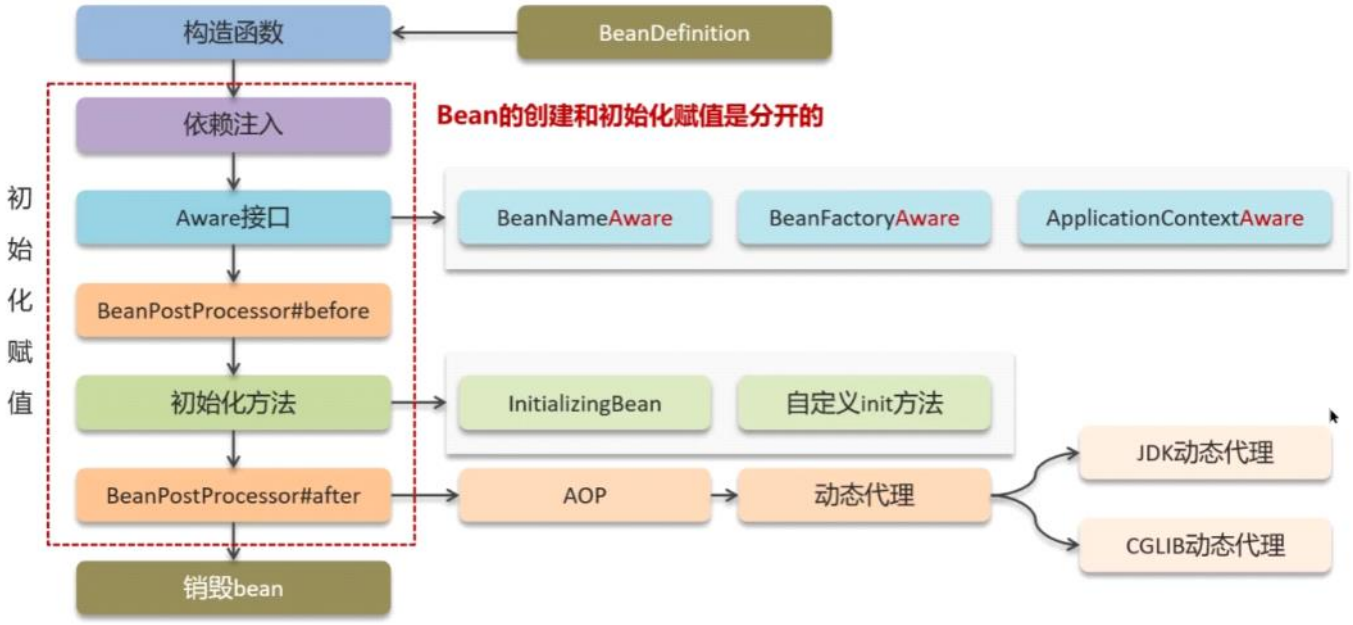
BeanDefinition

Spring容器在进行实例化时，会将xml配置的<bean>的信息封装成一个BeanDefinition对象，Spring根据BeanDefinition来创建Bean对象，里面有很多的属性用来描述Bean

```
<bean id="userDao" class="com.itheima.dao.impl.UserDaoImpl" lazy-init="true"/>
<bean id="userService" class="com.itheima.service.UserServiceImpl" scope="singleton">
    <property name="userDao" ref="userDao"></property>
</bean>
```



- beanClassName：bean 的类名
- initMethodName：初始化方法名称
- properryValues：bean 的属性值
- scope：作用域
- lazyInit：延迟初始化

