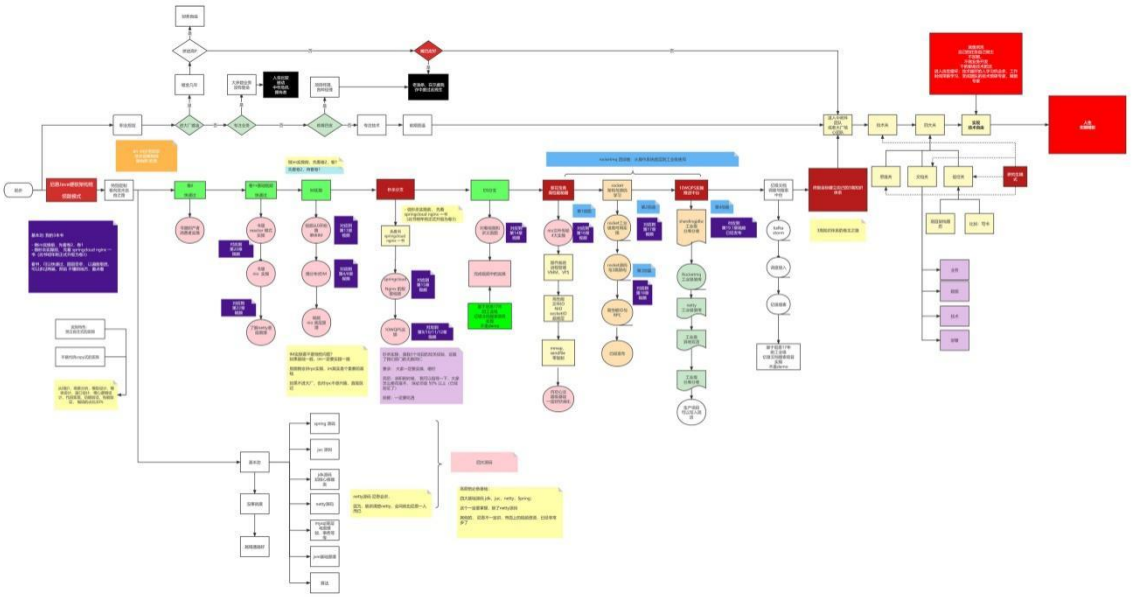
牛逼的职业发展之路

40 岁老架构尼恩用一张图揭秘：Java 工程师的高端职业发展路径，走向食物链顶端的之路

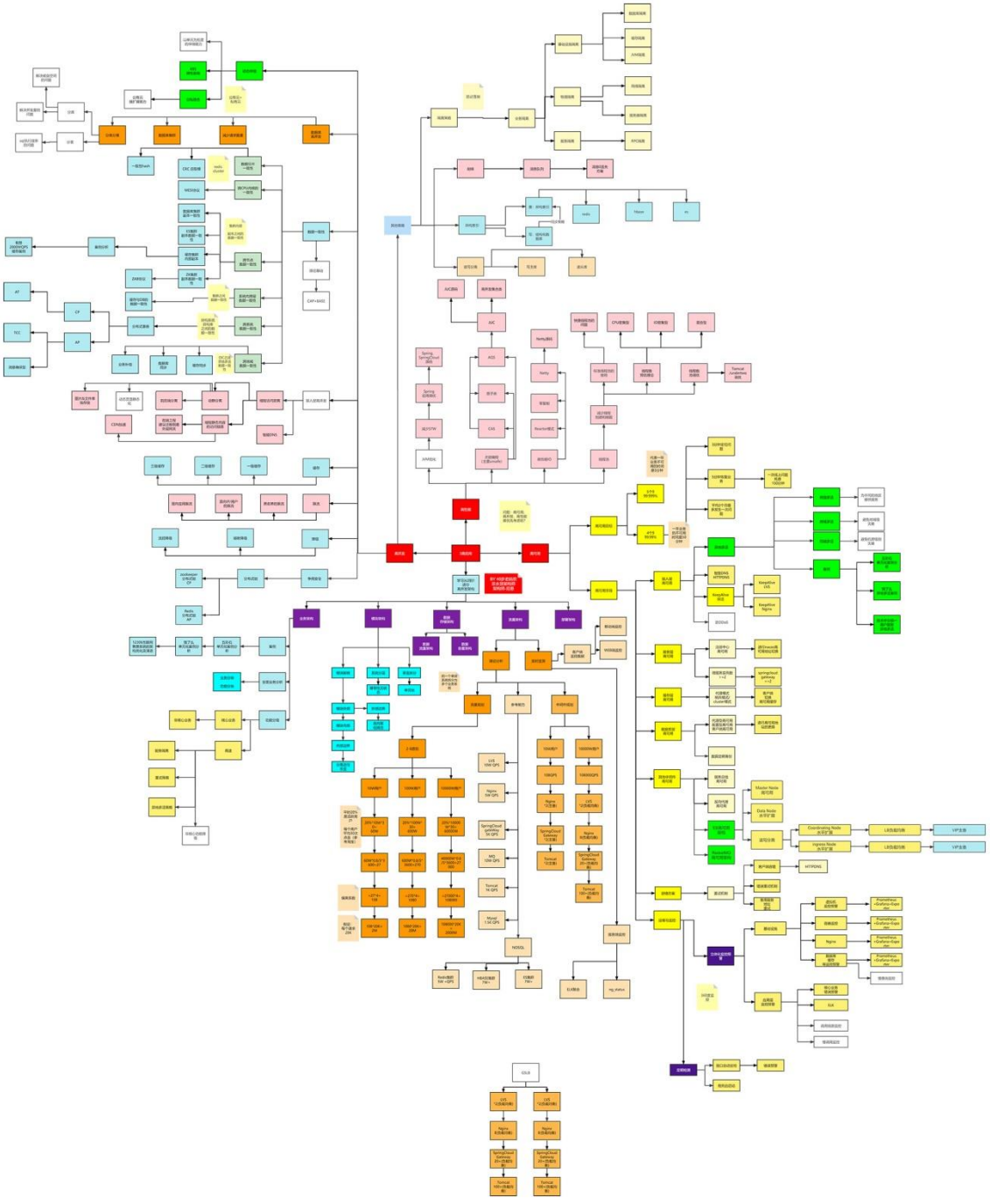
链接：https://www.processon.com/view/link/618a2b62e0b34d73f7eb3cd7



