01-Spring面试专题

讲师：邓澎波

# **Spring面试专题**

# **1.Spring应该很熟悉吧？来介绍下你的Spring的理解**

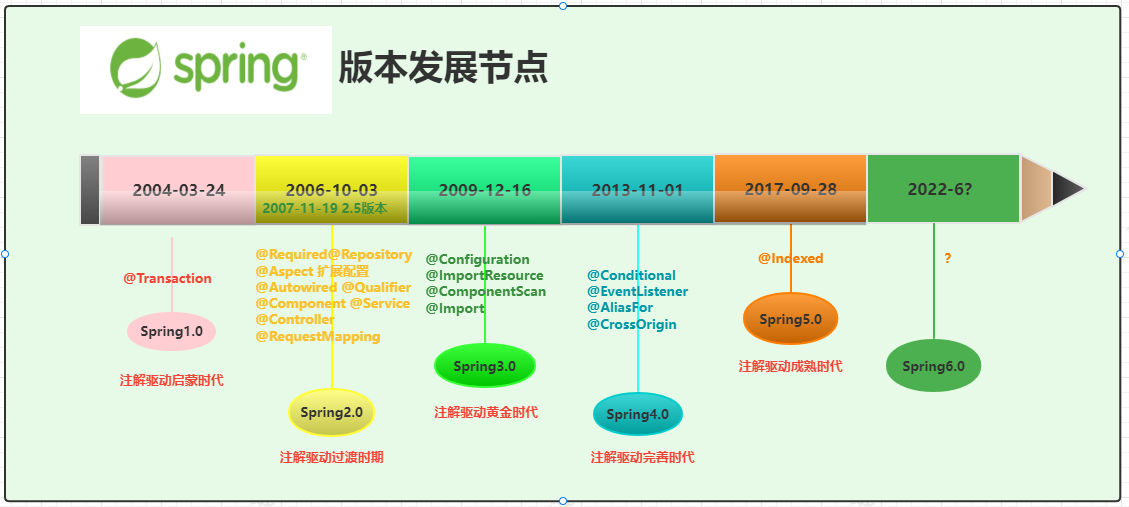
  有些同学可能会抢答，不熟悉!!!



  好了，不开玩笑，面对这个问题我们应该怎么来回答呢？我们给大家梳理这个几个维度来回答

## **1.1 Spring的发展历程**

  先介绍Spring是怎么来的，发展中有哪些核心的节点，当前的最新版本是什么等



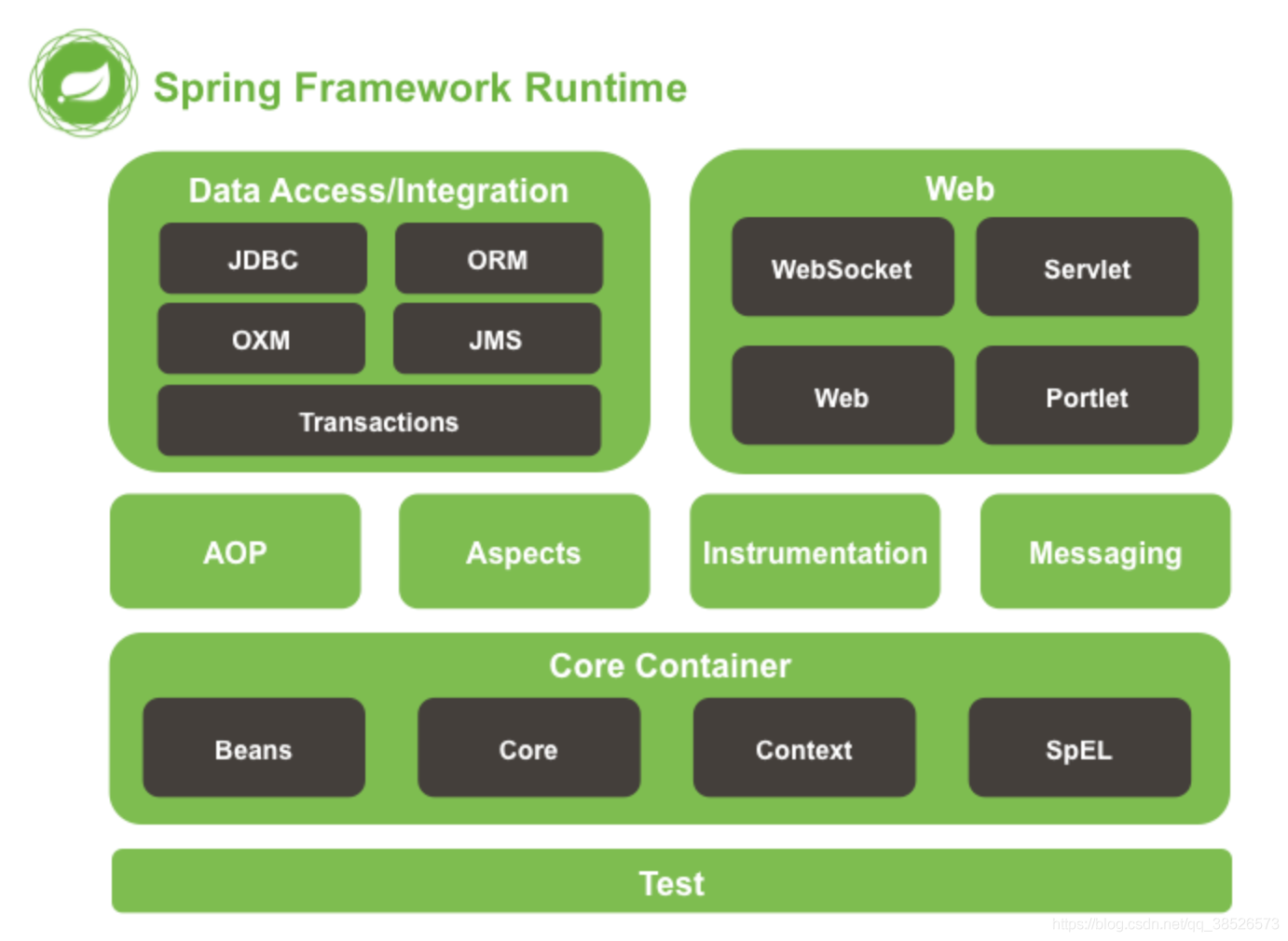
 通过上图可以比较清晰的看到Spring的各个时间版本对应的时间节点了。也就是Spring从之前单纯的xml的配置方式，到现在的完全基于注解的编程方式发展。

## **1.2 Spring的组成**

  Spring是一个轻量级的IoC和AOP容器框架。是为Java应用程序提供基础性服务的一套框架，目的是用于简化企业应用程序的开发，它使得开发者只需要关心业务需求。常见的配置方式有三种：基于XML的配置、基于注解的配置、基于Java的配置.

主要由以下几个模块组成：

* Spring Core：核心类库，提供IOC服务；
* Spring Context：提供框架式的Bean访问方式，以及企业级功能（JNDI、定时任务等）；
* Spring AOP：AOP服务；
* Spring DAO：对JDBC的抽象，简化了数据访问异常的处理；
* Spring ORM：对现有的ORM框架的支持；
* Spring Web：提供了基本的面向Web的综合特性，例如多方文件上传；
* Spring MVC：提供面向Web应用的Model-View-Controller实现。



## **1.3 Spring的好处**

| **序号** | **好处** | **说明** |
| --- | --- | --- |
| 1 | 轻量 | Spring 是轻量的，基本的版本大约2MB。 |
| 2 | 控制反转 | Spring通过控制反转实现了松散耦合，对象们给出它们的依赖，``而不是创建或查找依赖的对象们。 |
| 3 | 面向切面编程(AOP) | Spring支持面向切面的编程，并且把应用业务逻辑和系统服务分开。 |
| 4 | 容器 | Spring 包含并管理应用中对象的生命周期和配置。 |
| 5 | MVC框架 | Spring的WEB框架是个精心设计的框架，是Web框架的一个很好的替代品。 |
| 6 | 事务管理 | Spring 提供一个持续的事务管理接口，``可以扩展到上至本地事务下至全局事务（JTA）。 |
| 7 | 异常处理 | Spring 提供方便的API把具体技术相关的异常 ``(比如由JDBC，Hibernate or JDO抛出的)转化为一致的unchecked 异常。 |
| 8 | 最重要的 | 用的人多！！！ |

# **2.Spring框架中用到了哪些设计模式**

## **2.1 单例模式**

  单例模式应该是大家印象最深的一种设计模式了。在Spring中最明显的使用场景是在配置文件中配置注册bean对象的时候**设置scope的值为singleton** 。

<?xml version="1.0" encoding="UTF-8"?><beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean class="com.dpb.pojo.User" id="user" scope="singleton">

<property name="name" value="波波烤鸭"></property>

</bean></beans>

## **2.2 原型模式**

  原型模式也叫克隆模式，Spring中该模式使用的很明显，和单例一样在bean标签中设置scope的属性prototype即表示该bean以克隆的方式生成

<?xml version="1.0" encoding="UTF-8"?><beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean class="com.dpb.pojo.User" id="user" scope="prototype">

<property name="name" value="波波烤鸭"></property>

</bean></beans>

## **2.3 模板模式**

  模板模式的核心是父类定义好流程，然后将流程中需要子类实现的方法就抽象话留给子类实现，Spring中的JdbcTemplate就是这样的实现。我们知道jdbc的步骤是固定

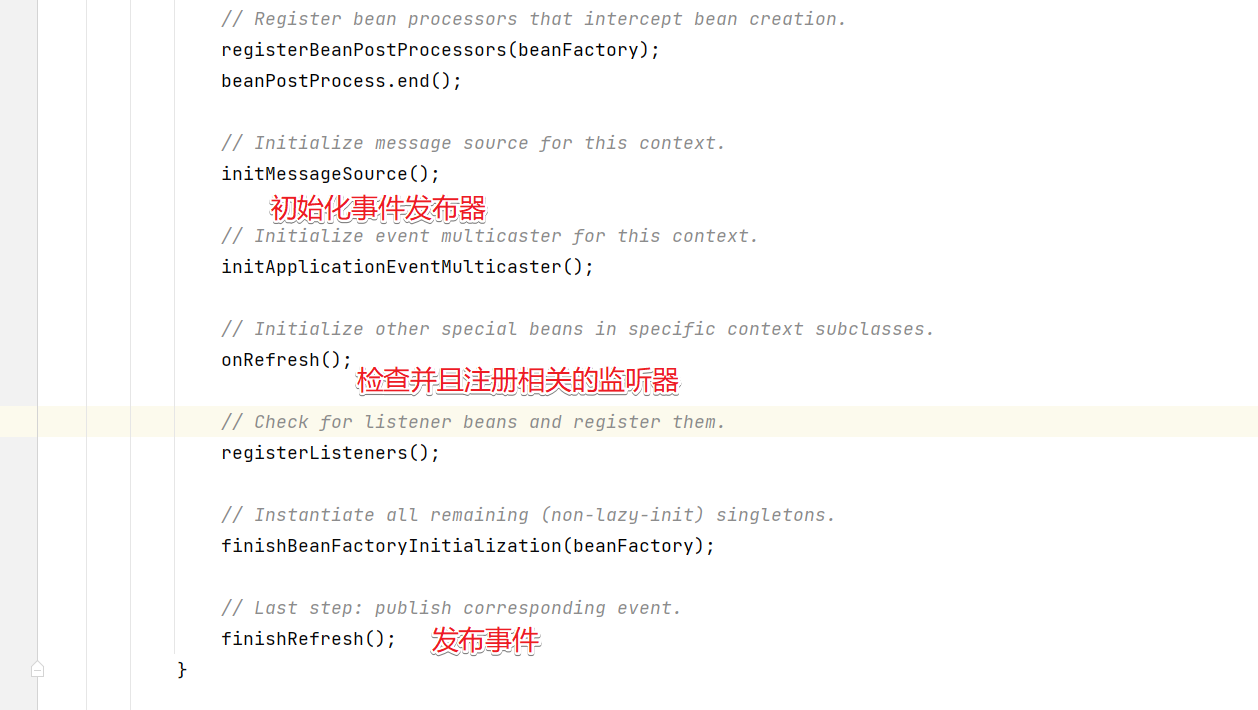
* 加载驱动,
* 获取连接通道,
* 构建sql语句.
* 执行sql语句,
* 关闭资源

  在这些步骤中第3步和第四步是不确定的,所以就留给客户实现，而我们实际使用JdbcTemplate的时候也确实是只需要构建SQL就可以了.这就是典型的模板模式。我们以query方法为例来看下JdbcTemplate中的代码.



## **2.4 观察者模式**

  观察者模式定义的是对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。使用比较场景是在监听器中而spring中Observer模式常用的地方也是listener的实现。如ApplicationListener.



## **2.5 工厂模式**

**简单工厂模式**：

  简单工厂模式就是通过工厂根据传递进来的参数决定产生哪个对象。Spring中我们通过getBean方法获取对象的时候根据id或者name获取就是简单工厂模式了。

<?xml version="1.0" encoding="UTF-8"?><beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.3.xsd">

<context:annotation-config/>

<bean class="com.dpb.pojo.User" id="user" >

<property name="name" value="波波烤鸭"></property>

</bean></beans>

**工厂方法模式**：

  在Spring中我们一般是将Bean的实例化直接交给容器去管理的，实现了使用和创建的分离，这时容器直接管理对象，还有种情况是，bean的创建过程我们交给一个工厂去实现，而Spring容器管理这个工厂。这个就是我们讲的工厂模式，在Spring中有两种实现一种是静态工厂方法模式，一种是动态工厂方法模式。以静态工厂来演示

/\*\*

\* User 工厂类

\* @author dpb[波波烤鸭]

\*

\*/public class UserFactory {

/\*\*

\* 必须是static方法

\* @return

\*/

public static UserBean getInstance(){

return new UserBean();

}

}

application.xml文件中注册

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<!-- 静态工厂方式配置 配置静态工厂及方法 -->

<bean class="com.dpb.factory.UserFactory" factory-method="getInstance" id="user2"/></beans>

## **2.6 适配器模式**

  将一个类的接口转换成客户希望的另外一个接口。使得原本由于接口不兼容而不能一起工作的那些类可以在一起工作。这就是适配器模式。在Spring中在AOP实现中的Advice和interceptor之间的转换就是通过适配器模式实现的。

class MethodBeforeAdviceAdapter implements AdvisorAdapter, Serializable {

@Override

public boolean supportsAdvice(Advice advice) {

return (advice instanceof MethodBeforeAdvice);

}

@Override

public MethodInterceptor getInterceptor(Advisor advisor) {

MethodBeforeAdvice advice = (MethodBeforeAdvice) advisor.getAdvice();

// 通知类型匹配对应的拦截器

return new MethodBeforeAdviceInterceptor(advice);

}

}

## **2.7 装饰者模式**

  装饰者模式又称为包装模式(Wrapper),作用是用来动态的为一个对象增加新的功能。装饰模式是一种用于代替继承的技术，无须通过继承增加子类就能扩展对象的新功能。使用对象的关联关系代替继承关系，更加灵活，同时避免类型体系的快速膨胀。  
  spring中用到的包装器模式在类名上有两种表现：一种是类名中含有Wrapper，另一种是类名中含有Decorator。基本上都是动态地给一个对象添加一些额外的职责。  
  具体的使用在Spring session框架中的SessionRepositoryRequestWrapper使用包装模式对原生的request的功能进行增强，可以将session中的数据和分布式数据库进行同步，这样即使当前tomcat崩溃，session中的数据也不会丢失。

<dependency>

<groupId>org.springframework.session</groupId>

<artifactId>spring-session</artifactId>

<version>1.3.1.RELEASE</version></dependency>

## **2.8 代理模式**

  代理模式应该是大家非常熟悉的设计模式了，在Spring中AOP的实现中代理模式使用的很彻底.

## **2.9 策略模式**

  策略模式对应于解决某一个问题的一个算法族，允许用户从该算法族中任选一个算法解决某一问题，同时可以方便的更换算法或者增加新的算法。并且由客户端决定调用哪个算法，spring中在实例化对象的时候用到Strategy模式。XmlBeanDefinitionReader,PropertiesBeanDefinitionReader

## **2.10 责任链默认**

AOP中的拦截器链

## **2.11 委托者模式**

DelegatingFilterProxy，整合Shiro，SpringSecurity的时候都有用到。

.....