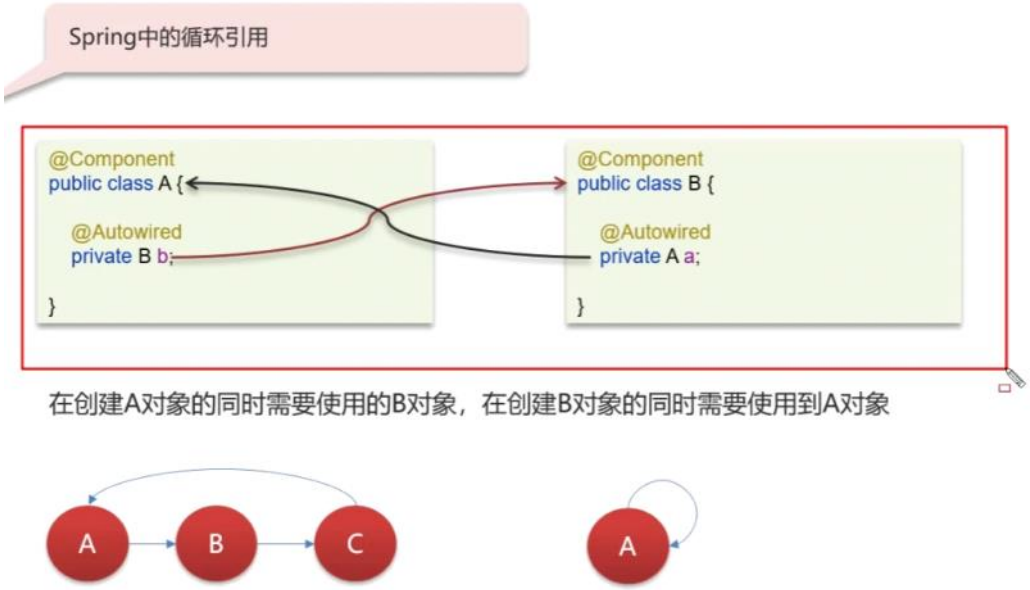


1. 创建 Bean 的实例：Bean 容器首先会找到配置文件中的 Bean 定义，然后使用 Java 反射 API 来创建 Bean 的实例。

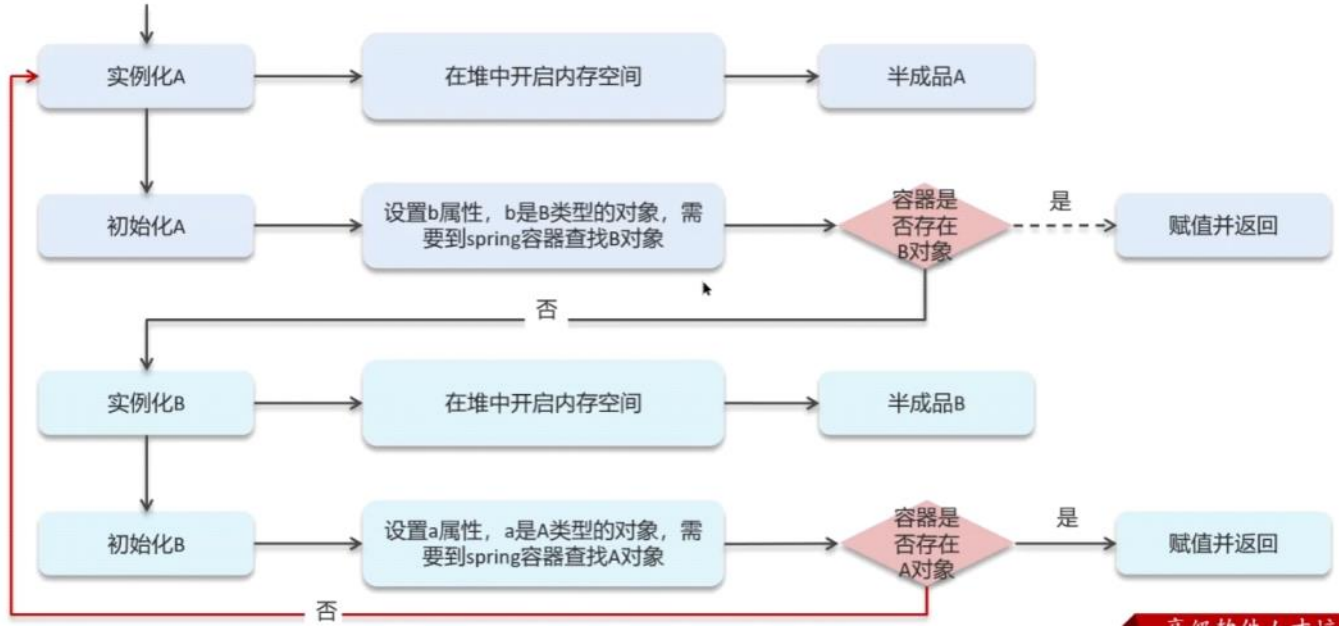
2. **Bean 属性赋值/填充：**为 Bean 设置相关属性和依赖，例如@Autowired 等注解注入的对象、@Value 注入的值、setter方法或构造函数注入依赖和值、@Resource注入的各种资源。