史上最全：价值10W的架构师知识图谱

此图梳理于尼恩的多个 3 高生产项目：多个亿级人民币的大型 SAAS 平台和智慧城市项目

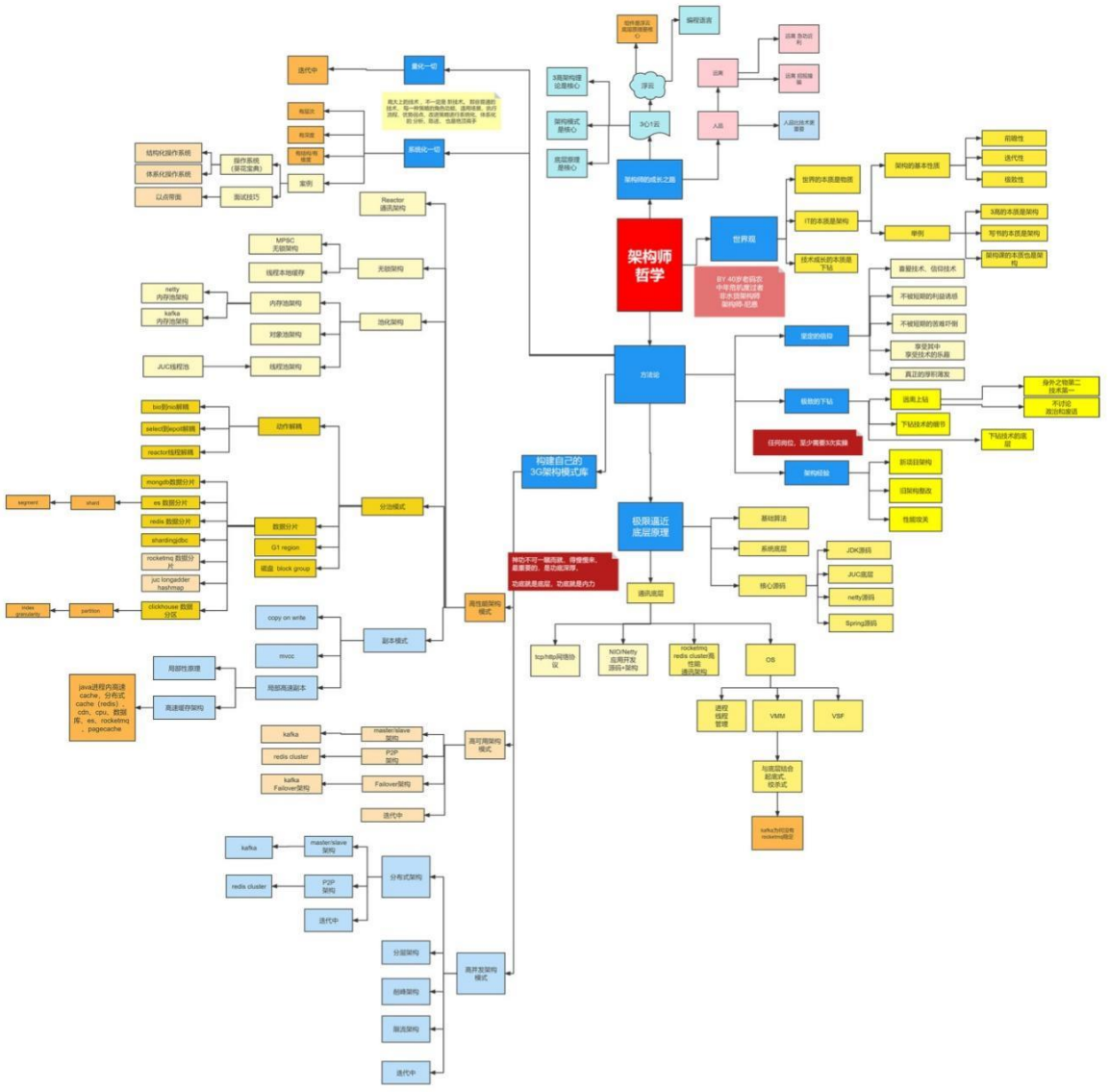
链接：https://www.processon.com/view/link/60fb94216376897l9d246739



牛逼的架构师哲学

40 岁老架构师尼恩对自己的 20 年的开发、架构经验总结

链接：https://www.processon.com/view/link/616f801963768961e9d9aec8



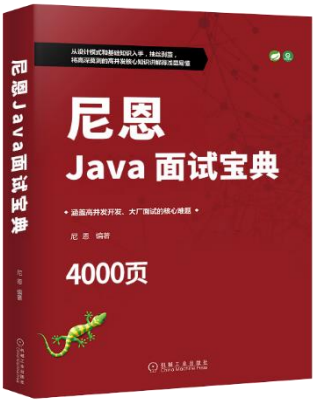
牛逼的3高架构知识宇宙

尼恩 3 高架构知识宇宙，帮助大家穿透 3 高架构，走向技术自由，远离中年危机

链接：https://www.processon.com/view/link/635097d2e0b34d40be778ab4

尼恩Java面试宝典

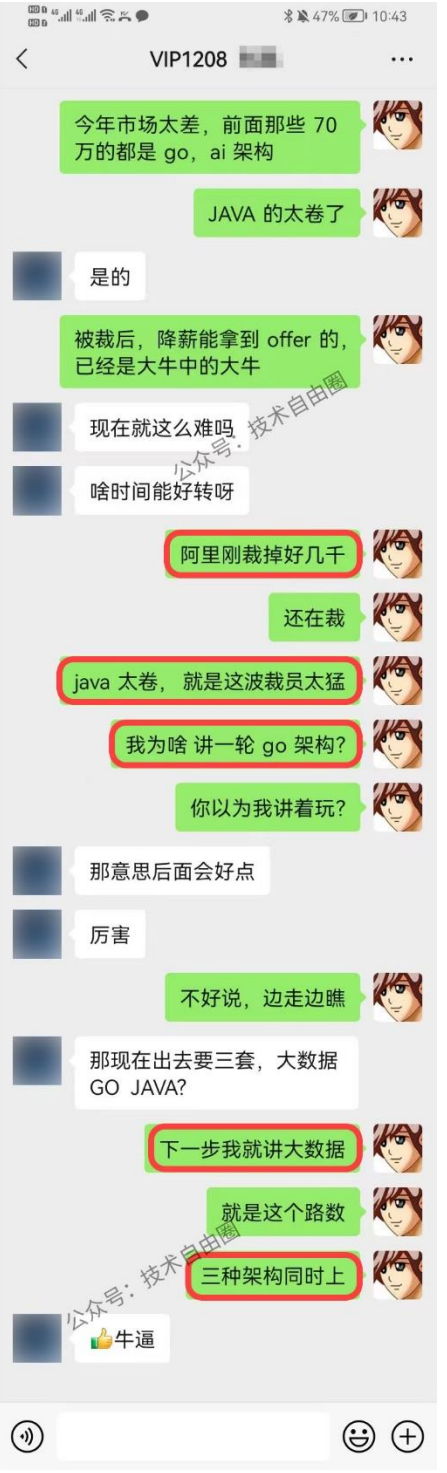
40 个专题（卷王专供+ 史上最全 + 2023 面试必备）
详情：https://www.cnblogs.com/crazymakercircle/p/13917138.html



名称

专题01：JVM面试题（卷王专供 + 史上最全 + 2022面试必备）-V81-from-尼恩Java面试宝典.pdf
专题02：Java算法面试题（卷王专供 + 史上最全 + 2022面试必备）-V80-from-Java面试红宝书.pdf
专题03：Java基础面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书 -release.pdf
专题04：架构设计面试题（卷王专供+ 史上最全 + 2023面试必备）-V86-from-尼恩Java面试宝典.pdf
专题05：Spring面试题_专题06：SpringMVC_专题07：Tomcat面试题（卷王专供+ 史上最全 + 2023面试必备）-V3-from-尼恩面试宝典-release.pdf
专题08：SpringBoot面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题09：网络协议面试题（卷王专供+ 史上最全 + 2023面试必备）-V46-from-尼恩Java面试宝典-release.pdf
专题10：TCPIP协议（卷王专供+ 史上最全 + 2022面试必备）-V57-from-Java面试红宝书.pdf
专题11：JUC并发包与容器类（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题12：设计模式面试题（卷王专供+ 史上最全 + 2022面试必备）-V84-from-Java面试红宝书.pdf
专题13：死锁面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题14：Redis 面试题 （卷王专供+ 史上最全 + 2022面试必备）-V65-from-Java面试红宝书.pdf
专题15：分布式锁 面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题16：Zookeeper 面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题17：分布式事务面试题（卷王专供 + 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题18：一致性协议 （卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题19：Zab协议（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题20：Paxos 协议（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题21：raft 协议（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题22：Linux面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题23：Mysql 面试题（卷王专供+ 史上最全 + 2023面试必备）-V82-from-尼恩Java面试宝典.pdf
专题24：SpringCloud 面试题（卷王专供+ 史上最全 + 2023面试必备）-V12-from-Java面试红宝书-release.pdf
专题25：Netty 面试题 （卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题26：消息队列面试题：RabbitMQ、Kafka、RocketMQ（卷王专供+ 史上最全 + 2023面试必备）-V10-from-Java面试红宝书-release.pdf
专题27：内存泄漏 内存溢出（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题28：JVM 内存溢出 实战（卷王专供+ 史上最全 + 2023面试必备）-V17-from-Java面试红宝书-release.pdf
专题29：多线程面试题（卷王专供+ 史上最全 + 2023面试必备）-V66-from-Java面试红宝书.pdf
专题30：HR面试题：过五关斩六将后，小心阴沟翻船！（史上最全、避坑宝典）-V2-from-Java面试红宝书-release.pdf
专题31：Hash连环炮面试题（卷王专供+ 史上最全 + 2022面试必备）-V68-from-Java面试红宝书.pdf
专题32：大厂面试的基本流程和面试准备（阿里、腾讯、网易、京东、头条.....）-V2-from-Java面试红宝书-release.pdf
专题33：BST、AVL、RBT红黑树、三大核心数据结构（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题34：Elasticsearch面试题（卷王专供+ 史上最全 + 2023面试必备）-V3-from-Java面试红宝书-release.pdf
专题35：Mybatis面试题（卷王专供+ 史上最全 + 2023面试必备）-V3-from-尼恩Java面试宝典-release.pdf
专题36：Dubbo面试题（卷王专供+ 史上最全 + 2023面试必备）-V21-from-尼恩Java面试宝典-release.pdf
专题37：Docker面试题（卷王专供+ 史上最全 + 2023面试必备）-V47-from-尼恩Java面试宝典.pdf
专题38：K8S面试题（卷王专供+ 史上最全 + 2023面试必备）-V59-from-尼恩Java面试宝典.pdf
专题39：Nginx面试题（卷王专供+ 史上最全 + 2023面试必备）-V27-from-尼恩Java面试宝典-release.pdf
专题40：操作系统面试题（卷王专供+ 史上最全 + 2023面试必备）-V28-from-尼恩Java面试宝典-release.pdf
专题41：大厂面试真题（卷王专供+ 史上最全 + 2023面试必备）-V84-from-尼恩Java面试宝典.pdf