# **3.Autowired和Resource关键字的区别？**

  这是一个相对比较简单的问题，@Resource和@Autowired都是做bean的注入时使用，其实@Resource并不是Spring的注解，它的包是javax.annotation.Resource，需要导入，但是Spring支持该注解的注入。

## **3.1 共同点**

  两者都可以写在字段和setter方法上。两者如果都写在字段上，那么就不需要再写setter方法.

## **3.2 不同点**

**@Autowired**

  @Autowired为Spring提供的注解，需要导入org.springframework.beans.factory.annotation.Autowired;只按照byType注入。

public class TestServiceImpl {

// 下面两种@Autowired只要使用一种即可

@Autowired

private UserDao userDao; // 用于字段上

@Autowired

public void setUserDao(UserDao userDao) { // 用于属性的方法上

this.userDao = userDao;

}

}

@Autowired注解是按照类型（byType）装配依赖对象，默认情况下它要求依赖对象必须存在，如果允许null值，可以设置它的required属性为false。如果我们想使用按照名称（byName）来装配，可以结合@Qualififier注解一起使用。如下：

public class TestServiceImpl {

@Autowired

@Qualifier("userDao")

private UserDao userDao; }

**@Resource**

  @Resource默认按照ByName自动注入，由J2EE提供，需要导入包javax.annotation.Resource。@Resource有两个重要的属性：name和type，而Spring将@Resource注解的name属性解析为bean的名字，而type属性则解析为bean的类型。所以，如果使用name属性，则使用byName的自动注入策略，而使用type属性时则使用byType自动注入策略。如果既不制定name也不制定type属性，这时将通过反射机制使用byName自动注入策略.

public class TestServiceImpl {

// 下面两种@Resource只要使用一种即可

@Resource(name="userDao")

private UserDao userDao; // 用于字段上

@Resource(name="userDao")

public void setUserDao(UserDao userDao) { // 用于属性的setter方法上

this.userDao = userDao;

}

}

@Resource装配顺序：

1. 如果同时指定了name和type，则从Spring上下文中找到唯一匹配的bean进行装配，找不到则抛出异常。
2. 如果指定了name，则从上下文中查找名称（id）匹配的bean进行装配，找不到则抛出异常。
3. 如果指定了type，则从上下文中找到类似匹配的唯一bean进行装配，找不到或是找到多个，都会抛出异常。
4. 如果既没有指定name，又没有指定type，则自动按照byName方式进行装配；如果没有匹配，则回退为一个原始类型进行匹配，如果匹配则自动装配。

@Resource的作用相当于@Autowired，只不过@Autowired按照byType自动注入。

# **4.Spring中常用的注解有哪些，重点介绍几个**

@Controller @Service @RestController @RequestBody,@Index @Import等

@Indexd提升 @ComponentScan的效率

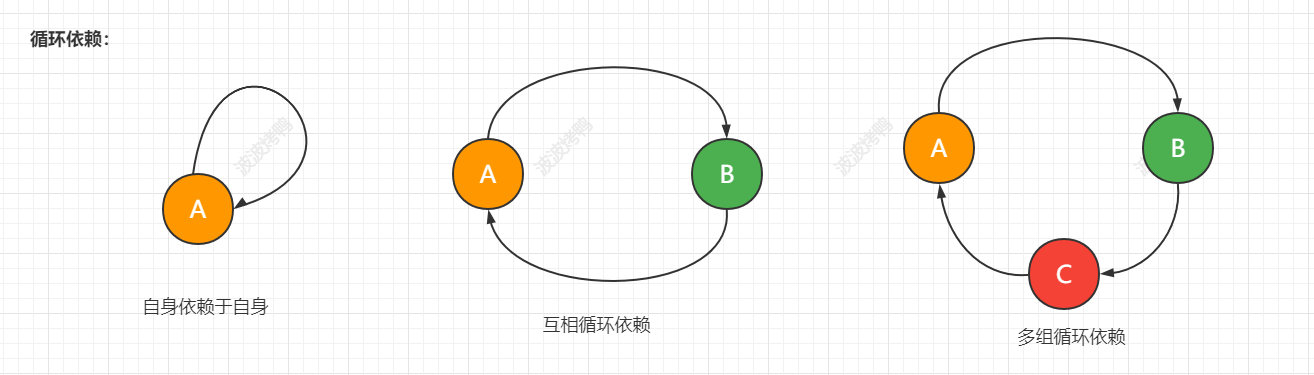
@Import注解是import标签的替换，在SpringBoot的自动装配中非常重要，也是EnableXXX的前置基础。

# **5.循环依赖**

面试的重点，大厂必问之一：

## **5.1 什么是循环依赖**

看下图



  上图是循环依赖的三种情况，虽然方式有点不一样，但是循环依赖的本质是一样的，就你的完整创建要依赖与我，我的完整创建也依赖于你。相互依赖从而没法完整创建造成失败。

## **5.2 代码演示**

  我们再通过代码的方式来演示下循环依赖的效果

public class CircularTest {

public static void main(String[] args) {

new CircularTest1();

}

}class CircularTest1{

private CircularTest2 circularTest2 = new CircularTest2();

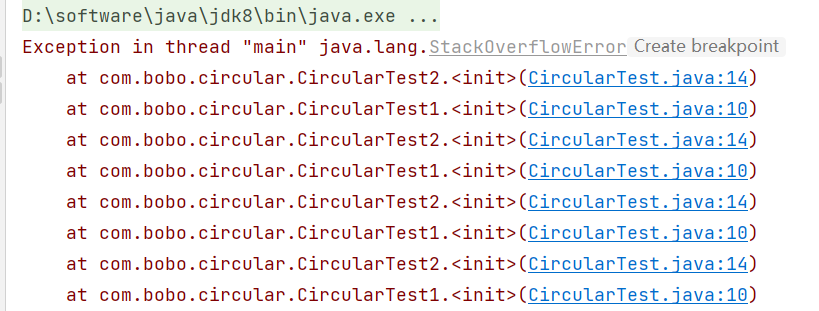
}

class CircularTest2{

private CircularTest1 circularTest1 = new CircularTest1();

}

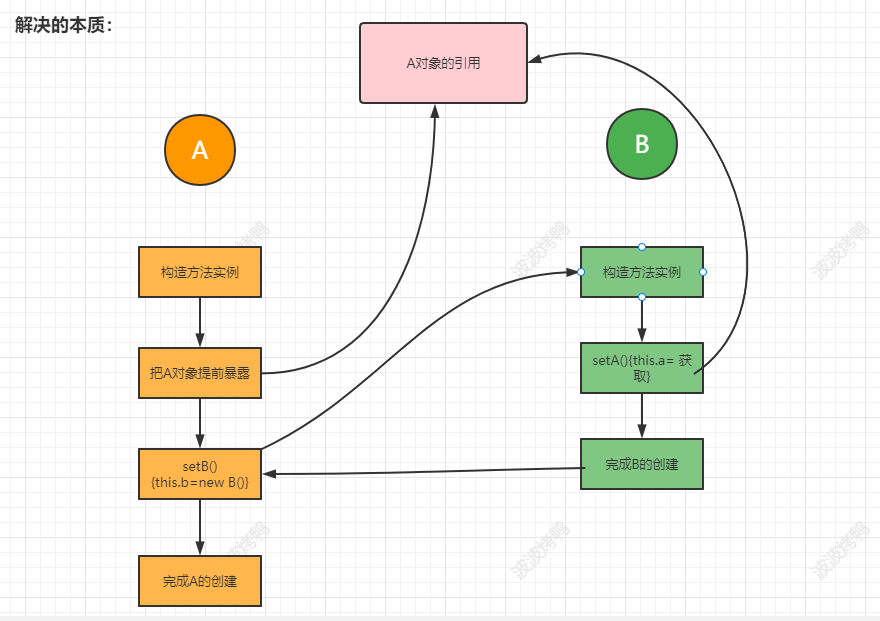
执行后出现了 StackOverflowError 错误



  上面的就是最基本的循环依赖的场景，你需要我，我需要你，然后就报错了。而且上面的这种设计情况我们是没有办法解决的。那么针对这种场景我们应该要怎么设计呢？这个是关键！

## **5.3 分析问题**

  首先我们要明确一点就是如果这个对象A还没创建成功，在创建的过程中要依赖另一个对象B，而另一个对象B也是在创建中要依赖对象A，这种肯定是无解的，这时我们就要转换思路，我们先把A创建出来，但是还没有完成初始化操作，也就是这是一个半成品的对象，然后在赋值的时候先把A暴露出来，然后创建B，让B创建完成后找到暴露的A完成整体的实例化，这时再把B交给A完成A的后续操作，从而揭开了循环依赖的密码。也就是如下图：



## **5.4 自己解决**

  明白了上面的本质后，我们可以自己来尝试解决下：

先来把上面的案例改为set/get来依赖关联

public class CircularTest {

public static void main(String[] args) throws Exception{

System.out.println(getBean(CircularTest1.class).getCircularTest2());

System.out.println(getBean(CircularTest2.class).getCircularTest1());

}

private static <T> T getBean(Class<T> beanClass) throws Exception{

// 1.获取 实例对象

Object obj = beanClass.newInstance();

// 2.完成属性填充

Field[] declaredFields = obj.getClass().getDeclaredFields();

// 遍历处理

for (Field field : declaredFields) {

field.setAccessible(true); // 针对private修饰

// 获取成员变量 对应的类对象

Class<?> fieldClass = field.getType();

// 获取对应的 beanName

String fieldBeanName = fieldClass.getSimpleName().toLowerCase();

// 给成员变量赋值 如果 singletonObjects 中有半成品就获取，否则创建对象

field.set(obj,getBean(fieldClass));

}

return (T) obj;

}

}

class CircularTest1{

private CircularTest2 circularTest2;

public CircularTest2 getCircularTest2() {

return circularTest2;

}

public void setCircularTest2(CircularTest2 circularTest2) {

this.circularTest2 = circularTest2;

}

}

class CircularTest2{

private CircularTest1 circularTest1;

public CircularTest1 getCircularTest1() {

return circularTest1;

}

public void setCircularTest1(CircularTest1 circularTest1) {

this.circularTest1 = circularTest1;

}

}

然后我们再通过把对象实例化和成员变量赋值拆解开来处理。从而解决循环依赖的问题

public class CircularTest {

// 保存提前暴露的对象，也就是半成品的对象

private final static Map<String,Object> singletonObjects = new ConcurrentHashMap<>();

public static void main(String[] args) throws Exception{

System.out.println(getBean(CircularTest1.class).getCircularTest2());

System.out.println(getBean(CircularTest2.class).getCircularTest1());

}

private static <T> T getBean(Class<T> beanClass) throws Exception{

//1.获取类对象对应的名称

String beanName = beanClass.getSimpleName().toLowerCase();

// 2.根据名称去 singletonObjects 中查看是否有半成品的对象

if(singletonObjects.containsKey(beanName)){

return (T) singletonObjects.get(beanName);

}

// 3. singletonObjects 没有半成品的对象，那么就反射实例化对象

Object obj = beanClass.newInstance();

// 还没有完整的创建完这个对象就把这个对象存储在了 singletonObjects中

singletonObjects.put(beanName,obj);

// 属性填充来补全对象

Field[] declaredFields = obj.getClass().getDeclaredFields();

// 遍历处理

for (Field field : declaredFields) {

field.setAccessible(true); // 针对private修饰

// 获取成员变量 对应的类对象

Class<?> fieldClass = field.getType();

// 获取对应的 beanName

String fieldBeanName = fieldClass.getSimpleName().toLowerCase();

// 给成员变量赋值 如果 singletonObjects 中有半成品就获取，否则创建对象

field.set(obj,singletonObjects.containsKey(fieldBeanName)?

singletonObjects.get(fieldBeanName):getBean(fieldClass));

}

return (T) obj;

}

}

class CircularTest1{

private CircularTest2 circularTest2;

public CircularTest2 getCircularTest2() {

return circularTest2;

}

public void setCircularTest2(CircularTest2 circularTest2) {

this.circularTest2 = circularTest2;

}

}

class CircularTest2{

private CircularTest1 circularTest1;

public CircularTest1 getCircularTest1() {

return circularTest1;

}

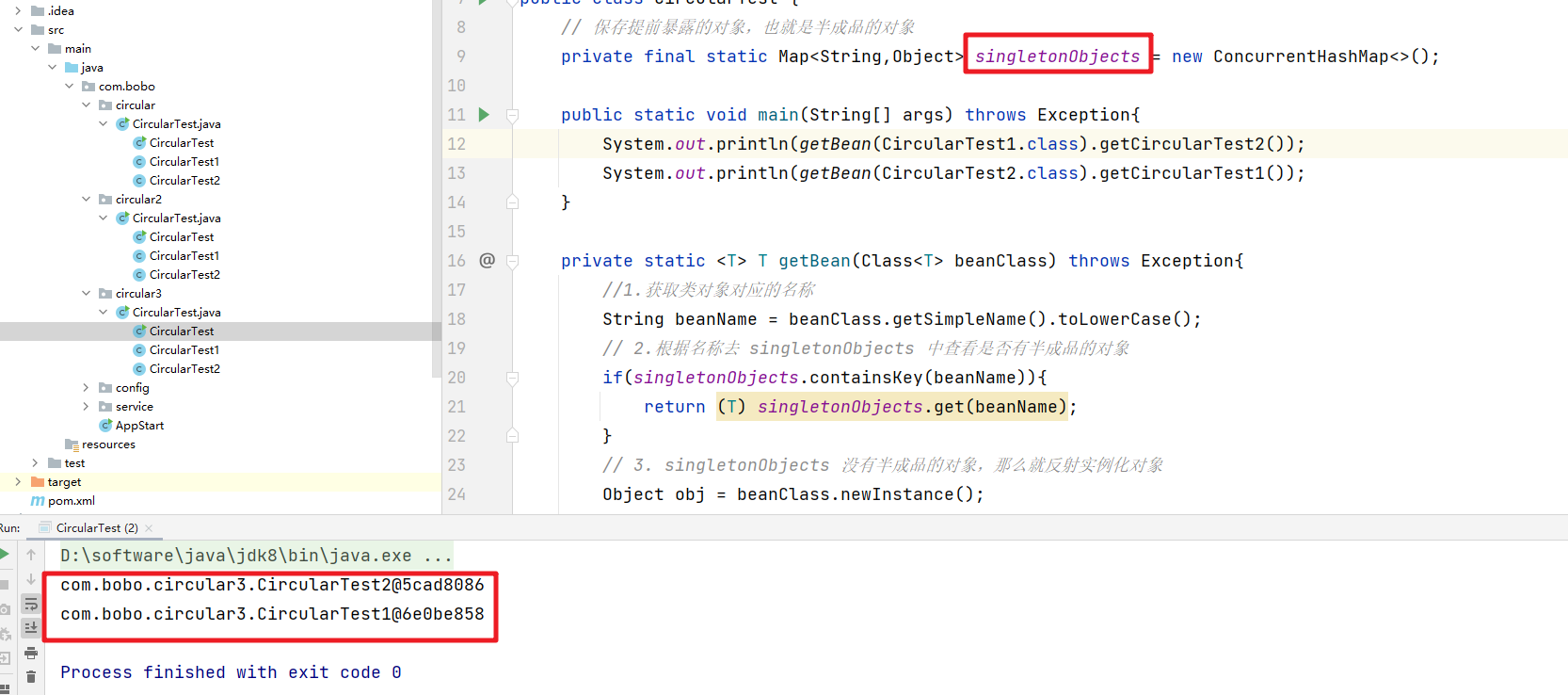
public void setCircularTest1(CircularTest1 circularTest1) {

this.circularTest1 = circularTest1;