3. **Bean 初始化：**
   - 如果 Bean 实现了 BeanNameAware 接口，调用 setBeanName()方法，传入 Bean 的名字。
   - 如果 Bean 实现了 BeanClassLoaderAware 接口，调用 setBeanClassLoader()方法，传入 ClassLoader对象的实例。
   - 如果 Bean 实现了 BeanFactoryAware 接口，调用 setBeanFactory()方法，传入 BeanFactory对象的实例。
   - 与上面的类似，如果实现了其他 *.Aware接口，就调用相应的方法。
   - 如果有和加载这个 Bean 的 Spring 容器相关的 BeanPostProcessor 对象，执行 postProcessBeforeInitialization() 方法
   - 如果 Bean 实现了InitializingBean接口，执行afterPropertiesSet()方法。
   - 如果 Bean 在配置文件中的定义包含 init-method 属性，执行指定的方法。
   - 如果有和加载这个 Bean 的 Spring 容器相关的 BeanPostProcessor 对象，执行 postProcessAfterInitialization() 方法。

4. **销毁 Bean**：销毁并不是说要立马把 Bean 给销毁掉，而是把 Bean 的销毁方法先记录下来，将来需要销毁 Bean 或者销毁容器的时候，就调用这些方法去释放 Bean 所持有的资源。
   - 如果 Bean 实现了 DisposableBean 接口，执行 destroy() 方法。
   - 如果 Bean 在配置文件中的定义包含 destroy-method 属性，执行指定的 Bean 销毁方法。或者，也可以直接通过@PreDestroy 注解标记 Bean 销毁之前执行的方法。



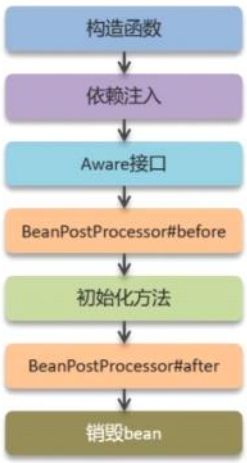
什么是Spring的循环依赖?



三级缓存解决循环依赖

Spring解决循环依赖是通过三级缓存，对应的三级缓存如下所示：

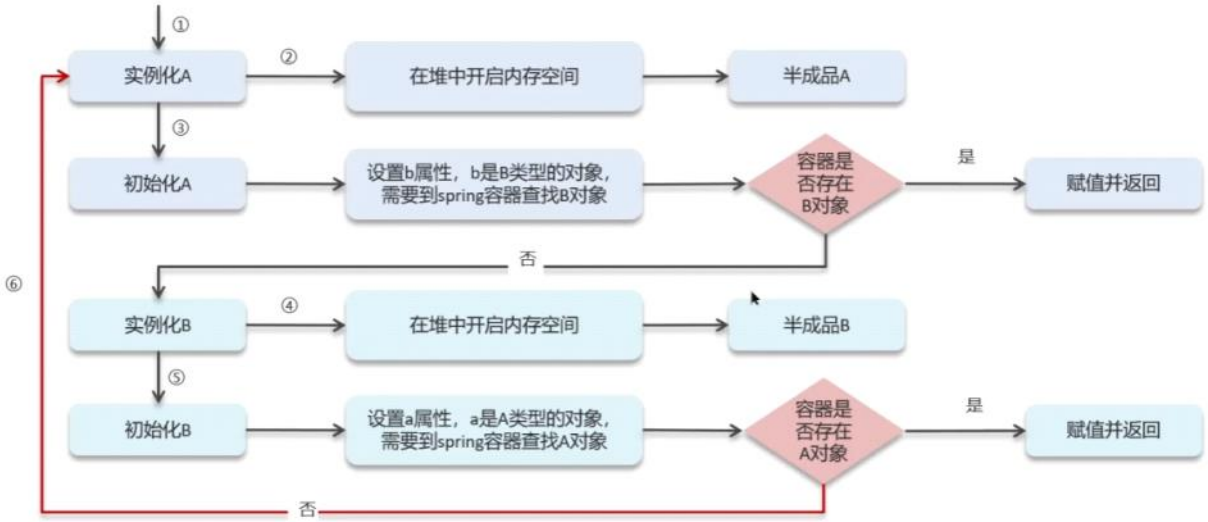
```
//单实例对象注册器
public class DefaultSingletonBeanRegistry extends SimpleAliasRegistry implements SingletonBeanRegistry {
    private static final int SUPPRESSED_EXCEPTIONS_LIMIT = 100;
    private final Map<String, Object> singletonObjects = new ConcurrentHashMap(256);          一级缓存
    private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap(16);         三级缓存
    private final Map<String, Object> earlySingletonObjects = new ConcurrentHashMap(16);      二级缓存
}
```