未来职业，如何突围：三栖架构师



专题5： Spring面试题（史上最全、定期更新）

本文版本说明：V77

> 尼恩面试宝典 ，早期叫做 《Java面试红宝书》
>
> 此文的格式，由markdown 通过程序转成而来，由于很多表格，没有来的及调整，出现一个格式问题，尼恩在此给大家道歉啦。
>
> 由于社群很多小伙伴，在面试，不断的交流最新的面试难题，所以，《尼恩Java面试宝典》， 后面会不断升级，迭代。
>
> 本专题，作为 《尼恩Java面试宝典》的第10个专题， 《尼恩Java面试宝典》一共30个面试专题。

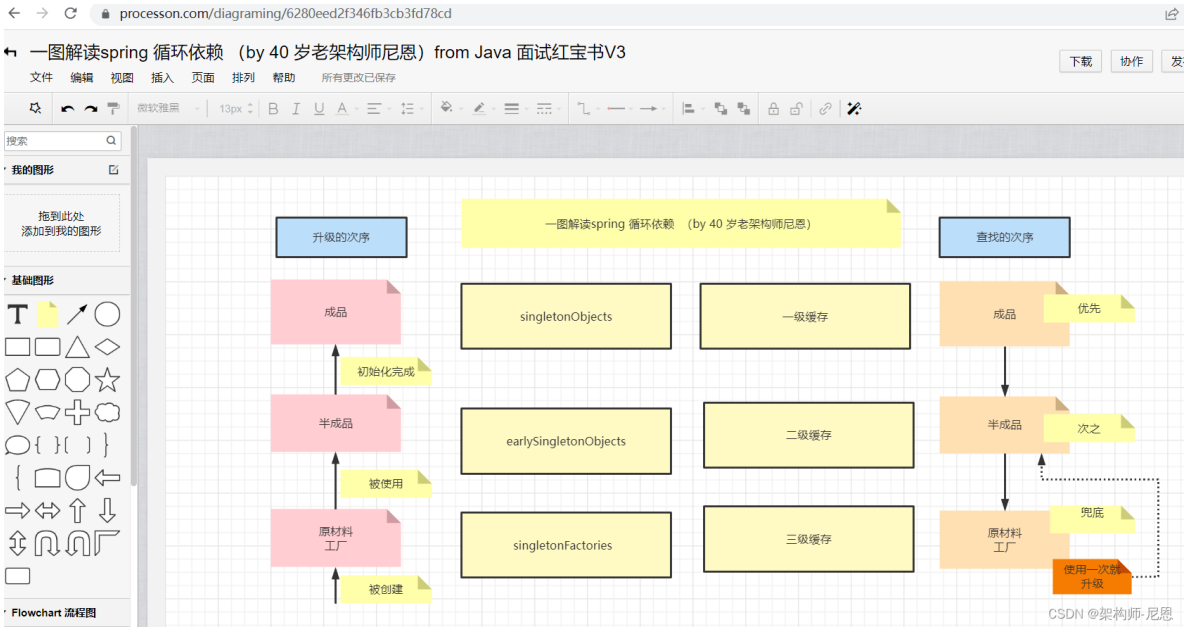
升级说明：

V77升级说明（2023-6-16）：

大厂面试题：读过Spring源码吗？说说Spring事务是怎么实现的？

V3升级说明（2022-5-16）：

对spring三级缓存， **使用成品、半成品、原材料工厂，这样的浅显易懂的比方**， 使得复杂的概念 ， 变得更容易好懂



《尼恩面试宝典》升级的规划为：

后续基本上，**每一个月，都会发布一次**，最新版本，可以扫描扫架构师尼恩微信， 发送 “领取电子书” 获取。

尼恩的微信二维码在哪里呢？ 请参见文末

面试问题交流说明：

如果遇到面试难题，或者职业发展问题， 或者中年危机问题，都可以来 疯狂创客圈社群交流，

加入交流群，加尼恩微信即可，

尼恩的微信二维码在哪里呢？ 具体可以百度搜索 **疯狂创客圈 总目录**

什么是spring?

Spring是**一个轻量级Java开发框架**，最早有**Rod Johnson**创建，目的是为了解决企业级应用开发的业务逻辑层和其他各层的耦合问题。它是一个分层的JavaSE/JavaEE full-stack（一站式）轻量级开源框架，为开发Java应用程序提供全面的基础架构支持。Spring负责基础架构，因此Java开发者可以专注于应用程序的开发。

Spring最根本的使命是**解决企业级应用开发的复杂性，即简化Java开发。**

Spring可以做很多事情，它为企业级开发提供给了丰富的功能，但是这些功能的底层都依赖于它的两个核心特性，也就是**依赖注入（dependency injection，DI）和面向切面编程（aspect-oriented programming，AOP）。**

为了降低Java开发的复杂性，Spring采取了以下4种关键策略

- 基于POJO的轻量级和最小侵入性编程；
- 通过依赖注入和面向接口实现松耦合；
- 基于切面和惯例进行声明式编程；
- 通过切面和模板减少样板式代码。

Spring框架的设计目标，设计理念，和核心是什么

Spring设计目标：Spring为开发者提供一个一站式轻量级应用开发平台；

Spring设计理念：在JavaEE开发中，支持POJO和JavaBean开发方式，使应用面向接口开发，充分支持OO（面向对象）设计方法；Spring通过IoC容器实现对象耦合关系的管理，并实现依赖反转，将对象之间的依赖关系交给IoC容器，实现解耦；

Spring框架的核心：IoC容器和AOP模块。通过IoC容器管理POJO对象以及他们之间的耦合关系；通过AOP以动态非侵入的方式增强服务。

IoC让相互协作的组件保持松散的耦合，而AOP编程允许你把遍布于应用各层的功能分离出来形成可重用的功能组件。

Spring的优缺点是什么?