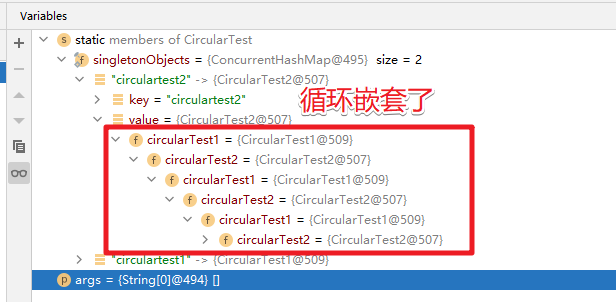
}

}

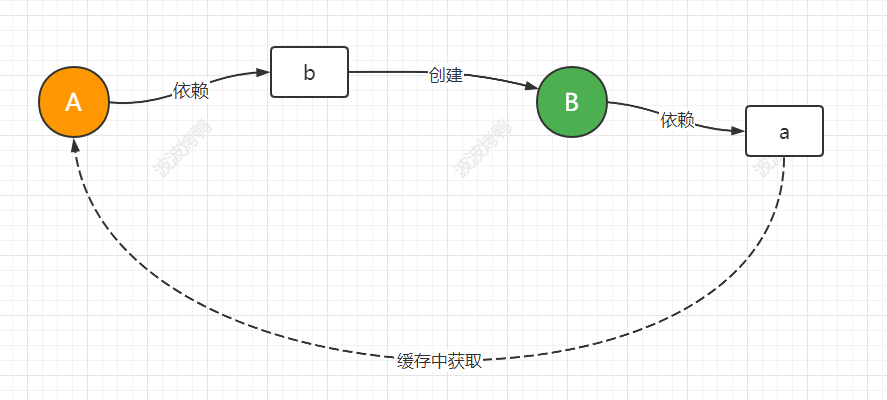
运行程序你会发现问题完美的解决了



  在上面的方法中的核心是getBean方法，Test1 创建后填充属性时依赖Test2，那么就去创建 Test2，在创建 Test2 开始填充时发现依赖于 Test1，但此时 Test1 这个半成品对象已经存放在缓存到 singletonObjects 中了，所以Test2可以正常创建，在通过递归把 Test1 也创建完整了。

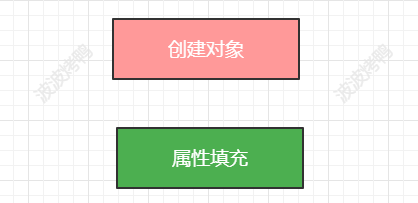


最后总结下该案例解决的本质：

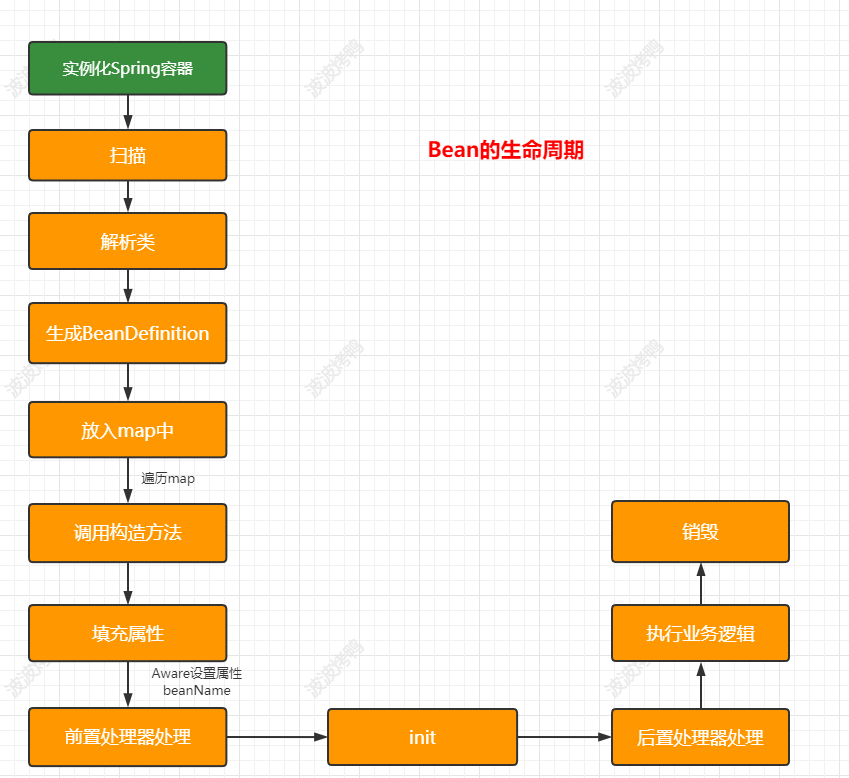


## **5.5 Spring循环依赖**

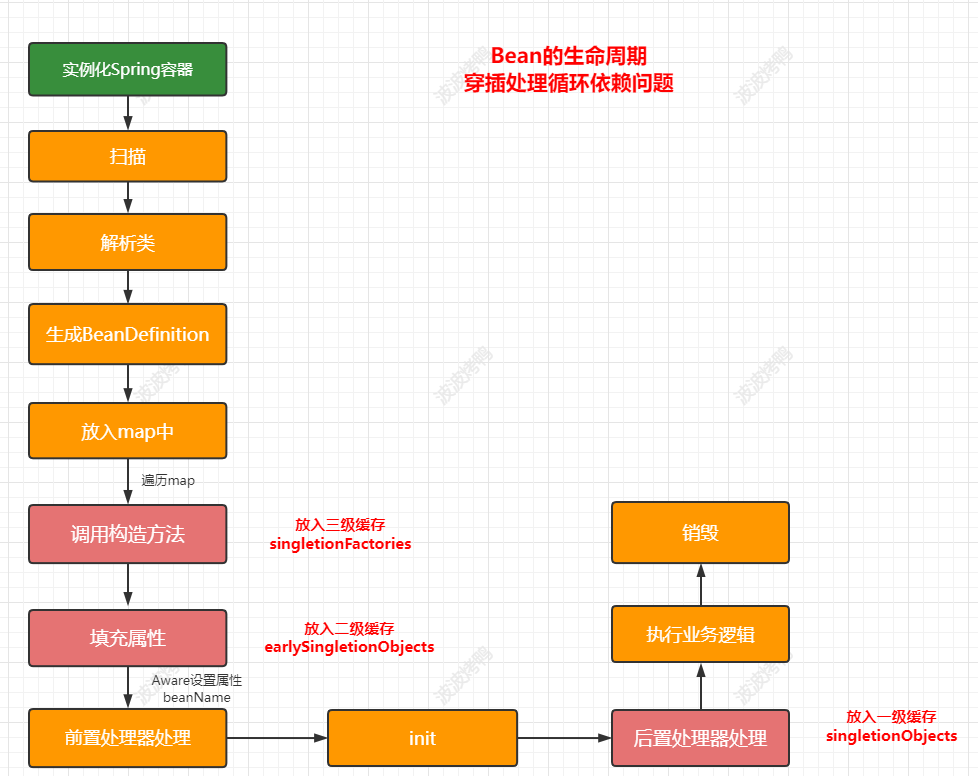
  然后我们再来看看Spring中是如何解决循环依赖问题的呢？刚刚上面的案例中的对象的生命周期的核心就两个



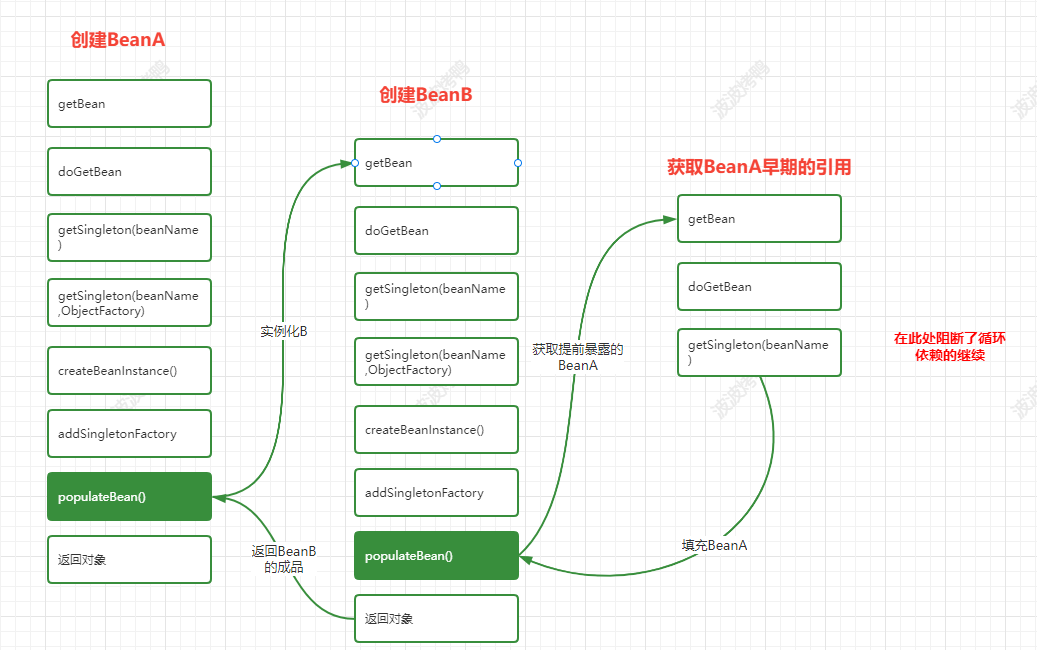
  而Spring创建Bean的生命周期中涉及到的方法就很多了。下面是简单列举了对应的方法



  基于前面案例的了解，我们知道肯定需要在调用构造方法方法创建完成后再暴露对象，在Spring中提供了三级缓存来处理这个事情，对应的处理节点如下图：

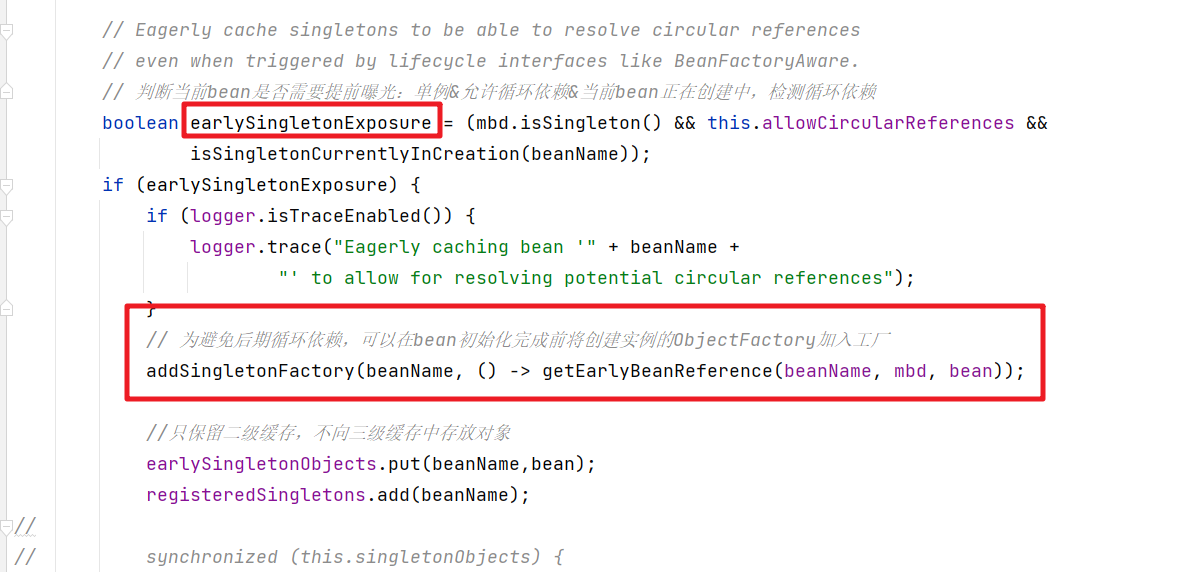


对应到源码中具体处理循环依赖的流程如下：



  上面就是在Spring的生命周期方法中和循环依赖出现相关的流程了。那么源码中的具体处理是怎么样的呢？我们继续往下面看。

首先在调用构造方法的后会放入到三级缓存中



下面就是放入三级缓存的逻辑

protected void addSingletonFactory(String beanName, ObjectFactory<?> singletonFactory) {

Assert.notNull(singletonFactory, "Singleton factory must not be null");

// 使用singletonObjects进行加锁，保证线程安全

synchronized (this.singletonObjects) {

// 如果单例对象的高速缓存【beam名称-bean实例】没有beanName的对象

if (!this.singletonObjects.containsKey(beanName)) {

// 将beanName,singletonFactory放到单例工厂的缓存【bean名称 - ObjectFactory】

this.singletonFactories.put(beanName, singletonFactory);

// 从早期单例对象的高速缓存【bean名称-bean实例】 移除beanName的相关缓存对象

this.earlySingletonObjects.remove(beanName);

// 将beanName添加已注册的单例集中

this.registeredSingletons.add(beanName);

}

}

}

然后在填充属性的时候会存入二级缓存中

earlySingletonObjects.put(beanName,bean);

registeredSingletons.add(beanName);

最后把创建的对象保存在了一级缓存中

protected void addSingleton(String beanName, Object singletonObject) {

synchronized (this.singletonObjects) {

// 将映射关系添加到单例对象的高速缓存中

this.singletonObjects.put(beanName, singletonObject);

// 移除beanName在单例工厂缓存中的数据

this.singletonFactories.remove(beanName);

// 移除beanName在早期单例对象的高速缓存的数据

this.earlySingletonObjects.remove(beanName);

// 将beanName添加到已注册的单例集中

this.registeredSingletons.add(beanName);