缓存名称	源码名称	作用
一级缓存	singletonObjects	单例池，缓存已经经历了完整的生命周期，已经初始化完成的bean对象
二级缓存	earlySingletonObjects	缓存早期的bean对象（生命周期还没走完）
三级缓存	singletonFactories	缓存的是ObjectFactory，表示对象工厂，用来创建某个对象的

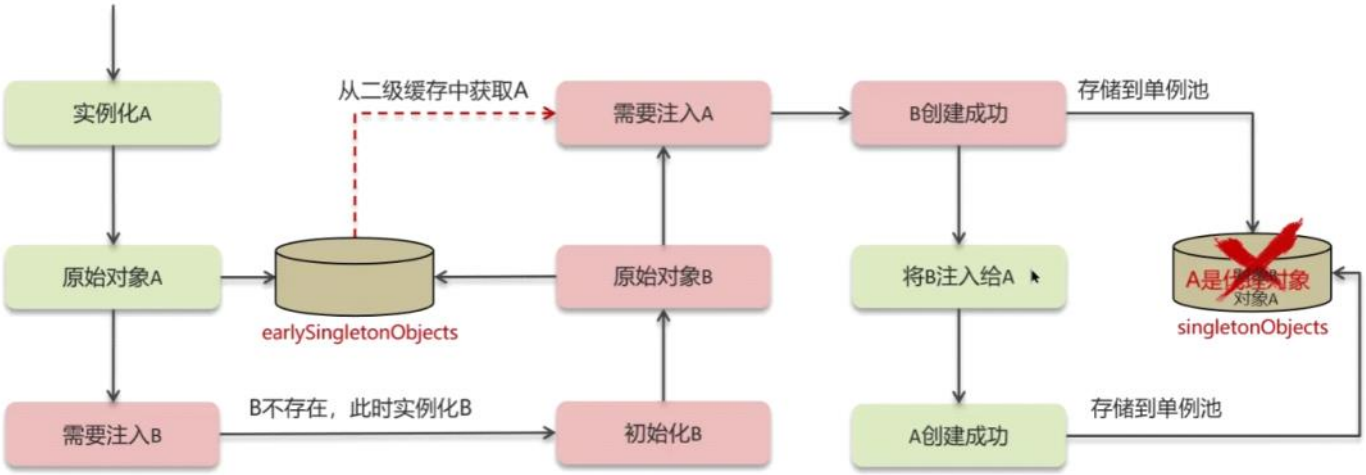
三级缓存解决循环依赖

一级缓存作用：限制bean在beanFactory中只存一份，即实现singleton scope，解决不了循环依赖

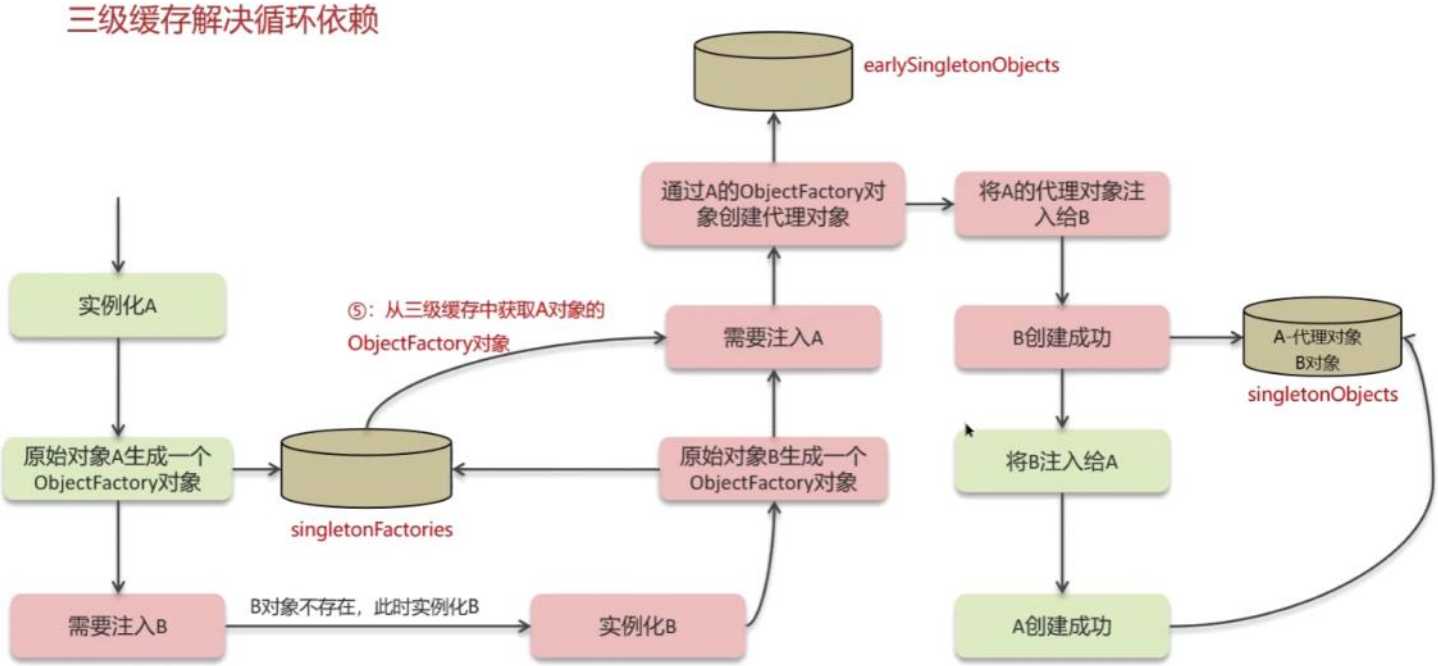


三级缓存解决循环依赖

如果要想打破循环依赖，就需要一个中间人的参与，这个中间人就是二级缓存。



三级缓存解决循环依赖



构造方法出现了循环依赖怎么解决?

```java
@Component
public class A {

    // B成员变量
    private B b;

    public A(B b){
        System.out.println("A的构造方法执行了...");
        this.b = b ;
    }
}
```

```java
@Component
public class B {

    // A成员变量
    private A a;

    public B(A a){
        System.out.println("B的构造方法执行了...");