优点

- 方便解耦，简化开发

  Spring就是一个大工厂，可以将所有对象的创建和依赖关系的维护，交给Spring管理。

- AOP编程的支持

  Spring提供面向切面编程，可以方便的实现对程序进行权限拦截、运行监控等功能。

- 声明式事务的支持

  只需要通过配置就可以完成对事务的管理，而无需手动编程。

- 方便程序的测试

  Spring对Junit4支持，可以通过注解方便的测试Spring程序。

- 方便集成各种优秀框架

  Spring不排斥各种优秀的开源框架，其内部提供了对各种优秀框架的直接支持（如：Struts、Hibernate、MyBatis等）。

- 降低JavaEE API的使用难度

  Spring对JavaEE开发中非常难用的一些API（JDBC、JavaMail、远程调用等），都提供了封装，使这些API应用难度大大降低。

缺点

- Spring明明一个很轻量级的框架，却给人感觉大而全
- Spring依赖反射，反射影响性能
- 使用门槛升高，入门Spring需要较长时间

Spring有哪些应用场景

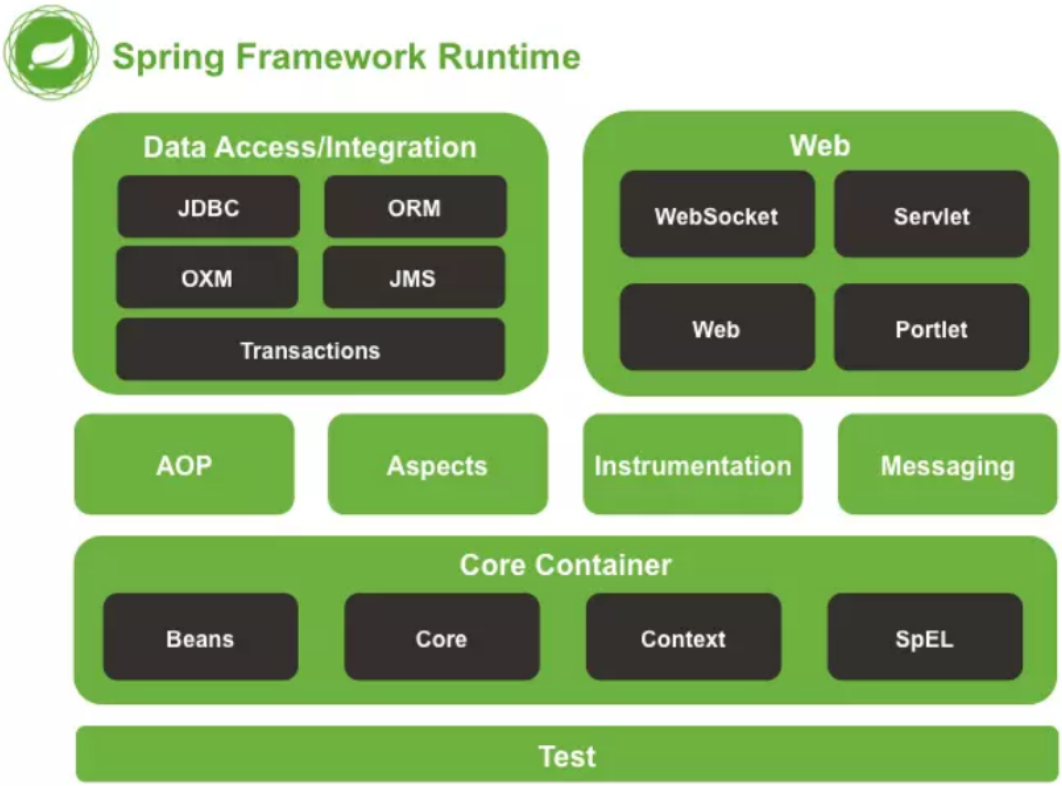
应用场景：JavaEE企业应用开发，包括SSH、SSM等

Spring价值：

- Spring是非侵入式的框架，目标是使应用程序代码对框架依赖最小化；
- Spring提供一个一致的编程模型，使应用直接使用POJO开发，与运行环境隔离开来；
- Spring推动应用设计风格向面向对象和面向接口开发转变，提高了代码的重用性和可测试性；

Spring由哪些模块组成?

Spring 总共大约有 20 个模块， 由 1300 多个不同的文件构成。 而这些组件被分别整合在`核心容器（Core Container）`、 `AOP（Aspect Oriented Programming）和设备支持（Instrmentation）`、`数据访问与集成（Data Access/Integeration）`、 `Web`、 `消息（Messaging）`、 `Test`等 6 个模块中。 以下是 Spring 5 的模块结构图：



- spring core：提供了框架的基本组成部分，包括控制反转（Inversion of Control，IOC）和依赖注入（Dependency Injection，DI）功能。
- spring beans：提供了BeanFactory，是工厂模式的一个经典实现，Spring将管理对象称为Bean。
- spring context：构建于 core 封装包基础上的 context 封装包，提供了一种框架式的对象访问方法。
- spring jdbc：提供了一个JDBC的抽象层，消除了烦琐的JDBC编码和数据库厂商特有的错误代码解析， 用于简化JDBC。
- spring aop：提供了面向切面的编程实现，让你可以自定义拦截器、切点等。
- spring Web：提供了针对 Web 开发的集成特性，例如文件上传，利用 servlet listeners 进行 ioc 容器初始化和针对 Web 的 ApplicationContext。
- spring test：主要为测试提供支持的，支持使用JUnit或TestNG对Spring组件进行单元测试和集成测试。

Spring 框架中都用到了哪些设计模式?

1. 工厂模式：BeanFactory就是简单工厂模式的体现，用来创建对象的实例；
2. 单例模式：Bean默认为单例模式。
3. 代理模式：Spring的AOP功能用到了JDK的动态代理和CGLIB字节码生成技术；
4. 模板方法：用来解决代码重复的问题。比如. RestTemplate, JmsTemplate, JpaTemplate。
5. 观察者模式：定义对象键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会得到通知被制动更新，如Spring中listener的实现–ApplicationListener。

详细讲解一下核心容器（spring context应用上下文) 模块

这是基本的Spring模块，提供spring 框架的基础功能，BeanFactory 是 任何以spring为基础的应用的核心。Spring 框架建立在此模块之上，它使Spring成为一个容器。

Bean 工厂是工厂模式的一个实现，提供了控制反转功能，用来把应用的配置和依赖从真正的应用代码中分离。最常用的就是org.springframework.beans.factory.xml.XmlBeanFactory ，它根据XML文件中的定义加载beans。该容器从XML 文件读取配置元数据并用它去创建一个完全配置的系统或应用。

Spring框架中有哪些不同类型的事件

Spring 提供了以下5种标准的事件：

1. 上下文更新事件（ContextRefreshedEvent）：在调用ConfigurableApplicationContext 接口中的refresh()方法时被触发。
2. 上下文开始事件（ContextStartedEvent）：当容器调用ConfigurableApplicationContext的Start()方法开始/重新开始容器时触发该事件。
3. 上下文停止事件（ContextStoppedEvent）：当容器调用ConfigurableApplicationContext的Stop()方法停止容器时触发该事件。
4. 上下文关闭事件（ContextClosedEvent）：当ApplicationContext被关闭时触发该事件。容器被关闭时，其管理的所有单例Bean都被销毁。
5. 请求处理事件（RequestHandledEvent）：在Web应用中，当一个http请求（request）结束触发该事件。如果一个bean实现了ApplicationListener接口，当一个ApplicationEvent 被发布以后，bean会自动被通知。

Spring 应用程序有哪些不同组件?

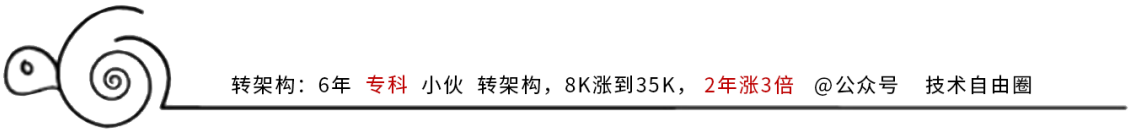
Spring 应用一般有以下组件：

- 接口 - 定义功能。
- Bean 类 - 它包含属性，setter 和 getter 方法，函数等。
- Bean 配置文件 - 包含类的信息以及如何配置它们。
- Spring 面向切面编程（AOP） - 提供面向切面编程的功能。
- 用户程序 - 它使用接口。

使用 Spring 有哪些方式?

使用 Spring 有以下方式：

- 作为一个成熟的 Spring Web 应用程序。
- 作为第三方 Web 框架，使用 Spring Frameworks 中间层。
- 作为企业级 Java Bean，它可以包装现有的 POJO（Plain Old Java Objects）。
- 用于远程使用。



Spring控制反转(IOC)（13）

什么是Spring IOC 容器?

控制反转即IoC (Inversion of Control)，它把传统上由程序代码直接操控的对象的调用权交给容器，通过容器来实现对象组件的装配和管理。所谓的"控制反转"概念就是对组件对象控制权的转移，从程序代码本身转移到了外部容器。

Spring IOC 负责创建对象，管理对象（通过依赖注入（DI），装配对象，配置对象，并且管理这些对象的整个生命周期。

控制反转(IoC)有什么作用

- 管理对象的创建和依赖关系的维护。对象的创建并不是一件简单的事，在对象关系比较复杂时，如果依赖关系需要程序猿来维护的话，那是相当头疼的
- 解耦，由容器去维护具体的对象
- 托管了类的产生过程，比如我们需要在类的产生过程中做一些处理，最直接的例子就是代理，如果有容器程序可以把这部分处理交给容器，应用程序则无需去关心类是如何完成代理的

IOC的优点是什么?