}

}

## **5.6 疑问点**

这些疑问点也是面试官喜欢问的问题点

### **为什么需要三级缓存**

三级缓存主要处理的是AOP的代理对象，存储的是一个ObjectFactory

三级缓存考虑的是带你对象，而二级缓存考虑的是性能-从三级缓存的工厂里创建出对象，再扔到二级缓存（这样就不用每次都要从工厂里拿）

### **没有三级环境能解决吗？**

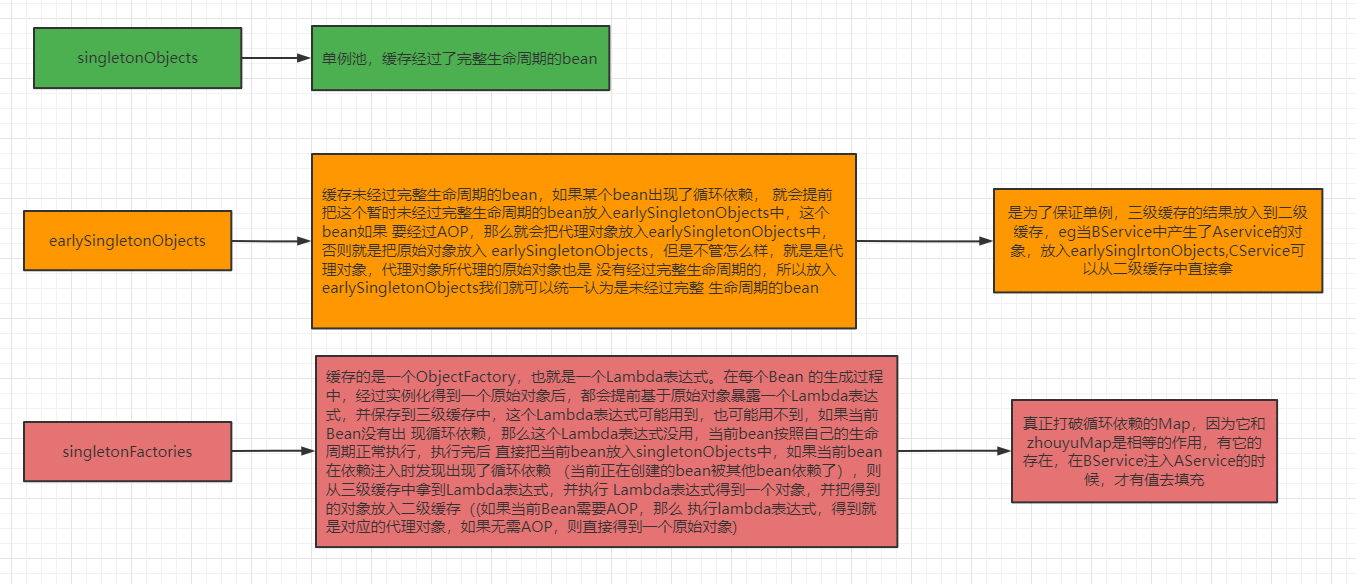
没有三级缓存是可以解决循环依赖问题的

### **三级缓存分别什么作用**

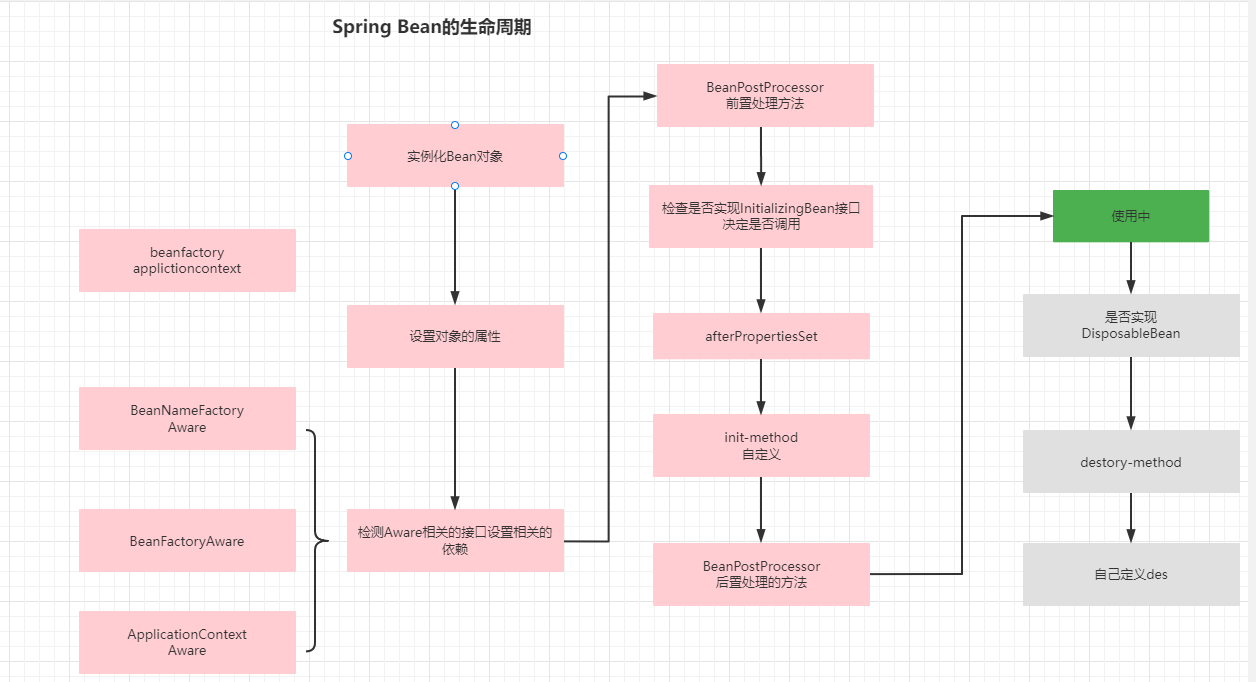
一级缓存：正式对象

二级缓存：半成品对象

三级缓存：工厂



# **6.Spring的生命周期**



结合图，把Bean对象在Spring中的关键节点介绍一遍

# **7.Spring中支持几种作用域**

Spring容器中的bean可以分为5个范围：

1. prototype：为每一个bean请求提供一个实例。
2. singleton：默认，每个容器中只有一个bean的实例，单例的模式由BeanFactory自身来维护。
3. request：为每一个网络请求创建一个实例，在请求完成以后，bean会失效并被垃圾回收器回收。
4. session：与request范围类似，确保每个session中有一个bean的实例，在session过期后，bean会随之失效。
5. global-session：全局作用域，global-session和Portlet应用相关。当你的应用部署在Portlet容器中工作时，它包含很多portlet。如果你想要声明让所有的portlet共用全局的存储变量的话，那么这全局变量需要存储在global-session中。全局作用域与Servlet中的session作用域效果相同。

# **8.说说事务的隔离级别**

  事务隔离级别指的是一个事务对数据的修改与另一个并行的事务的隔离程度，当多个事务同时访问相同数据时，如果没有采取必要的隔离机制，就可能发生以下问题：

| **问题** | **描述** |
| --- | --- |
| 脏读 | 一个事务读到另一个事务未提交的更新数据，所谓脏读，就是指事务A读到了事务B还没有提交的数据，比如银行取钱，事务A开启事务，此时切换到事务B，事务B开启事务-->取走100元，此时切换回事务A，事务A读取的肯定是数据库里面的原始数据，因为事务B取走了100块钱，并没有提交，数据库里面的账务余额肯定还是原始余额，这就是脏读 |
| 幻读 | 是指当事务不是独立执行时发生的一种现象，例如第一个事务对一个表中的数据进行了修改，这种修改涉及到表中的全部数据行。 同时，第二个事务也修改这个表中的数据，这种修改是向表中插入一行新数据。那么，以后就会发生操作第一个事务的用户发现表中还有没有修改的数据行，就好象 发生了幻觉一样。 |
| 不可重复读 | 在一个事务里面的操作中发现了未被操作的数据 比方说在同一个事务中先后执行两条一模一样的select语句，期间在此次事务中没有执行过任何DDL语句，但先后得到的结果不一致，这就是不可重复读 |

**Spring支持的隔离级别**

| **隔离级别** | **描述** |
| --- | --- |
| DEFAULT | 使用数据库本身使用的隔离级别 `` ORACLE（读已提交） MySQL（可重复读） |
| READ\_UNCOMITTED | 读未提交（脏读）最低的隔离级别，一切皆有可能。 |
| READ\_COMMITED | 读已提交，ORACLE默认隔离级别，有幻读以及不可重复读风险。 |
| REPEATABLE\_READ | 可重复读，解决不可重复读的隔离级别，但还是有幻读风险。 |
| SERLALIZABLE | 串行化，最高的事务隔离级别，不管多少事务，挨个运行完一个事务的所有子事务之后才可以执行另外一个事务里面的所有子事务，这样就解决了脏读、不可重复读和幻读的问题了 |



  再必须强调一遍，不是事务隔离级别设置得越高越好，事务隔离级别设置得越高，意味着势必要花手段去加锁用以保证事务的正确性，那么效率就要降低，因此实际开发中往往要在效率和并发正确性之间做一个取舍，一般情况下会设置为READ\_COMMITED，此时避免了脏读，并发性也还不错，之后再通过一些别的手段去解决不可重复读和幻读的问题就好了。

# **9.事务的传播行为**

Spring中的7个事务传播行为:

| **事务行为** | **说明** |
| --- | --- |
| PROPAGATION\_REQUIRED | 支持当前事务，假设当前没有事务。就新建一个事务 |
| PROPAGATION\_SUPPORTS | 支持当前事务，假设当前没有事务，就以非事务方式运行 |
| PROPAGATION\_MANDATORY | 支持当前事务，假设当前没有事务，就抛出异常 |
| PROPAGATION\_REQUIRES\_NEW | 新建事务，假设当前存在事务。把当前事务挂起 |
| PROPAGATION\_NOT\_SUPPORTED | 以非事务方式运行操作。假设当前存在事务，就把当前事务挂起 |
| PROPAGATION\_NEVER | 以非事务方式运行，假设当前存在事务，则抛出异常 |
| PROPAGATION\_NESTED | 如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION\_REQUIRED类似的操作。 |

**举例说明**

**案例代码**

ServiceA

ServiceA {

void methodA() {

ServiceB.methodB();

}

}

ServiceB

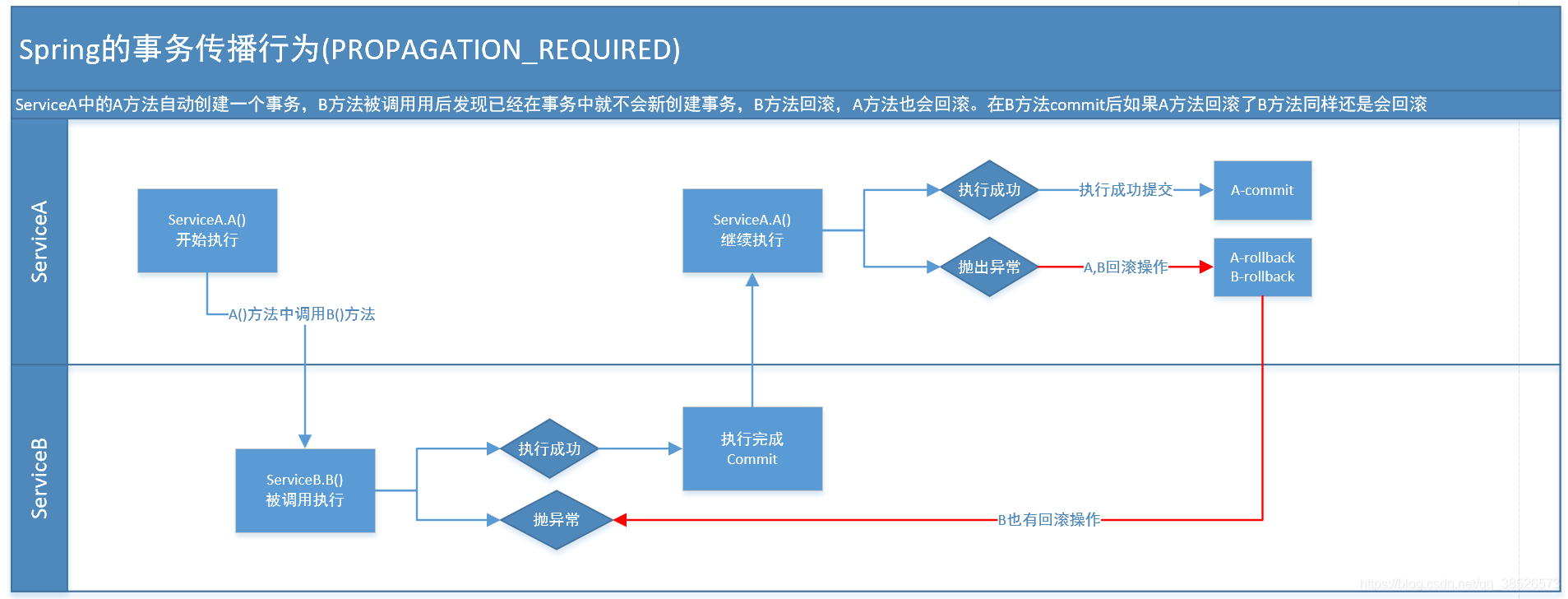
ServiceB {

void methodB() {

}

}

### **1.PROPAGATION\_REQUIRED**

  假如当前正要运行的事务不在另外一个事务里，那么就起一个新的事务 比方说，ServiceB.methodB的事务级别定义PROPAGATION\_REQUIRED, 那么因为执行ServiceA.methodA的时候，ServiceA.methodA已经起了事务。这时调用ServiceB.methodB，ServiceB.methodB看到自己已经执行在ServiceA.methodA的事务内部。就不再起新的事务。而假如ServiceA.methodA执行的时候发现自己没有在事务中，他就会为自己分配一个事务。这样，在ServiceA.methodA或者在ServiceB.methodB内的不论什么地方出现异常。事务都会被回滚。即使ServiceB.methodB的事务已经被提交，可是ServiceA.methodA在接下来fail要回滚，ServiceB.methodB也要回滚  


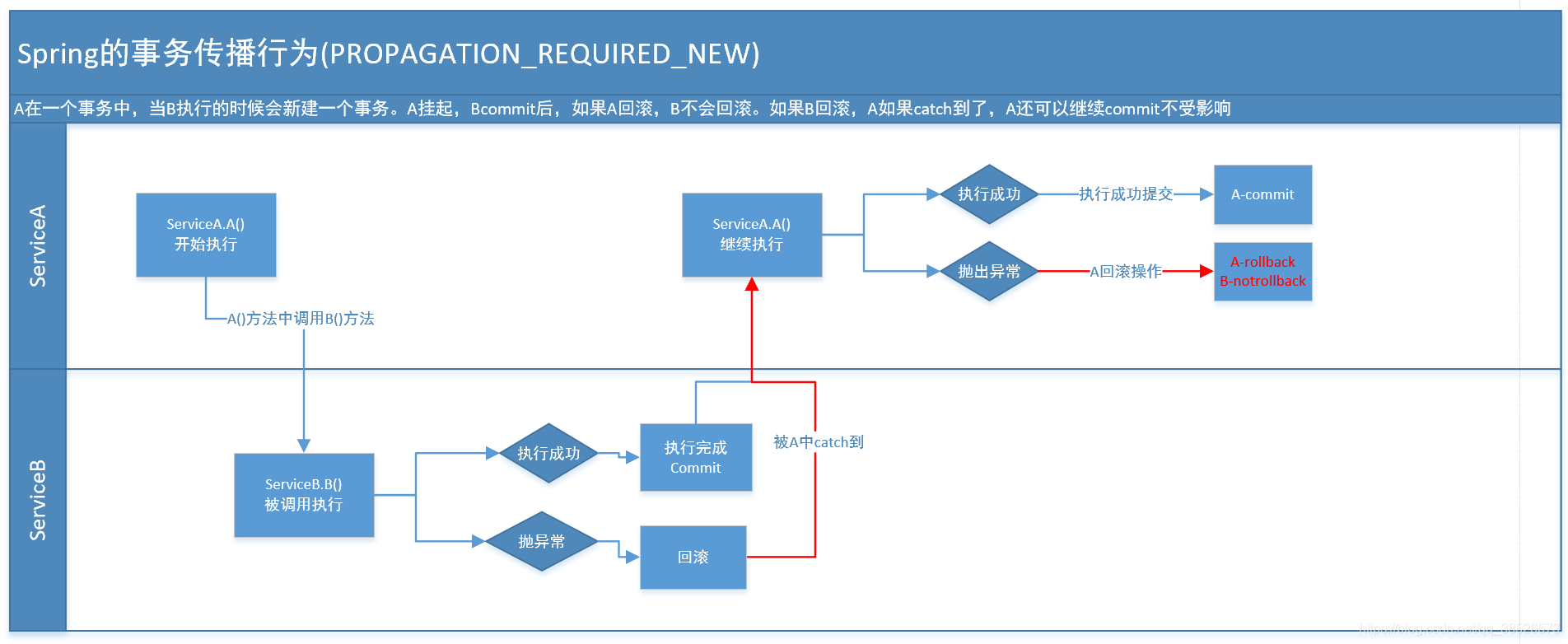
### **2.PROPAGATION\_SUPPORTS**

  假设当前在事务中。即以事务的形式执行。假设当前不在一个事务中，那么就以非事务的形式执行

### **3PROPAGATION\_MANDATORY**

  必须在一个事务中执行。也就是说，他仅仅能被一个父事务调用。否则，他就要抛出异常

### **4.PROPAGATION\_REQUIRES\_NEW**

  这个就比较绕口了。 比方我们设计ServiceA.methodA的事务级别为PROPAGATION\_REQUIRED，ServiceB.methodB的事务级别为PROPAGATION\_REQUIRES\_NEW。那么当运行到ServiceB.methodB的时候，ServiceA.methodA所在的事务就会挂起。ServiceB.methodB会起一个新的事务。等待ServiceB.methodB的事务完毕以后，他才继续运行。  
他与PROPAGATION\_REQUIRED 的事务差别在于事务的回滚程度了。由于ServiceB.methodB是新起一个事务，那么就是存在两个不同的事务。假设ServiceB.methodB已经提交，那么ServiceA.methodA失败回滚。ServiceB.methodB是不会回滚的。假设ServiceB.methodB失败回滚，假设他抛出的异常被ServiceA.methodA捕获，ServiceA.methodA事务仍然可能提交。  


### **5.PROPAGATION\_NOT\_SUPPORTED**

  当前不支持事务。比方ServiceA.methodA的事务级别是PROPAGATION\_REQUIRED 。而ServiceB.methodB的事务级别是PROPAGATION\_NOT\_SUPPORTED ，那么当执行到ServiceB.methodB时。ServiceA.methodA的事务挂起。而他以非事务的状态执行完，再继续ServiceA.methodA的事务。

### **6.PROPAGATION\_NEVER**

  不能在事务中执行。  
如果ServiceA.methodA的事务级别是PROPAGATION\_REQUIRED。 而ServiceB.methodB的事务级别是PROPAGATION\_NEVER ，那么ServiceB.methodB就要抛出异常了。

### **7.PROPAGATION\_NESTED**

  如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION\_REQUIRED类似的操作。

# **10.Spring事务实现的方式**

**编程式事务管理**：这意味着你可以通过编程的方式管理事务，这种方式带来了很大的灵活性，但很难维护。

**声明式事务管理**：这种方式意味着你可以将事务管理和业务代码分离。你只需要通过注解或者XML配置管理事务。

# **11.事务注解的本质是什么**

  @Transactional 这个注解仅仅是一些（和事务相关的）元数据，在运行时被事务基础设施读取消费，并使用这些元数据来配置bean的事务行为。 大致来说具有两方面功能，一是表明该方法要参与事务，二是配置相关属性来定制事务的参与方式和运行行为

  声明式事务主要是得益于Spring AOP。使用一个事务拦截器，在方法调用的前后/周围进行事务性增强（advice），来驱动事务完成。

  @Transactional注解既可以标注在类上，也可以标注在方法上。当在类上时，默认应用到类里的所有方法。如果此时方法上也标注了，则方法上的优先级高。 另外

01-Spring面试专题

讲师：邓澎波

# **Spring面试专题**

# **1.Spring应该很熟悉吧？来介绍下你的Spring的理解**

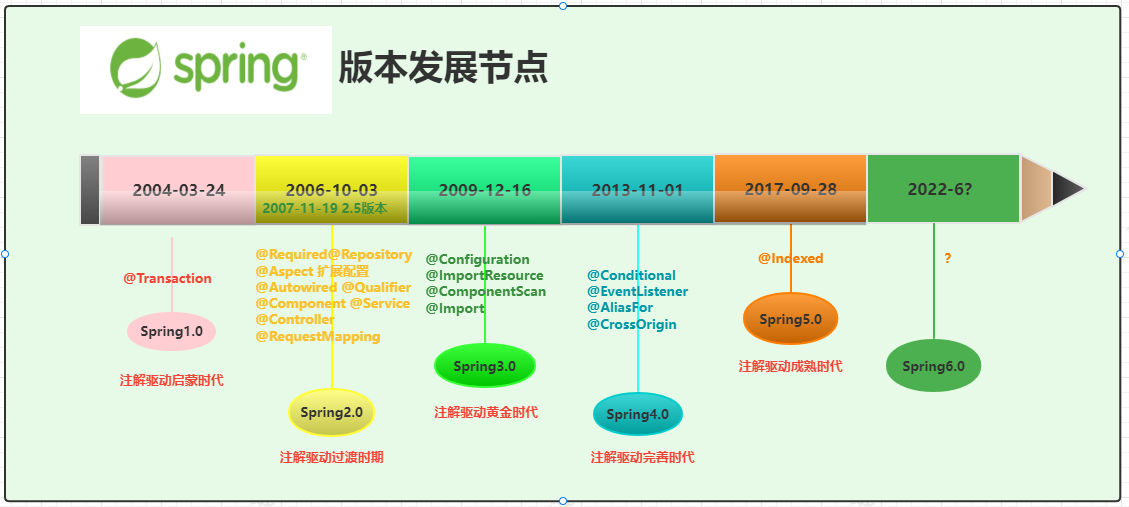
  有些同学可能会抢答，不熟悉!!!