        this.a = a ;
    }
}
```

报错信息：

```
Is there an unresolvable circular reference?
```

解决：

```java
public A(@Lazy B b){
    System.out.println("A的构造方法执行了...");
    this.b = b ;
}
```

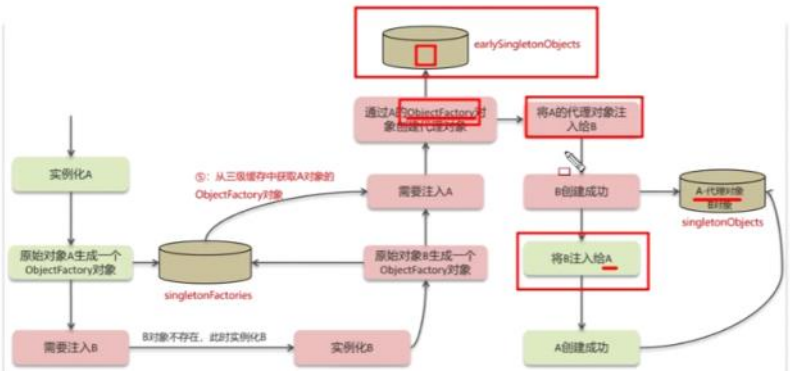
Spring中的循环引用

- 循环依赖：循环依赖其实就是循环引用,也就是两个或两个以上的bean互相持有对方,最终形成闭环。比如A依赖于B,B依赖于A
- 循环依赖在spring中是允许存在，spring框架依据三级缓存已经解决了大部分的循环依赖

① 一级缓存：单例池，缓存已经经历了完整的生命周期，已经初始化完成的bean对象

② 二级缓存：缓存早期的bean对象（生命周期还没走完）

③ 三级缓存：缓存的是ObjectFactory，表示对象工厂，用来创建某个对象的



构造方法出现了循环依赖怎么解决?

A依赖于B，B依赖于A，注入的方式是构造函数

原因：由于bean的生命周期中构造函数是第一个执行的，spring框架并不能解决构造函数的的依赖注入

解决方案：使用@Lazy进行懒加载，什么时候需要对象再进行bean对象的创建

```
public A(@Lazy B b){
    System.out.println("A的构造方法执行了...");
    this.b = b ;
}
```