- IOC 或 依赖注入把应用的代码量降到最低。
- 它使应用容易测试，单元测试不再需要单例和JNDI查找机制。
- 最小的代价和最小的侵入性使松散耦合得以实现。
- IOC容器支持加载服务时的饿汉式初始化和懒加载。

Spring IoC 的实现机制

Spring 中的 IoC 的实现原理就是工厂模式加反射机制。

示例:

```
interface Fruit {
   public abstract void eat();
 }

class Apple implements Fruit {
    public void eat(){
        System.out.println("Apple");
    }
}

class Orange implements Fruit {
    public void eat(){
        System.out.println("Orange");
    }
}

class Factory {
    public static Fruit getInstance(String ClassName) {
        Fruit f=null;
        try {
            f=(Fruit)Class.forName(ClassName).newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return f;
    }
}

class Client {
    public static void main(String[] a) {
        Fruit f=Factory.getInstance("io.github.dunwu.spring.Apple");
        if(f!=null){
            f.eat();
        }
    }
}
```

Spring 的 IoC支持哪些功能

Spring 的 IoC 设计支持以下功能：

- 依赖注入
- 依赖检查
- 自动装配
- 支持集合
- 指定初始化方法和销毁方法
- 支持回调某些方法（但是需要实现 Spring 接口，略有侵入）

其中，最重要的就是依赖注入，从 XML 的配置上说，即 ref 标签。对应 Spring RuntimeBeanReference 对象。

对于 IoC 来说，最重要的就是容器。容器管理着 Bean 的生命周期，控制着 Bean 的依赖注入。

BeanFactory 和 ApplicationContext有什么区别?

BeanFactory和ApplicationContext是Spring的两大核心接口，都可以当做Spring的容器。

其中ApplicationContext是BeanFactory的子接口。

依赖关系

BeanFactory：是Spring里面最底层的接口，包含了各种Bean的定义，读取bean配置文档，管理bean的加载、实例化，控制bean的生命周期，维护bean之间的依赖关系。

ApplicationContext接口作为BeanFactory的派生，除了提供BeanFactory所具有的功能外，还提供了更完整的框架功能：

- 继承MessageSource，因此支持国际化。
- 统一的资源文件访问方式。
- 提供在监听器中注册bean的事件。
- 同时加载多个配置文件。
- 载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的web层。

加载方式

BeanFactroy采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化。这样，我们就不能发现一些存在的Spring的配置问题。如果Bean的某一个属性没有注入，BeanFacotry加载后，直至第一次使用调用getBean方法才会抛出异常。

ApplicationContext，它是在容器启动时，一次性创建了所有的Bean。这样，在容器启动时，我们就可以发现Spring中存在的配置错误，这样有利于检查所依赖属性是否注入。 ApplicationContext启动后预载入所有的单实例Bean，通过预载入单实例bean ,确保当你需要的时候，你就不用等待，因为它们已经创建好了。

相对于基本的BeanFactory，ApplicationContext 唯一的不足是占用内存空间。当应用程序配置Bean较多时，程序启动较慢。

创建方式

BeanFactory通常以编程的方式被创建，ApplicationContext还能以声明的方式创建，如使用ContextLoader。

注册方式

BeanFactory和ApplicationContext都支持BeanPostProcessor、BeanFactoryPostProcessor的使用，但两者之间的区别是：BeanFactory需要手动注册，而ApplicationContext则是自动注册。

Spring 如何设计容器的，BeanFactory和ApplicationContext的关系详解

Spring 作者 Rod Johnson 设计了两个接口用以表示容器。

- BeanFactory
- ApplicationContext

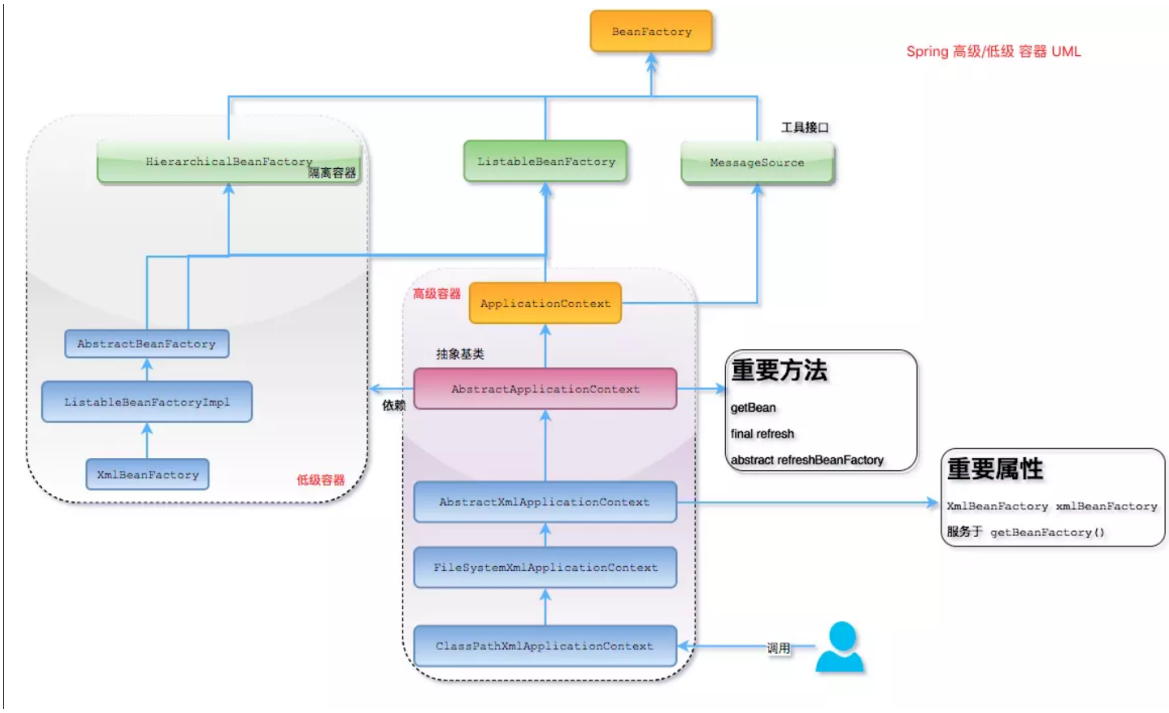
BeanFactory 简单粗暴，可以理解为就是个 HashMap，Key 是 BeanName，Value 是 Bean 实例。通常只提供注册（put），获取（get）这两个功能。我们可以称之为 **“低级容器”**。

ApplicationContext 可以称之为 **“高级容器”**。因为他比 BeanFactory 多了更多的功能。他继承了多个接口。因此具备了更多的功能。例如资源的获取，支持多种消息（例如 JSP tag 的支持），对 BeanFactory 多了工具级别的支持等待。所以你看他的名字，已经不是 BeanFactory 之类的工厂了，而是 “应用上下文”， 代表着整个大容器的所有功能。该接口定义了一个 refresh 方法，此方法是所有阅读 Spring 源码的人的最熟悉的方法，用于刷新整个容器，即重新加载/刷新所有的 bean。

当然，除了这两个大接口，还有其他的辅助接口，这里就不介绍他们了。

BeanFactory和ApplicationContext的关系

为了更直观的展示 “低级容器” 和 “高级容器” 的关系，这里通过常用的 ClassPathXmlApplicationContext 类来展示整个容器的层级 UML 关系。



有点复杂？ 先不要慌，我来解释一下。

最上面的是 BeanFactory，下面的 3 个绿色的，都是功能扩展接口，这里就不展开讲。

看下面的隶属 ApplicationContext 粉红色的 “高级容器”，依赖着 “低级容器”，这里说的是依赖，不是继承哦。他依赖着 “低级容器” 的 getBean 功能。而高级容器有更多的功能：支持不同的信息源头，可以访问文件资源，支持应用事件（Observer 模式）。

通常用户看到的就是 “高级容器”。 但 BeanFactory 也非常够用啦！

左边灰色区域的是 “低级容器”， 只负载加载 Bean，获取 Bean。容器其他的高级功能是没有的。例如上图画的 refresh 刷新 Bean 工厂所有配置，生命周期事件回调等。

小结

说了这么多，不知道你有没有理解Spring IoC？ 这里小结一下：IoC 在 Spring 里，只需要低级容器就可以实现，2 个步骤：

1. 加载配置文件，解析成 BeanDefinition 放在 Map 里。
2. 调用 getBean 的时候，从 BeanDefinition 所属的 Map 里，拿出 Class 对象进行实例化，同时，如果有依赖关系，将递归调用 getBean 方法 —— 完成依赖注入。

上面就是 Spring 低级容器（BeanFactory）的 IoC。

至于高级容器 ApplicationContext，他包含了低级容器的功能，当他执行 refresh 模板方法的时候，将刷新整个容器的 Bean。同时其作为高级容器，包含了太多的功能。一句话，他不仅仅是 IoC。他支持不同信息源头，支持 BeanFactory 工具类，支持层级容器，支持访问文件资源，支持事件发布通知，支持接口回调等等。

ApplicationContext通常的实现是什么?