  好了，不开玩笑，面对这个问题我们应该怎么来回答呢？我们给大家梳理这个几个维度来回答

## **1.1 Spring的发展历程**

  先介绍Spring是怎么来的，发展中有哪些核心的节点，当前的最新版本是什么等



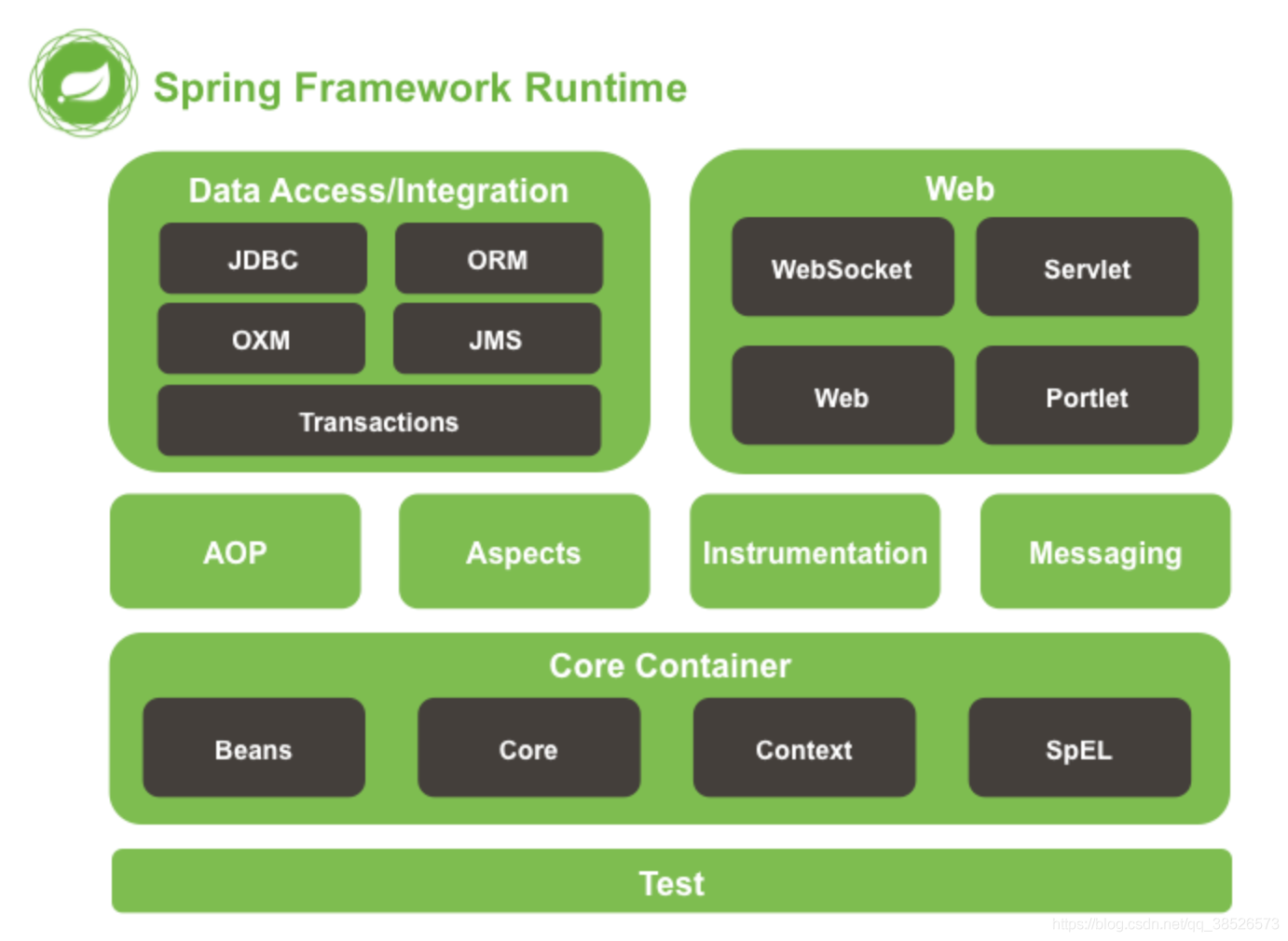
 通过上图可以比较清晰的看到Spring的各个时间版本对应的时间节点了。也就是Spring从之前单纯的xml的配置方式，到现在的完全基于注解的编程方式发展。

## **1.2 Spring的组成**

  Spring是一个轻量级的IoC和AOP容器框架。是为Java应用程序提供基础性服务的一套框架，目的是用于简化企业应用程序的开发，它使得开发者只需要关心业务需求。常见的配置方式有三种：基于XML的配置、基于注解的配置、基于Java的配置.

主要由以下几个模块组成：

* Spring Core：核心类库，提供IOC服务；
* Spring Context：提供框架式的Bean访问方式，以及企业级功能（JNDI、定时任务等）；
* Spring AOP：AOP服务；
* Spring DAO：对JDBC的抽象，简化了数据访问异常的处理；
* Spring ORM：对现有的ORM框架的支持；
* Spring Web：提供了基本的面向Web的综合特性，例如多方文件上传；
* Spring MVC：提供面向Web应用的Model-View-Controller实现。



## **1.3 Spring的好处**

| **序号** | **好处** | **说明** |
| --- | --- | --- |
| 1 | 轻量 | Spring 是轻量的，基本的版本大约2MB。 |
| 2 | 控制反转 | Spring通过控制反转实现了松散耦合，对象们给出它们的依赖，``而不是创建或查找依赖的对象们。 |
| 3 | 面向切面编程(AOP) | Spring支持面向切面的编程，并且把应用业务逻辑和系统服务分开。 |
| 4 | 容器 | Spring 包含并管理应用中对象的生命周期和配置。 |
| 5 | MVC框架 | Spring的WEB框架是个精心设计的框架，是Web框架的一个很好的替代品。 |
| 6 | 事务管理 | Spring 提供一个持续的事务管理接口，``可以扩展到上至本地事务下至全局事务（JTA）。 |
| 7 | 异常处理 | Spring 提供方便的API把具体技术相关的异常 ``(比如由JDBC，Hibernate or JDO抛出的)转化为一致的unchecked 异常。 |
| 8 | 最重要的 | 用的人多！！！ |

# **2.Spring框架中用到了哪些设计模式**

## **2.1 单例模式**

  单例模式应该是大家印象最深的一种设计模式了。在Spring中最明显的使用场景是在配置文件中配置注册bean对象的时候**设置scope的值为singleton** 。

<?xml version="1.0" encoding="UTF-8"?><beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean class="com.dpb.pojo.User" id="user" scope="singleton">

<property name="name" value="波波烤鸭"></property>

</bean></beans>

## **2.2 原型模式**

  原型模式也叫克隆模式，Spring中该模式使用的很明显，和单例一样在bean标签中设置scope的属性prototype即表示该bean以克隆的方式生成

<?xml version="1.0" encoding="UTF-8"?><beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean class="com.dpb.pojo.User" id="user" scope="prototype">

<property name="name" value="波波烤鸭"></property>

</bean></beans>

## **2.3 模板模式**

  模板模式的核心是父类定义好流程，然后将流程中需要子类实现的方法就抽象话留给子类实现，Spring中的JdbcTemplate就是这样的实现。我们知道jdbc的步骤是固定

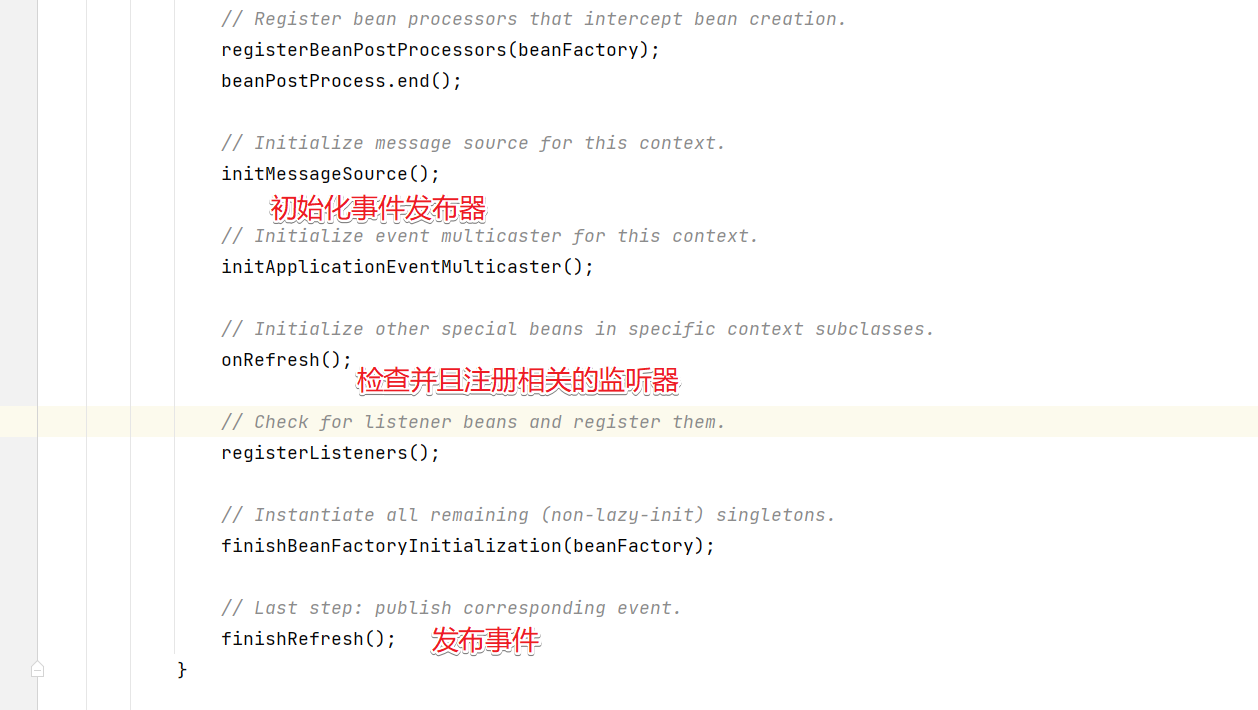
* 加载驱动,
* 获取连接通道,
* 构建sql语句.
* 执行sql语句,
* 关闭资源

  在这些步骤中第3步和第四步是不确定的,所以就留给客户实现，而我们实际使用JdbcTemplate的时候也确实是只需要构建SQL就可以了.这就是典型的模板模式。我们以query方法为例来看下JdbcTemplate中的代码.



## **2.4 观察者模式**

  观察者模式定义的是对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。使用比较场景是在监听器中而spring中Observer模式常用的地方也是listener的实现。如ApplicationListener.



## **2.5 工厂模式**

**简单工厂模式**：

  简单工厂模式就是通过工厂根据传递进来的参数决定产生哪个对象。Spring中我们通过getBean方法获取对象的时候根据id或者name获取就是简单工厂模式了。

<?xml version="1.0" encoding="UTF-8"?><beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.3.xsd">

<context:annotation-config/>

<bean class="com.dpb.pojo.User" id="user" >

<property name="name" value="波波烤鸭"></property>

</bean></beans>

**工厂方法模式**：

  在Spring中我们一般是将Bean的实例化直接交给容器去管理的，实现了使用和创建的分离，这时容器直接管理对象，还有种情况是，bean的创建过程我们交给一个工厂去实现，而Spring容器管理这个工厂。这个就是我们讲的工厂模式，在Spring中有两种实现一种是静态工厂方法模式，一种是动态工厂方法模式。以静态工厂来演示

/\*\*

\* User 工厂类

\* @author dpb[波波烤鸭]

\*

\*/public class UserFactory {

/\*\*

\* 必须是static方法

\* @return

\*/

public static UserBean getInstance(){

return new UserBean();

}

}

application.xml文件中注册

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<!-- 静态工厂方式配置 配置静态工厂及方法 -->

<bean class="com.dpb.factory.UserFactory" factory-method="getInstance" id="user2"/></beans>

## **2.6 适配器模式**

  将一个类的接口转换成客户希望的另外一个接口。使得原本由于接口不兼容而不能一起工作的那些类可以在一起工作。这就是适配器模式。在Spring中在AOP实现中的Advice和interceptor之间的转换就是通过适配器模式实现的。

class MethodBeforeAdviceAdapter implements AdvisorAdapter, Serializable {

@Override

public boolean supportsAdvice(Advice advice) {

return (advice instanceof MethodBeforeAdvice);

}

@Override

public MethodInterceptor getInterceptor(Advisor advisor) {

MethodBeforeAdvice advice = (MethodBeforeAdvice) advisor.getAdvice();

// 通知类型匹配对应的拦截器

return new MethodBeforeAdviceInterceptor(advice);

}

}

## **2.7 装饰者模式**

  装饰者模式又称为包装模式(Wrapper),作用是用来动态的为一个对象增加新的功能。装饰模式是一种用于代替继承的技术，无须通过继承增加子类就能扩展对象的新功能。使用对象的关联关系代替继承关系，更加灵活，同时避免类型体系的快速膨胀。  
  spring中用到的包装器模式在类名上有两种表现：一种是类名中含有Wrapper，另一种是类名中含有Decorator。基本上都是动态地给一个对象添加一些额外的职责。  
  具体的使用在Spring session框架中的SessionRepositoryRequestWrapper使用包装模式对原生的request的功能进行增强，可以将session中的数据和分布式数据库进行同步，这样即使当前tomcat崩溃，session中的数据也不会丢失。

<dependency>

<groupId>org.springframework.session</groupId>

<artifactId>spring-session</artifactId>

<version>1.3.1.RELEASE</version></dependency>

## **2.8 代理模式**

  代理模式应该是大家非常熟悉的设计模式了，在Spring中AOP的实现中代理模式使用的很彻底.

## **2.9 策略模式**

  策略模式对应于解决某一个问题的一个算法族，允许用户从该算法族中任选一个算法解决某一问题，同时可以方便的更换算法或者增加新的算法。并且由客户端决定调用哪个算法，spring中在实例化对象的时候用到Strategy模式。XmlBeanDefinitionReader,PropertiesBeanDefinitionReader

## **2.10 责任链默认**

AOP中的拦截器链

## **2.11 委托者模式**

DelegatingFilterProxy，整合Shiro，SpringSecurity的时候都有用到。

.....