FileSystemXmlApplicationContext：此容器从一个XML文件中加载beans的定义，XML Bean 配置文件的全路径名必须提供给它的构造函数。

ClassPathXmlApplicationContext：此容器也从一个XML文件中加载beans的定义，这里，你需要正确设置classpath因为这个容器将在classpath里找bean配置。

WebXmlApplicationContext：此容器加载一个XML文件，此文件定义了一个WEB应用的所有bean。

什么是Spring的依赖注入?

控制反转IoC是一个很大的概念，可以用不同的方式来实现。其主要实现方式有两种：依赖注入和依赖查找

依赖注入：相对于IoC而言，依赖注入(DI)更加准确地描述了IoC的设计理念。所谓依赖注入（Dependency Injection），即组件之间的依赖关系由容器在应用系统运行期来决定，也就是由容器动态地将某种依赖关系的目标对象实例注入到应用系统中的各个关联的组件之中。组件不做定位查询，只提供普通的Java方法让容器去决定依赖关系。

依赖注入的基本原则

依赖注入的基本原则是：应用组件不应该负责查找资源或者其他依赖的协作对象。配置对象的工作应该由IoC容器负责，“查找资源”的逻辑应该从应用组件的代码中抽取出来，交给IoC容器负责。容器全权负责组件的装配，它会把符合依赖关系的对象通过属性（JavaBean中的setter）或者是构造器传递给需要的对象。

依赖注入有什么优势

依赖注入之所以更流行是因为它是一种更可取的方式：让容器全权负责依赖查询，受管组件只需要暴露JavaBean的setter方法或者带参数的构造器或者接口，使容器可以在初始化时组装对象的依赖关系。其与依赖查找方式相比，主要优势为：

- 查找定位操作与应用代码完全无关。
- 不依赖于容器的API，可以很容易地在任何容器以外使用应用对象。
- 不需要特殊的接口，绝大多数对象可以做到完全不必依赖容器。

有哪些不同类型的依赖注入实现方式?

依赖注入是时下最流行的IoC实现方式，依赖注入分为接口注入（Interface Injection），Setter方法注入（Setter Injection）和构造器注入（Constructor Injection）三种方式。其中接口注入由于在灵活性和易用性比较差，现在从Spring4开始已被废弃。

构造器依赖注入：构造器依赖注入通过容器触发一个类的构造器来实现的，该类有一系列参数，每个参数代表一个对其他类的依赖。

Setter方法注入：Setter方法注入是容器通过调用无参构造器或无参static工厂 方法实例化bean之后，调用该bean的setter方法，即实现了基于setter的依赖注入。

构造器依赖注入和 Setter方法注入的区别

构造函数注入	setter 注入
没有部分注入	有部分注入
不会覆盖 setter 属性	会覆盖 setter 属性
任意修改都会创建一个新实例	任意修改不会创建一个新实例
适用于设置很多属性	适用于设置少量属性

两种依赖方式都可以使用，构造器注入和Setter方法注入。最好的解决方案是用构造器参数实现强制依赖，setter方法实现可选依赖。

Spring Beans（19）

什么是Spring beans?

Spring beans 是那些形成Spring应用的主干的java对象。它们被Spring IOC容器初始化，装配，和管理。这些beans通过容器中配置的元数据创建。比如，以XML文件中 的形式定义。

一个 Spring Bean 定义 包含什么?

一个Spring Bean 的定义包含容器必知的所有配置元数据，包括如何创建一个bean，它的生命周期详情及它的依赖。

如何给Spring 容器提供配置元数据？Spring有几种配置方式

这里有三种重要的方法给Spring 容器提供配置元数据。

- XML配置文件。
- 基于注解的配置。
- 基于java的配置。

Spring配置文件包含了哪些信息

Spring配置文件是个XML 文件，这个文件包含了类信息，描述了如何配置它们，以及如何相互调用。

Spring基于xml注入bean的几种方式

1. Set方法注入；
2. 构造器注入：①通过index设置参数的位置；②通过type设置参数类型；
3. 静态工厂注入；
4. 实例工厂；

你怎样定义类的作用域?

当定义一个 在Spring里，我们还能给这个bean声明一个作用域。它可以通过bean 定义中的scope属性来定义。如，当Spring要在需要的时候每次生产一个新的bean实例，bean的scope属性被指定为prototype。另一方面，一个bean每次使用的时候必须返回同一个实例，这个bean的scope 属性 必须设为 singleton。

解释Spring支持的几种bean的作用域

Spring框架支持以下五种bean的作用域：

- **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。
- **request**：每次http请求都会创建一个bean，该作用域仅在基于web的Spring ApplicationContext情形下有效。
- **session**：在一个HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。
- **global-session**：在一个全局的HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。

注意： 缺省的Spring bean 的作用域是Singleton。使用 prototype 作用域需要慎重的思考，因为频繁创建和销毁 bean 会带来很大的性能开销。

Spring框架中的单例bean是线程安全的吗?

不是，Spring框架中的单例bean不是线程安全的。

spring 中的 bean 默认是单例模式，spring 框架并没有对单例 bean 进行多线程的封装处理。

实际上大部分时候 spring bean 无状态的（比如 dao 类），所有某种程度上来说 bean 也是安全的，但如果 bean 有状态的话（比如 view model 对象），那就要开发者自己去保证线程安全了，最简单的就是改变 bean 的作用域，把“singleton”变更为“prototype”，这样请求 bean 相当于 new Bean()了，所以就可以保证线程安全了。

- 有状态就是有数据存储功能。
- 无状态就是不会保存数据。

Spring如何处理线程并发问题?

在一般情况下，只有无状态的Bean才可以在多线程环境下共享，在Spring中，绝大部分Bean都可以声明为singleton作用域，因为Spring对一些Bean中非线程安全状态采用ThreadLocal进行处理，解决线程安全问题。

ThreadLocal和线程同步机制都是为了解决多线程中相同变量的访问冲突问题。同步机制采用了“时间换空间”的方式，仅提供一份变量，不同的线程在访问前需要获取锁，没获得锁的线程则需要排队。而ThreadLocal采用了“空间换时间”的方式。

ThreadLocal会为每一个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进ThreadLocal。

问题：解释Spring框架中bean的生命周期

> 注意，这个是重点问题

Spring Bean的生命周期是Spring面试热点问题。这个问题即考察对Spring的微观了解，又考察对Spring的宏观认识，想要答好并不容易！本文希望能够从源码角度入手，帮助面试者彻底搞定Spring Bean的生命周期。

参考答案:

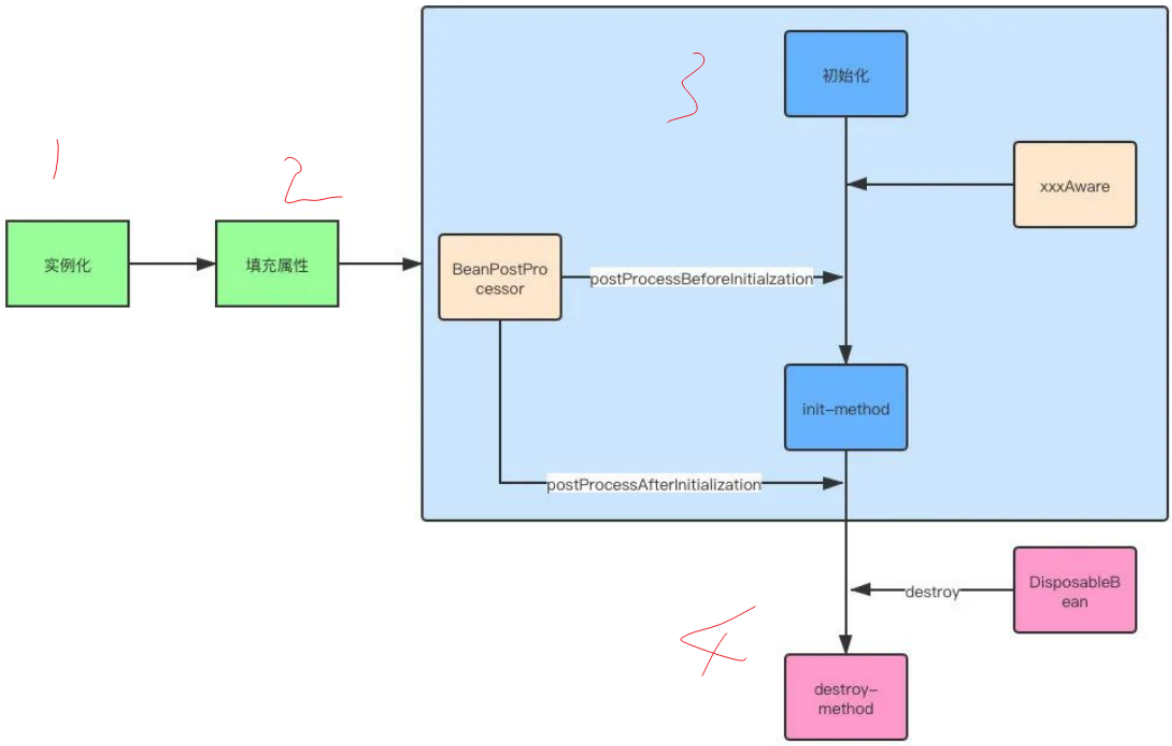
首先，回答阶段的数量：**只有四个!**

> 是的，Spring Bean的生命周期只有这四个阶段。把这四个阶段和每个阶段对应的扩展点糅合在一起虽然没有问题，但是这样非常凌乱，难以记忆。

要彻底搞清楚Spring的生命周期，首先要把这四个阶段牢牢记住。实例化和属性赋值对应构造方法和setter方法的注入，初始化和销毁是用户能自定义扩展的两个阶段。在这四步之间穿插的各种扩展点，稍后会讲。

1. 实例化 Instantiation
2. 属性赋值 Populate
3. 初始化 Initialization
4. 销毁 Destruction

实例化 -> 属性赋值 -> 初始化 -> 销毁



各个阶段的工作:

1. 实例化，创建一个Bean对象
2. 填充属性，为属性赋值
3. 初始化
4.
    - 如果实现了`xxxAware`接口，通过不同类型的Aware接口拿到Spring容器的资源
    - 如果实现了BeanPostProcessor接口，则会回调该接口的`postProcessBeforeInitialzation`和`postProcessAfterInitialization`方法
    - 如果配置了`init-method`方法，则会执行`init-method`配置的方法
5. 销毁
6.
    - 容器关闭后，如果Bean实现了`DisposableBean`接口，则会回调该接口的`destroy`方法
    - 如果配置了`destroy-method`方法，则会执行`destroy-method`配置的方法

源码学习：

前三个阶段，主要逻辑都在doCreate()方法中，逻辑很清晰，就是顺序调用以下三个方法，这三个方法与三个生命周期阶段一一对应，非常重要，在后续扩展接口分析中也会涉及。

1. createBeanInstance() -> 实例化
2. populateBean() -> 属性赋值
3. initializeBean() -> 初始化

> 注：bean的生命周期是从将bean定义全部注册到BeanFacotry中以后开始的。

源码如下，能证明实例化，属性赋值和初始化这三个生命周期的存在。关于本文的Spring源码都将忽略无关部分，便于理解：

前三个阶段的源码：

```
// 忽略了无关代码
protected Object doCreateBean(final String beanName, final
RootBeanDefinition mbd, final @Nullable Object[] args)
      throws BeanCreationException {
   // Instantiate the bean.
   BeanWrapper instanceWrapper = null;
   if (instanceWrapper == null) {
       // 实例化阶段!
      instanceWrapper = createBeanInstance(beanName, mbd, args);
   }
   // Initialize the bean instance.
   Object exposedObject = bean;
   try {
       // 属性赋值阶段!
      populateBean(beanName, mbd, instanceWrapper);
       // 初始化阶段!
      exposedObject = initializeBean(beanName, exposedObject, mbd);
   }
}
```

上面这些这个实例化Bean的方法是在getBean()方法中调用的，而getBean是在finishBeanFactoryInitialization方法中调用的，用来实例化单例非懒加载Bean，源码如下：

```
@Override
public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
```

```
        try {
            // Allows post-processing of the bean factory in context
subclasses.
            postProcessBeanFactory(beanFactory);
            // Invoke factory processors registered as beans in the
context.
            invokeBeanFactoryPostProcessors(beanFactory);
            // Register bean processors that intercept bean creation.

            // 所有BeanPostProcesser初始化的调用点
            registerBeanPostProcessors(beanFactory);
            // Initialize message source for this context.
            initMessageSource();
            // Initialize event multicaster for this context.
            initApplicationEventMulticaster();
            // Initialize other special beans in specific context
subclasses.
            onRefresh();
            // Check for listener beans and register them.
            registerListeners();
            // Instantiate all remaining (non-lazy-init) singletons.

            // 所有单例非懒加载Bean的调用点
            finishBeanFactoryInitialization(beanFactory);
            // Last step: publish corresponding event.
            finishRefresh();
        }
    }
}
```

销毁Bean阶段:

至于销毁，是在容器关闭时调用的，详见 ConfigurableApplicationContext#close()

高分答题的技巧:

> 如果回答了上面的答案可以拿到100分的话，加上下面的内容，就是120分

生命周期常用扩展点

Spring生命周期相关的常用扩展点非常多，所以问题不是不知道，而是记不住或者记不牢。其实记不住的根本原因还是不够了解，这里通过源码+分类的方式帮大家记忆。

区分影响一个bean或者多个bean是从源码分析得出的.

以BeanPostProcessor为例:

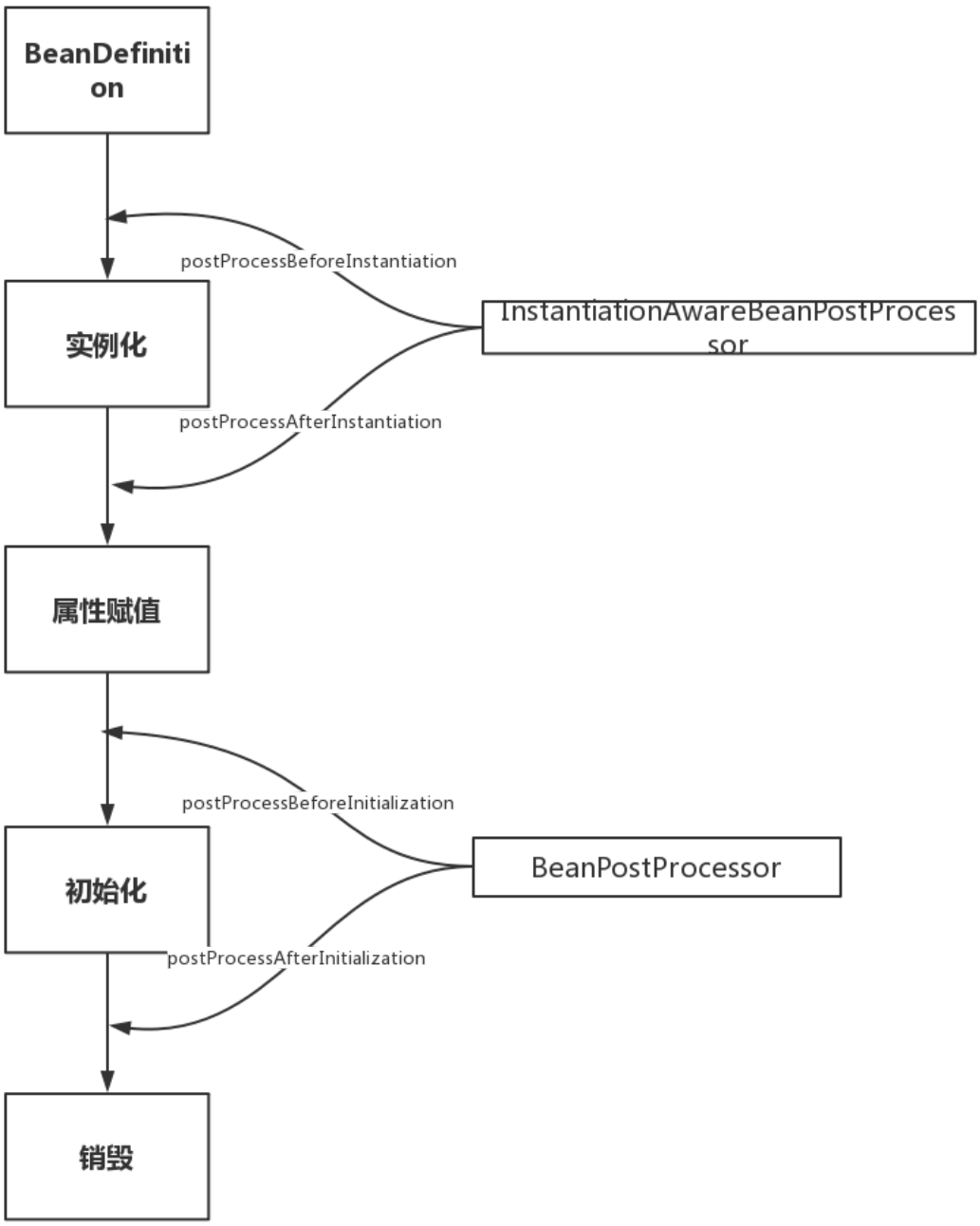
1. 从refresh方法来看,BeanPostProcessor 实例化比正常的bean早.
2. 从initializeBean方法看,每个bean初始化前后都调用所有BeanPostProcessor的postProcessBeforeInitialization和postProcessAfterInitialization方法.