# **3.Autowired和Resource关键字的区别？**

  这是一个相对比较简单的问题，@Resource和@Autowired都是做bean的注入时使用，其实@Resource并不是Spring的注解，它的包是javax.annotation.Resource，需要导入，但是Spring支持该注解的注入。

## **3.1 共同点**

  两者都可以写在字段和setter方法上。两者如果都写在字段上，那么就不需要再写setter方法.

## **3.2 不同点**

**@Autowired**

  @Autowired为Spring提供的注解，需要导入org.springframework.beans.factory.annotation.Autowired;只按照byType注入。

public class TestServiceImpl {

// 下面两种@Autowired只要使用一种即可

@Autowired

private UserDao userDao; // 用于字段上

@Autowired

public void setUserDao(UserDao userDao) { // 用于属性的方法上

this.userDao = userDao;

}

}

@Autowired注解是按照类型（byType）装配依赖对象，默认情况下它要求依赖对象必须存在，如果允许null值，可以设置它的required属性为false。如果我们想使用按照名称（byName）来装配，可以结合@Qualififier注解一起使用。如下：

public class TestServiceImpl {

@Autowired

@Qualifier("userDao")

private UserDao userDao; }

**@Resource**

  @Resource默认按照ByName自动注入，由J2EE提供，需要导入包javax.annotation.Resource。@Resource有两个重要的属性：name和type，而Spring将@Resource注解的name属性解析为bean的名字，而type属性则解析为bean的类型。所以，如果使用name属性，则使用byName的自动注入策略，而使用type属性时则使用byType自动注入策略。如果既不制定name也不制定type属性，这时将通过反射机制使用byName自动注入策略.

public class TestServiceImpl {

// 下面两种@Resource只要使用一种即可

@Resource(name="userDao")

private UserDao userDao; // 用于字段上

@Resource(name="userDao")

public void setUserDao(UserDao userDao) { // 用于属性的setter方法上

this.userDao = userDao;

}

}

@Resource装配顺序：

1. 如果同时指定了name和type，则从Spring上下文中找到唯一匹配的bean进行装配，找不到则抛出异常。
2. 如果指定了name，则从上下文中查找名称（id）匹配的bean进行装配，找不到则抛出异常。
3. 如果指定了type，则从上下文中找到类似匹配的唯一bean进行装配，找不到或是找到多个，都会抛出异常。
4. 如果既没有指定name，又没有指定type，则自动按照byName方式进行装配；如果没有匹配，则回退为一个原始类型进行匹配，如果匹配则自动装配。

@Resource的作用相当于@Autowired，只不过@Autowired按照byType自动注入。

# **4.Spring中常用的注解有哪些，重点介绍几个**

@Controller @Service @RestController @RequestBody,@Index @Import等

@Indexd提升 @ComponentScan的效率

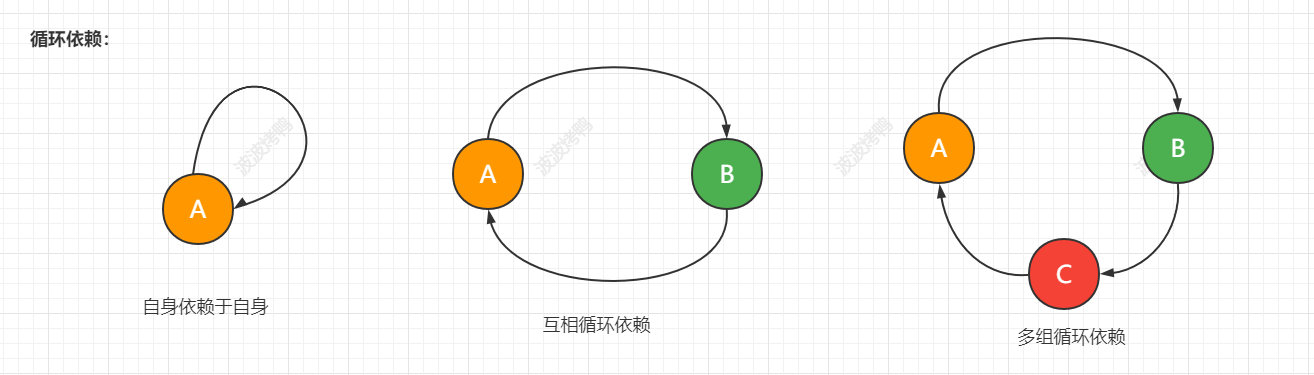
@Import注解是import标签的替换，在SpringBoot的自动装配中非常重要，也是EnableXXX的前置基础。

# **5.循环依赖**

面试的重点，大厂必问之一：

## **5.1 什么是循环依赖**

看下图



  上图是循环依赖的三种情况，虽然方式有点不一样，但是循环依赖的本质是一样的，就你的完整创建要依赖与我，我的完整创建也依赖于你。相互依赖从而没法完整创建造成失败。

## **5.2 代码演示**

  我们再通过代码的方式来演示下循环依赖的效果

public class CircularTest {

public static void main(String[] args) {

new CircularTest1();

}

}class CircularTest1{

private CircularTest2 circularTest2 = new CircularTest2();

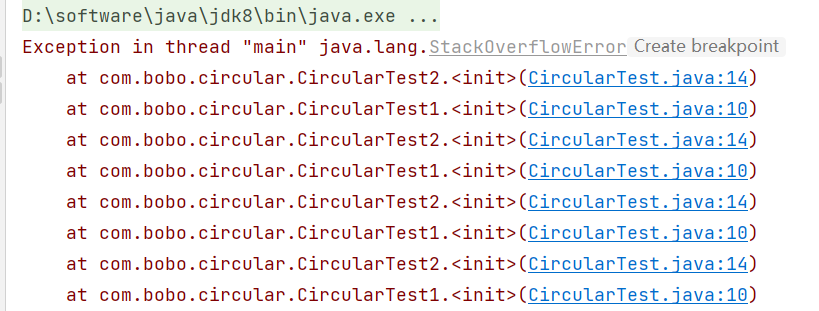
}

class CircularTest2{

private CircularTest1 circularTest1 = new CircularTest1();

}

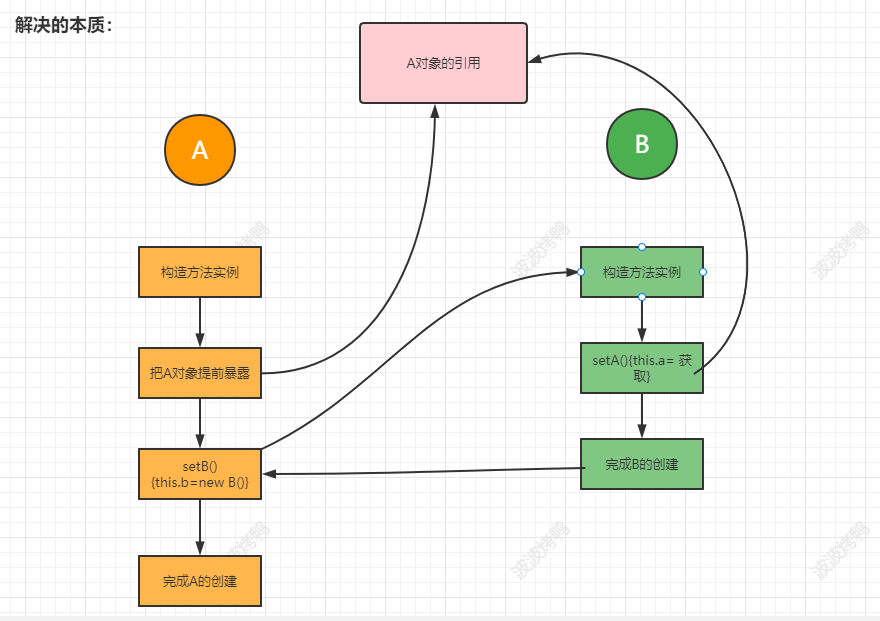
执行后出现了 StackOverflowError 错误



  上面的就是最基本的循环依赖的场景，你需要我，我需要你，然后就报错了。而且上面的这种设计情况我们是没有办法解决的。那么针对这种场景我们应该要怎么设计呢？这个是关键！

## **5.3 分析问题**

  首先我们要明确一点就是如果这个对象A还没创建成功，在创建的过程中要依赖另一个对象B，而另一个对象B也是在创建中要依赖对象A，这种肯定是无解的，这时我们就要转换思路，我们先把A创建出来，但是还没有完成初始化操作，也就是这是一个半成品的对象，然后在赋值的时候先把A暴露出来，然后创建B，让B创建完成后找到暴露的A完成整体的实例化，这时再把B交给A完成A的后续操作，从而揭开了循环依赖的密码。也就是如下图：



## **5.4 自己解决**

  明白了上面的本质后，我们可以自己来尝试解决下：

先来把上面的案例改为set/get来依赖关联

public class CircularTest {

public static void main(String[] args) throws Exception{

System.out.println(getBean(CircularTest1.class).getCircularTest2());

System.out.println(getBean(CircularTest2.class).getCircularTest1());

}

private static <T> T getBean(Class<T> beanClass) throws Exception{

// 1.获取 实例对象

Object obj = beanClass.newInstance();

// 2.完成属性填充

Field[] declaredFields = obj.getClass().getDeclaredFields();

// 遍历处理

for (Field field : declaredFields) {

field.setAccessible(true); // 针对private修饰

// 获取成员变量 对应的类对象

Class<?> fieldClass = field.getType();

// 获取对应的 beanName

String fieldBeanName = fieldClass.getSimpleName().toLowerCase();

// 给成员变量赋值 如果 singletonObjects 中有半成品就获取，否则创建对象

field.set(obj,getBean(fieldClass));

}

return (T) obj;

}

}

class CircularTest1{

private CircularTest2 circularTest2;

public CircularTest2 getCircularTest2() {

return circularTest2;

}

public void setCircularTest2(CircularTest2 circularTest2) {

this.circularTest2 = circularTest2;

}

}

class CircularTest2{

private CircularTest1 circularTest1;

public CircularTest1 getCircularTest1() {

return circularTest1;

}

public void setCircularTest1(CircularTest1 circularTest1) {

this.circularTest1 = circularTest1;

}

}

然后我们再通过把对象实例化和成员变量赋值拆解开来处理。从而解决循环依赖的问题

public class CircularTest {

// 保存提前暴露的对象，也就是半成品的对象

private final static Map<String,Object> singletonObjects = new ConcurrentHashMap<>();

public static void main(String[] args) throws Exception{

System.out.println(getBean(CircularTest1.class).getCircularTest2());

System.out.println(getBean(CircularTest2.class).getCircularTest1());

}

private static <T> T getBean(Class<T> beanClass) throws Exception{

//1.获取类对象对应的名称

String beanName = beanClass.getSimpleName().toLowerCase();

// 2.根据名称去 singletonObjects 中查看是否有半成品的对象

if(singletonObjects.containsKey(beanName)){

return (T) singletonObjects.get(beanName);

}

// 3. singletonObjects 没有半成品的对象，那么就反射实例化对象

Object obj = beanClass.newInstance();

// 还没有完整的创建完这个对象就把这个对象存储在了 singletonObjects中

singletonObjects.put(beanName,obj);

// 属性填充来补全对象

Field[] declaredFields = obj.getClass().getDeclaredFields();

// 遍历处理

for (Field field : declaredFields) {

field.setAccessible(true); // 针对private修饰

// 获取成员变量 对应的类对象

Class<?> fieldClass = field.getType();

// 获取对应的 beanName

String fieldBeanName = fieldClass.getSimpleName().toLowerCase();

// 给成员变量赋值 如果 singletonObjects 中有半成品就获取，否则创建对象

field.set(obj,singletonObjects.containsKey(fieldBeanName)?

singletonObjects.get(fieldBeanName):getBean(fieldClass));

}

return (T) obj;

}

}

class CircularTest1{

private CircularTest2 circularTest2;

public CircularTest2 getCircularTest2() {

return circularTest2;

}

public void setCircularTest2(CircularTest2 circularTest2) {

this.circularTest2 = circularTest2;

}

}

class CircularTest2{

private CircularTest1 circularTest1;

public CircularTest1 getCircularTest1() {

return circularTest1;

}

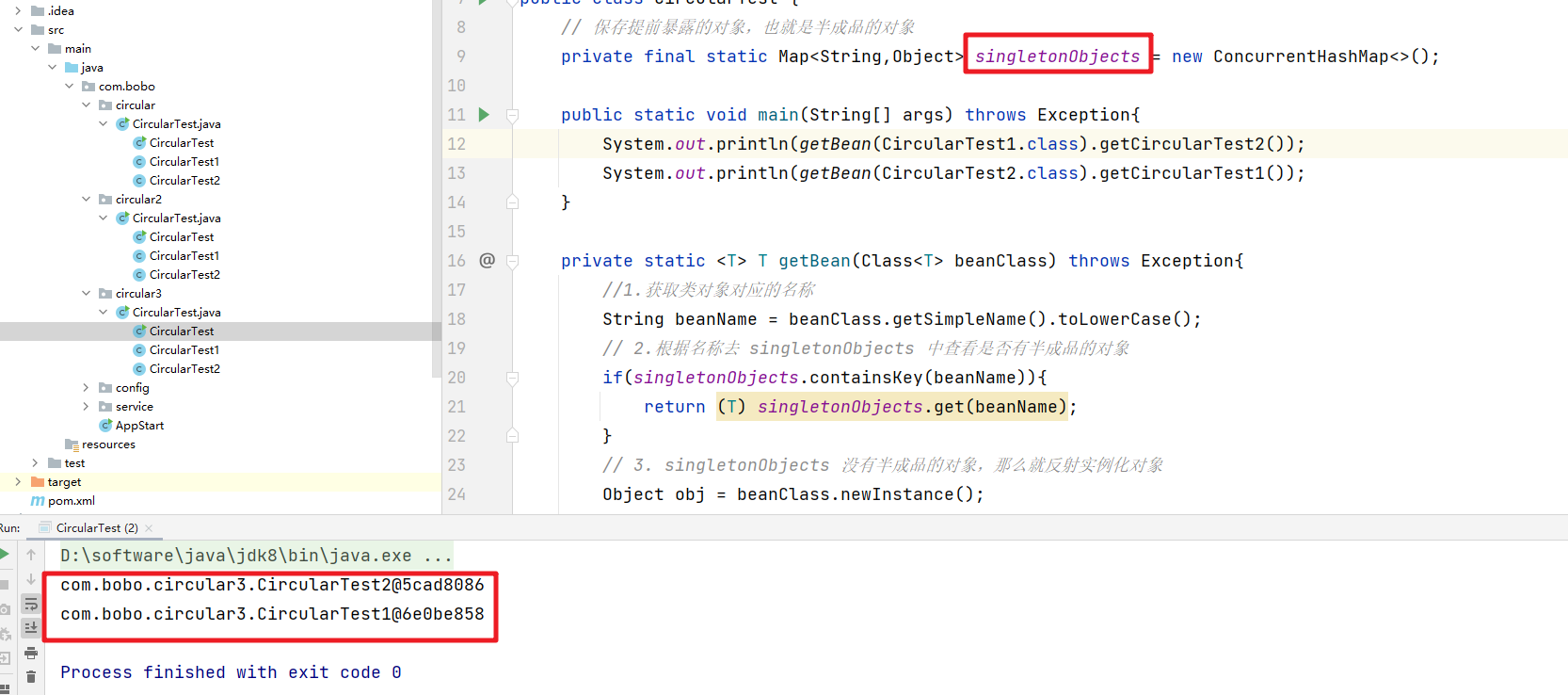
public void setCircularTest1(CircularTest1 circularTest1) {

this.circularTest1 = circularTest1;

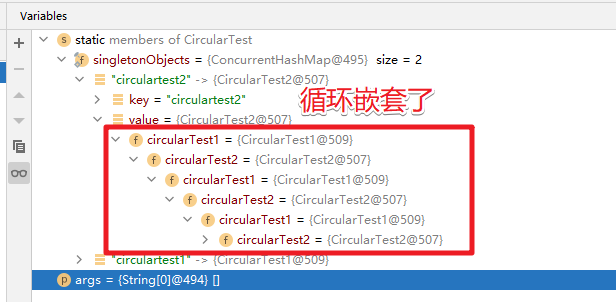
}

}

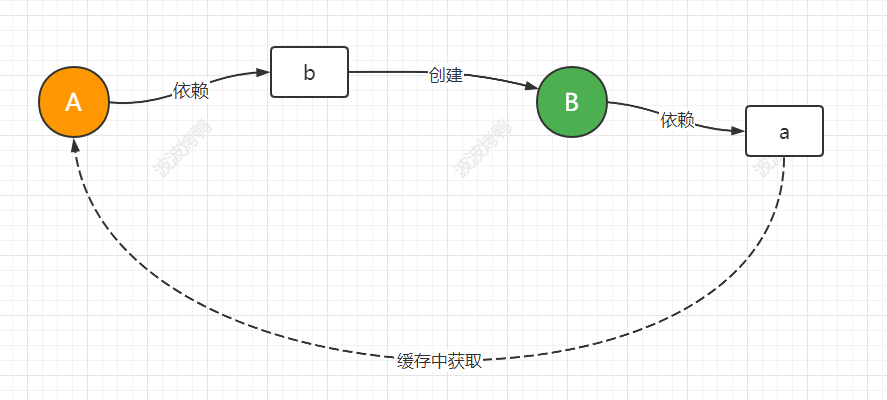
运行程序你会发现问题完美的解决了



  在上面的方法中的核心是getBean方法，Test1 创建后填充属性时依赖Test2，那么就去创建 Test2，在创建 Test2 开始填充时发现依赖于 Test1，但此时 Test1 这个半成品对象已经存放在缓存到 singletonObjects 中了，所以Test2可以正常创建，在通过递归把 Test1 也创建完整了。

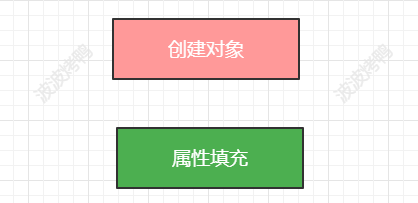


最后总结下该案例解决的本质：

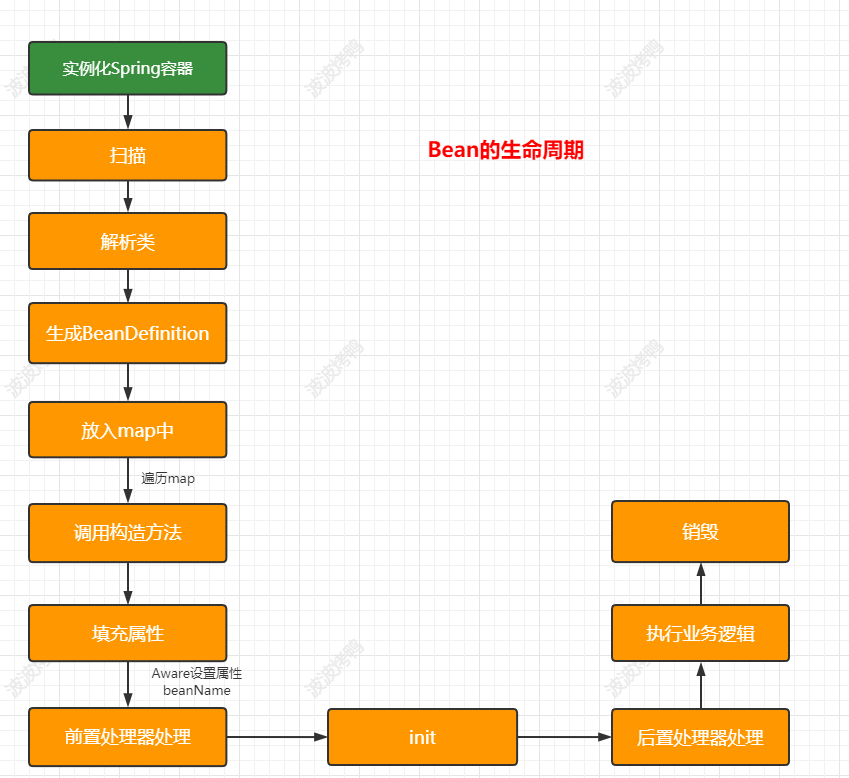


## **5.5 Spring循环依赖**

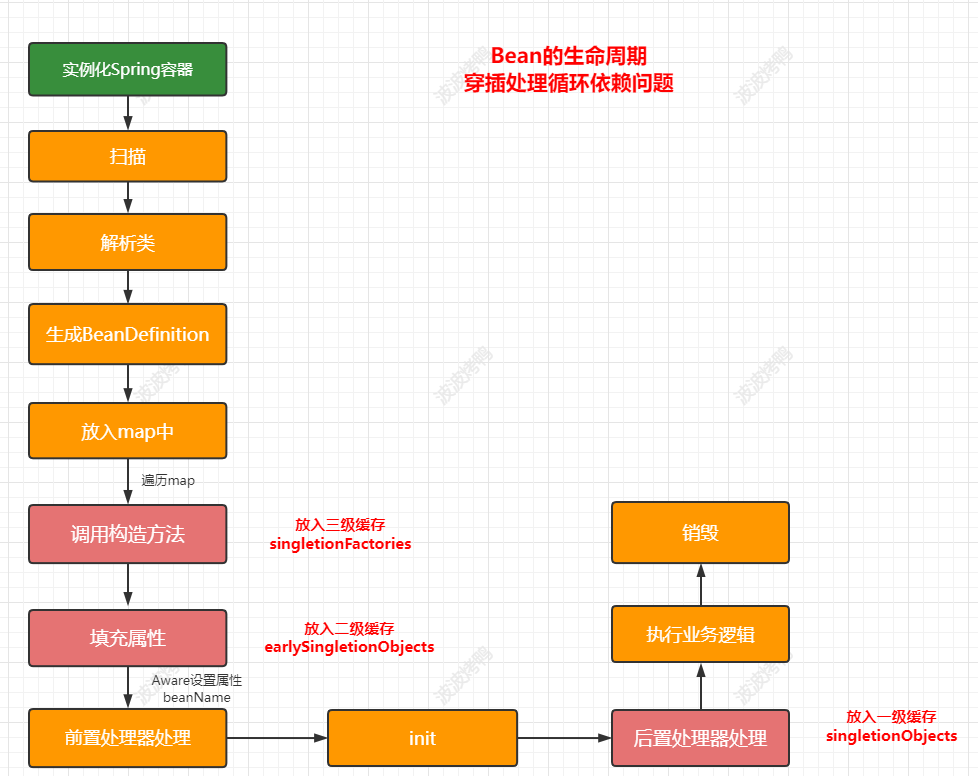
  然后我们再来看看Spring中是如何解决循环依赖问题的呢？刚刚上面的案例中的对象的生命周期的核心就两个



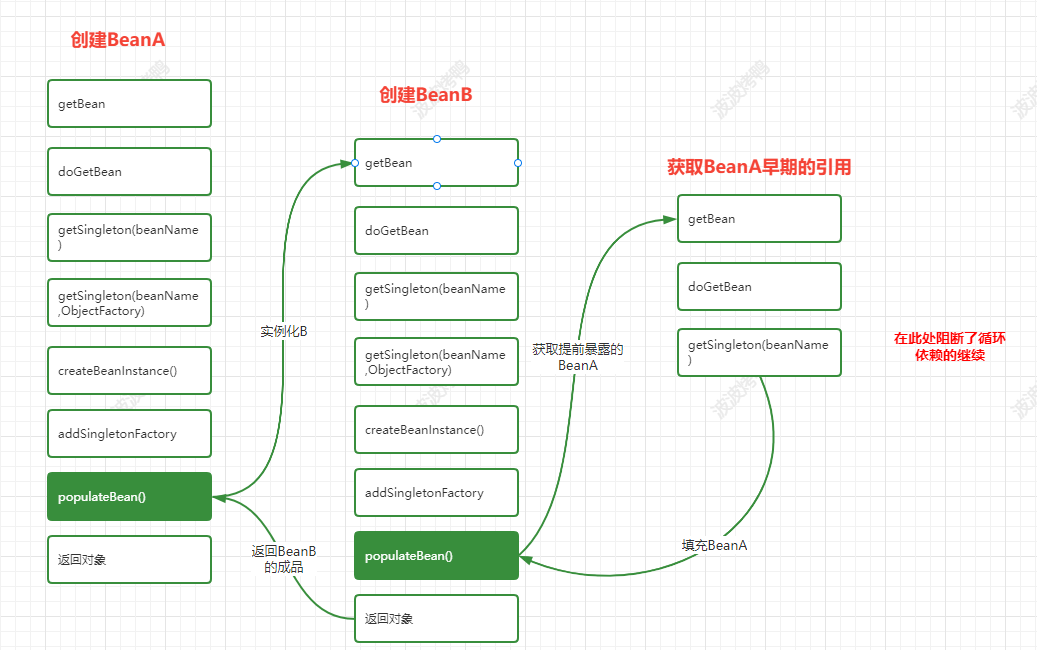
  而Spring创建Bean的生命周期中涉及到的方法就很多了。下面是简单列举了对应的方法



  基于前面案例的了解，我们知道肯定需要在调用构造方法方法创建完成后再暴露对象，在Spring中提供了三级缓存来处理这个事情，对应的处理节点如下图：

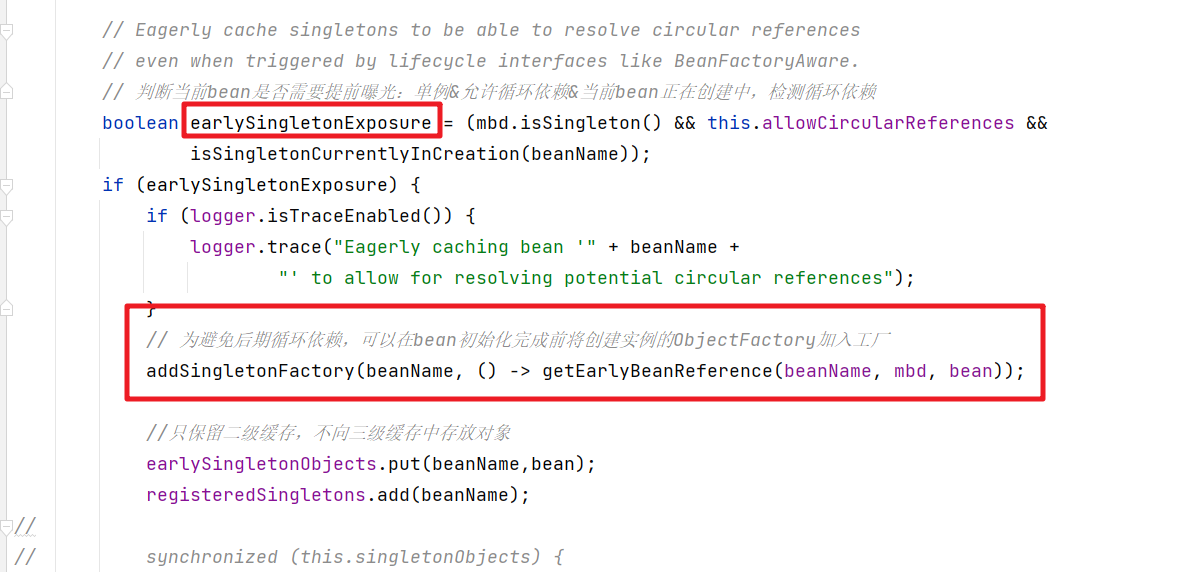


对应到源码中具体处理循环依赖的流程如下：



  上面就是在Spring的生命周期方法中和循环依赖出现相关的流程了。那么源码中的具体处理是怎么样的呢？我们继续往下面看。

首先在调用构造方法的后会放入到三级缓存中



下面就是放入三级缓存的逻辑

protected void addSingletonFactory(String beanName, ObjectFactory<?> singletonFactory) {

Assert.notNull(singletonFactory, "Singleton factory must not be null");

// 使用singletonObjects进行加锁，保证线程安全

synchronized (this.singletonObjects) {

// 如果单例对象的高速缓存【beam名称-bean实例】没有beanName的对象

if (!this.singletonObjects.containsKey(beanName)) {

// 将beanName,singletonFactory放到单例工厂的缓存【bean名称 - ObjectFactory】

this.singletonFactories.put(beanName, singletonFactory);

// 从早期单例对象的高速缓存【bean名称-bean实例】 移除beanName的相关缓存对象

this.earlySingletonObjects.remove(beanName);

// 将beanName添加已注册的单例集中

this.registeredSingletons.add(beanName);

}

}

}

然后在填充属性的时候会存入二级缓存中

earlySingletonObjects.put(beanName,bean);

registeredSingletons.add(beanName);

最后把创建的对象保存在了一级缓存中

protected void addSingleton(String beanName, Object singletonObject) {

synchronized (this.singletonObjects) {

// 将映射关系添加到单例对象的高速缓存中

this.singletonObjects.put(beanName, singletonObject);

// 移除beanName在单例工厂缓存中的数据

this.singletonFactories.remove(beanName);

// 移除beanName在早期单例对象的高速缓存的数据

this.earlySingletonObjects.remove(beanName);

// 将beanName添加到已注册的单例集中

this.registeredSingletons.add(beanName);

}

}

## **5.6 疑问点**

这些疑问点也是面试官喜欢问的问题点

### **为什么需要三级缓存**

三级缓存主要处理的是AOP的代理对象，存储的是一个ObjectFactory

三级缓存考虑的是带你对象，而二级缓存考虑的是性能-从三级缓存的工厂里创建出对象，再扔到二级缓存（这样就不用每次都要从工厂里拿）

### **没有三级环境能解决吗？**

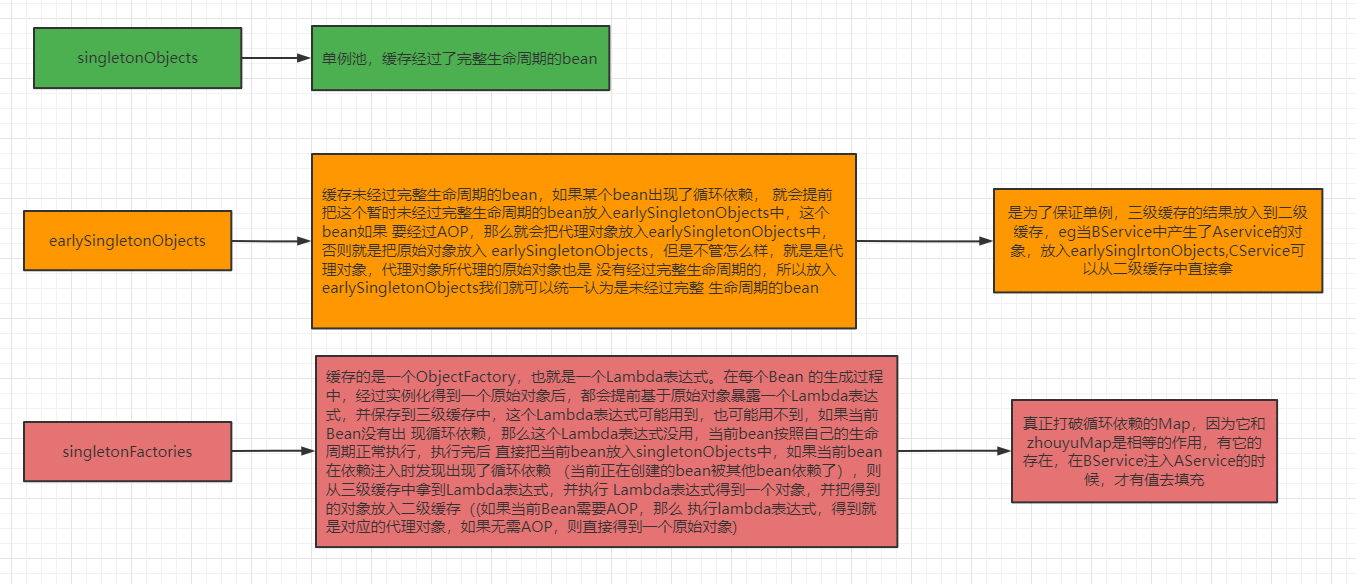
没有三级缓存是可以解决循环依赖问题的

### **三级缓存分别什么作用**

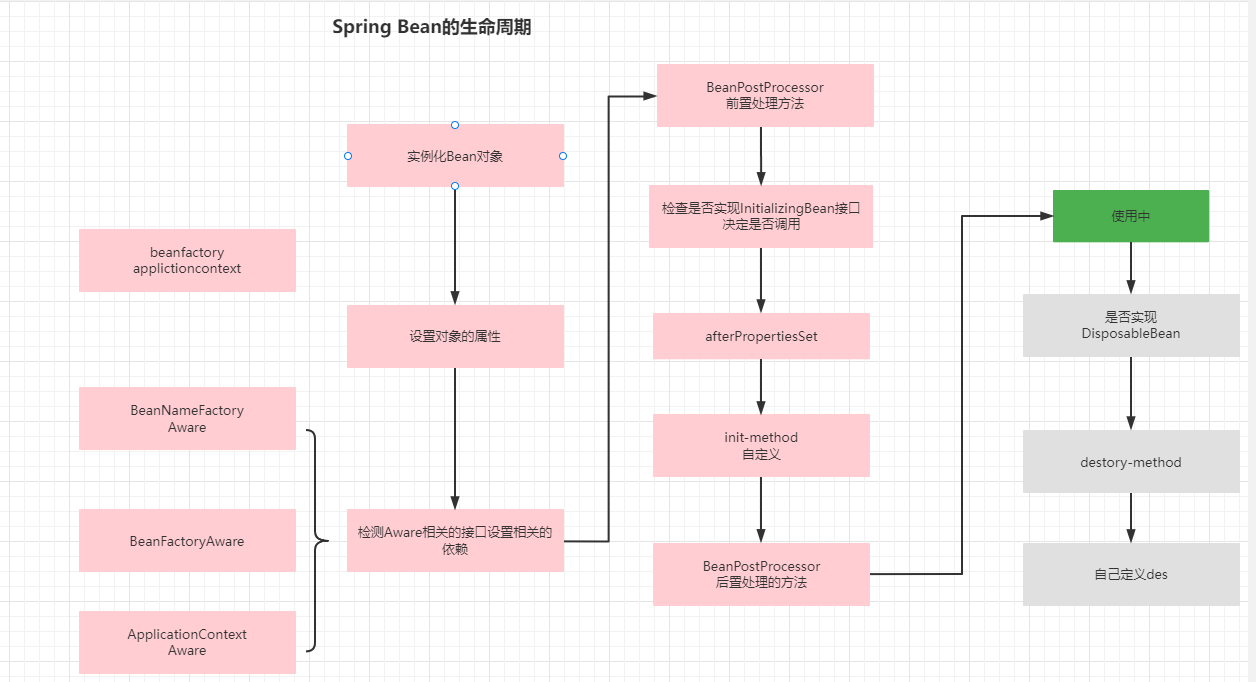
一级缓存：正式对象

二级缓存：半成品对象

三级缓存：工厂



# **6.Spring的生命周期**



结合图，把Bean对象在Spring中的关键节点介绍一遍

# **7.Spring中支持几种作用域**

Spring容器中的bean可以分为5个范围：

1. prototype：为每一个bean请求提供一个实例。
2. singleton：默认，每个容器中只有一个bean的实例，单例的模式由BeanFactory自身来维护。
3. request：为每一个网络请求创建一个实例，在请求完成以后，bean会失效并被垃圾回收器回收。
4. session：与request范围类似，确保每个session中有一个bean的实例，在session过期后，bean会随之失效。
5. global-session：全局作用域，global-session和Portlet应用相关。当你的应用部署在Portlet容器中工作时，它包含很多portlet。如果你想要声明让所有的portlet共用全局的存储变量的话，那么这全局变量需要存储在global-session中。全局作用域与Servlet中的session作用域效果相同。

# **8.说说事务的隔离级别**

  事务隔离级别指的是一个事务对数据的修改与另一个并行的事务的隔离程度，当多个事务同时访问相同数据时，如果没有采取必要的隔离机制，就可能发生以下问题：

| **问题** | **描述** |
| --- | --- |
| 脏读 | 一个事务读到另一个事务未提交的更新数据，所谓脏读，就是指事务A读到了事务B还没有提交的数据，比如银行取钱，事务A开启事务，此时切换到事务B，事务B开启事务-->取走100元，此时切换回事务A，事务A读取的肯定是数据库里面的原始数据，因为事务B取走了100块钱，并没有提交，数据库里面的账务余额肯定还是原始余额，这就是脏读 |
| 幻读 | 是指当事务不是独立执行时发生的一种现象，例如第一个事务对一个表中的数据进行了修改，这种修改涉及到表中的全部数据行。 同时，第二个事务也修改这个表中的数据，这种修改是向表中插入一行新数据。那么，以后就会发生操作第一个事务的用户发现表中还有没有修改的数据行，就好象 发生了幻觉一样。 |
| 不可重复读 | 在一个事务里面的操作中发现了未被操作的数据 比方说在同一个事务中先后执行两条一模一样的select语句，期间在此次事务中没有执行过任何DDL语句，但先后得到的结果不一致，这就是不可重复读 |

**Spring支持的隔离级别**

| **隔离级别** | **描述** |
| --- | --- |
| DEFAULT | 使用数据库本身使用的隔离级别 `` ORACLE（读已提交） MySQL（可重复读） |
| READ\_UNCOMITTED | 读未提交（脏读）最低的隔离级别，一切皆有可能。 |
| READ\_COMMITED | 读已提交，ORACLE默认隔离级别，有幻读以及不可重复读风险。 |
| REPEATABLE\_READ | 可重复读，解决不可重复读的隔离级别，但还是有幻读风险。 |
| SERLALIZABLE | 串行化，最高的事务隔离级别，不管多少事务，挨个运行完一个事务的所有子事务之后才可以执行另外一个事务里面的所有子事务，这样就解决了脏读、不可重复读和幻读的问题了 |



  再必须强调一遍，不是事务隔离级别设置得越高越好，事务隔离级别设置得越高，意味着势必要花手段去加锁用以保证事务的正确性，那么效率就要降低，因此实际开发中往往要在效率和并发正确性之间做一个取舍，一般情况下会设置为READ\_COMMITED，此时避免了脏读，并发性也还不错，之后再通过一些别的手段去解决不可重复读和幻读的问题就好了。

# **9.事务的传播行为**

Spring中的7个事务传播行为:

| **事务行为** | **说明** |
| --- | --- |
| PROPAGATION\_REQUIRED | 支持当前事务，假设当前没有事务。就新建一个事务 |
| PROPAGATION\_SUPPORTS | 支持当前事务，假设当前没有事务，就以非事务方式运行 |
| PROPAGATION\_MANDATORY | 支持当前事务，假设当前没有事务，就抛出异常 |
| PROPAGATION\_REQUIRES\_NEW | 新建事务，假设当前存在事务。把当前事务挂起 |
| PROPAGATION\_NOT\_SUPPORTED | 以非事务方式运行操作。假设当前存在事务，就把当前事务挂起 |
| PROPAGATION\_NEVER | 以非事务方式运行，假设当前存在事务，则抛出异常 |
| PROPAGATION\_NESTED | 如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION\_REQUIRED类似的操作。 |

**举例说明**

**案例代码**

ServiceA

ServiceA {

void methodA() {

ServiceB.methodB();

}

}

ServiceB

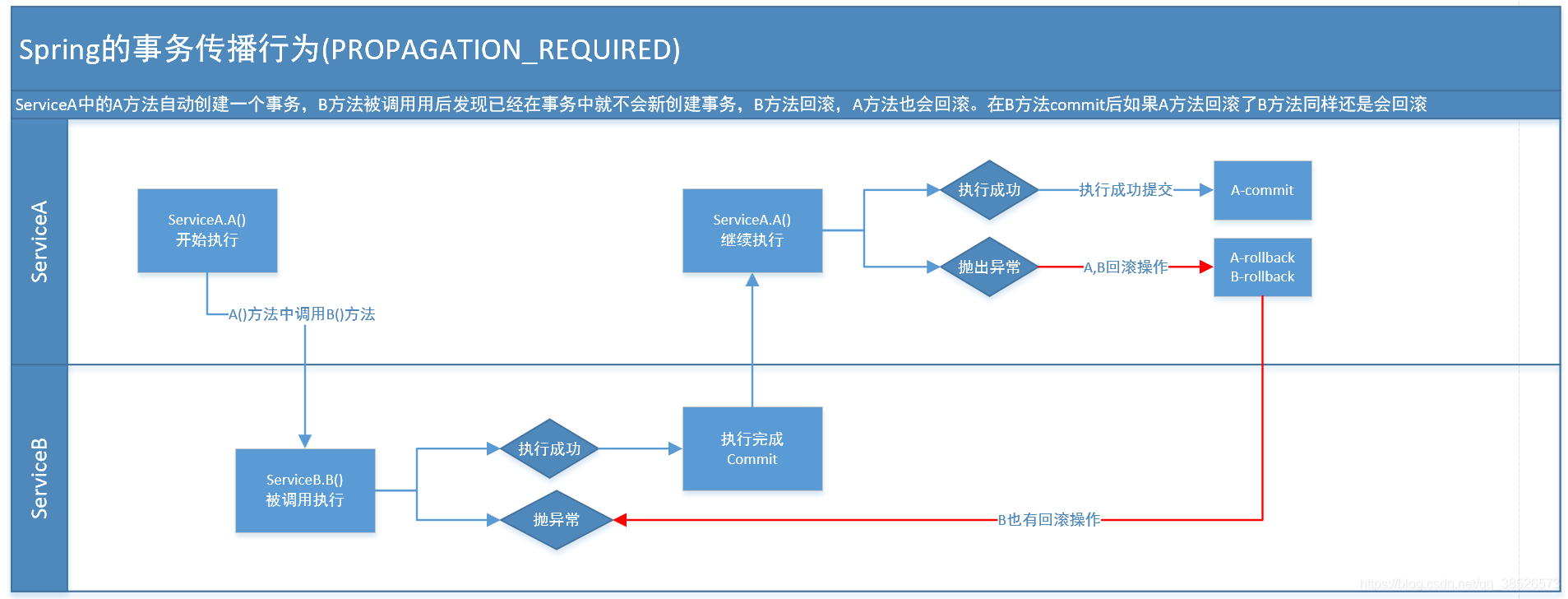
ServiceB {

void methodB() {

}

}

### **1.PROPAGATION\_REQUIRED**

  假如当前正要运行的事务不在另外一个事务里，那么就起一个新的事务 比方说，ServiceB.methodB的事务级别定义PROPAGATION\_REQUIRED, 那么因为执行ServiceA.methodA的时候，ServiceA.methodA已经起了事务。这时调用ServiceB.methodB，ServiceB.methodB看到自己已经执行在ServiceA.methodA的事务内部。就不再起新的事务。而假如ServiceA.methodA执行的时候发现自己没有在事务中，他就会为自己分配一个事务。这样，在ServiceA.methodA或者在ServiceB.methodB内的不论什么地方出现异常。事务都会被回滚。即使ServiceB.methodB的事务已经被提交，可是ServiceA.methodA在接下来fail要回滚，ServiceB.methodB也要回滚  


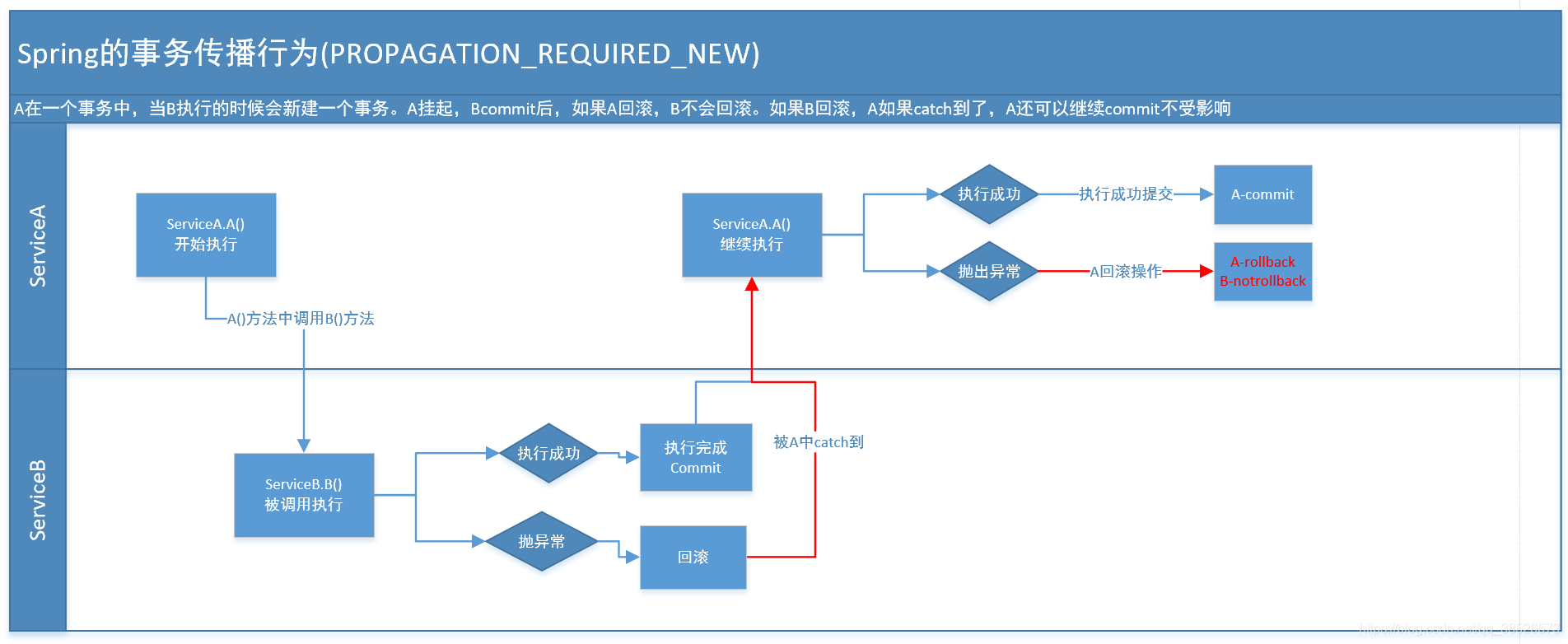
### **2.PROPAGATION\_SUPPORTS**

  假设当前在事务中。即以事务的形式执行。假设当前不在一个事务中，那么就以非事务的形式执行

### **3PROPAGATION\_MANDATORY**

  必须在一个事务中执行。也就是说，他仅仅能被一个父事务调用。否则，他就要抛出异常

### **4.PROPAGATION\_REQUIRES\_NEW**

  这个就比较绕口了。 比方我们设计ServiceA.methodA的事务级别为PROPAGATION\_REQUIRED，ServiceB.methodB的事务级别为PROPAGATION\_REQUIRES\_NEW。那么当运行到ServiceB.methodB的时候，ServiceA.methodA所在的事务就会挂起。ServiceB.methodB会起一个新的事务。等待ServiceB.methodB的事务完毕以后，他才继续运行。  
他与PROPAGATION\_REQUIRED 的事务差别在于事务的回滚程度了。由于ServiceB.methodB是新起一个事务，那么就是存在两个不同的事务。假设ServiceB.methodB已经提交，那么ServiceA.methodA失败回滚。ServiceB.methodB是不会回滚的。假设ServiceB.methodB失败回滚，假设他抛出的异常被ServiceA.methodA捕获，ServiceA.methodA事务仍然可能提交。  


### **5.PROPAGATION\_NOT\_SUPPORTED**

  当前不支持事务。比方ServiceA.methodA的事务级别是PROPAGATION\_REQUIRED 。而ServiceB.methodB的事务级别是PROPAGATION\_NOT\_SUPPORTED ，那么当执行到ServiceB.methodB时。ServiceA.methodA的事务挂起。而他以非事务的状态执行完，再继续ServiceA.methodA的事务。

### **6.PROPAGATION\_NEVER**

  不能在事务中执行。  
如果ServiceA.methodA的事务级别是PROPAGATION\_REQUIRED。 而ServiceB.methodB的事务级别是PROPAGATION\_NEVER ，那么ServiceB.methodB就要抛出异常了。

### **7.PROPAGATION\_NESTED**

  如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION\_REQUIRED类似的操作。

# **10.Spring事务实现的方式**

**编程式事务管理**：这意味着你可以通过编程的方式管理事务，这种方式带来了很大的灵活性，但很难维护。

**声明式事务管理**：这种方式意味着你可以将事务管理和业务代码分离。你只需要通过注解或者XML配置管理事务。

# **11.事务注解的本质是什么**

  @Transactional 这个注解仅仅是一些（和事务相关的）元数据，在运行时被事务基础设施读取消费，并使用这些元数据来配置bean的事务行为。 大致来说具有两方面功能，一是表明该方法要参与事务，二是配置相关属性来定制事务的参与方式和运行行为

  声明式事务主要是得益于Spring AOP。使用一个事务拦截器，在方法调用的前后/周围进行事务性增强（advice），来驱动事务完成。

  @Transactional注解既可以标注在类上，也可以标注在方法上。当在类上时，默认应用到类里的所有方法。如果此时方法上也标注了，则方法上的优先级高。 另外注意方法一定要是public的。