第一大类：影响多个Bean的接口

实现了这些接口的Bean会切入到多个Bean的生命周期中。正因为如此，这些接口的功能非常强大，Spring内部扩展也经常使用这些接口，例如自动注入以及AOP的实现都和他们有关。

- InstantiationAwareBeanPostProcessor
- BeanPostProcessor

这两兄弟可能是Spring扩展中**最重要**的两个接口！InstantiationAwareBeanPostProcessor作用于**实例化**阶段的前后，BeanPostProcessor作用于**初始化**阶段的前后。正好和第一、第三个生命周期阶段对应。通过图能更好理解：



InstantiationAwareBeanPostProcessor

InstantiationAwareBeanPostProcessor实际上继承了BeanPostProcessor接口，严格意义上来看他们不是两兄弟，而是两父子。但是从生命周期角度我们重点关注其特有的对实例化阶段的影响，图中省略了从BeanPostProcessor继承的方法。

```
InstantiationAwareBeanPostProcessor extends BeanPostProcessor
```

InstantiationAwareBeanPostProcessor源码分析：

- **postProcessBeforeInstantiation调用点**，忽略无关代码：

```
@Override
protected Object createBean(String beanName, RootBeanDefinition mbd,
@Nullable Object[] args)
        throws BeanCreationException {
    try {
        // Give BeanPostProcessors a chance to return a proxy instead of
the target bean instance.
        // postProcessBeforeInstantiation方法调用点，这里就不跟进了，
        // 有兴趣的同学可以自己看下，就是for循环调用所有的
InstantiationAwareBeanPostProcessor
        Object bean = resolveBeforeInstantiation(beanName, mbdToUse);
        if (bean != null) {
            return bean;
        }
    }

    try {
        // 上文提到的doCreateBean方法，可以看到
        // postProcessBeforeInstantiation方法在创建Bean之前调用
        Object beanInstance = doCreateBean(beanName, mbdToUse, args);
        if (logger.isTraceEnabled()) {
            logger.trace("Finished creating instance of bean '" + beanName
+ "'");
        }
        return beanInstance;
    }

}
```

可以看到，postProcessBeforeInstantiation在doCreateBean之前调用，也就是在bean实例化之前调用的，英文源码注释解释道该方法的返回值会替换原本的Bean作为代理，这也是Aop等功能实现的关键点。

- **postProcessAfterInstantiation调用点**，忽略无关代码：

```
protected void populateBean(String beanName, RootBeanDefinition mbd,
@Nullable BeanWrapper bw) {
    // Give any InstantiationAwareBeanPostProcessors the opportunity to
modify the
    // state of the bean before properties are set. This can be used, for
example,
    // to support styles of field injection.
    boolean continueWithPropertyPopulation = true;

     // InstantiationAwareBeanPostProcessor#postProcessAfterInstantiation()
     // 方法作为属性赋值的前置检查条件，在属性赋值之前执行，能够影响是否进行属性赋值！
    if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
       for (BeanPostProcessor bp : getBeanPostProcessors()) {
          if (bp instanceof InstantiationAwareBeanPostProcessor) {
```

```
            InstantiationAwareBeanPostProcessor ibp =
(InstantiationAwareBeanPostProcessor) bp;
            if
(!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {
                continueWithPropertyPopulation = false;
                break;
            }
          }
        }
    }

    // 忽略后续的属性赋值操作代码
}
```

可以看到该方法在属性赋值方法内，但是在真正执行赋值操作之前。其返回值为boolean，返回false时可以阻断属性赋值阶段（continueWithPropertyPopulation = false;）。

BeanPostProcessor

关于BeanPostProcessor执行阶段的源码穿插在下文Aware接口的调用时机分析中，因为部分Aware功能的就是通过他实现的!只需要先记住BeanPostProcessor在初始化前后调用就可以了。

接口源码：

```
public interface BeanPostProcessor {
     //bean初始化之前调用
    @Nullable
    default Object postProcessBeforeInitialization(Object bean, String
beanName) throws BeansException {
        return bean;
    }

    //bean初始化之后调用
    @Nullable
    default Object postProcessAfterInitialization(Object bean, String
beanName) throws BeansException {
        return bean;
    }
}

```

第二大类：只调用一次的接口

这一大类接口的特点是功能丰富，常用于用户自定义扩展。

第二大类中又可以分为两类：

1. Aware类型的接口
2. 生命周期接口

无所不知的Aware

Aware类型的接口的作用就是让我们能够拿到Spring容器中的一些资源。基本都能够见名知意，Aware之前的名字就是可以拿到什么资源，例如BeanNameAware可以拿到BeanName，以此类推。调用时机需要注意：**所有的Aware方法都是在初始化阶段之前调用的！**

Aware接口众多，这里同样通过分类的方式帮助大家记忆。

Aware接口具体可以分为两组，至于为什么这么分，详见下面的源码分析。如下排列顺序同样也是Aware接口的执行顺序，能够见名知意的接口不再解释。

Aware Group1

1. BeanNameAware
2. BeanClassLoaderAware
3. BeanFactoryAware

Aware Group2

1. EnvironmentAware
2. EmbeddedValueResolverAware 这个知道的人可能不多，实现该接口能够获取Spring EL解析器，用户的自定义注解需要支持spel表达式的时候可以使用，非常方便。
3. ApplicationContextAware(ResourceLoaderAware\ApplicationEventPublisherAware\MessageSourceAware) 这几个接口可能让人有点懵，实际上这几个接口可以一起记，其返回值实质上都是当前的ApplicationContext对象，因为ApplicationContext是一个复合接口，如下：

```
public interface ApplicationContext extends EnvironmentCapable,
ListableBeanFactory, HierarchicalBeanFactory,
        MessageSource, ApplicationEventPublisher, ResourcePatternResolver {}
```

这里涉及到另一道面试题，ApplicationContext和BeanFactory的区别，可以从ApplicationContext继承的这几个接口入手，除去BeanFactory相关的两个接口就是ApplicationContext独有的功能，这里不详细说明。

Aware调用时机源码分析

详情如下，忽略了部分无关代码。代码位置就是我们上文提到的initializeBean方法详情，这也说明了Aware都是在初始化阶段之前调用的！

```
// 见名知意，初始化阶段调用的方法
protected Object initializeBean(final String beanName, final Object bean,
@Nullable RootBeanDefinition mbd) {
    // 这里调用的是Group1中的三个Bean开头的Aware
    invokeAwareMethods(beanName, bean);

    Object wrappedBean = bean;

    // 这里调用的是Group2中的几个Aware，
    // 而实质上这里就是前面所说的BeanPostProcessor的调用点！
    // 也就是说与Group1中的Aware不同，这里是通过
BeanPostProcessor（ApplicationContextAwareProcessor）实现的。
    wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean,
beanName);

    // 这个是初始化方法，下文要介绍的InitializingBean调用点就是在这个方法里面
    invokeInitMethods(beanName, wrappedBean, mbd);
```

```
15
16      // BeanPostProcessor的另一个调用点
17      wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean,
    beanName);
18
19      return wrappedBean;
20  }
```

可以看到并不是所有的Aware接口都使用同样的方式调用。Bean××Aware都是在代码中直接调用的，而ApplicationContext相关的Aware都是通过BeanPostProcessor#postProcessBeforeInitialization()实现的。感兴趣的可以自己看一下ApplicationContextAwareProcessor这个类的源码，就是判断当前创建的Bean是否实现了相关的Aware方法，如果实现了会调用回调方法将资源传递给Bean。

至于Spring为什么这么实现，应该没什么特殊的考量。也许和Spring的版本升级有关。基于对修改关闭，对扩展开放的原则，Spring对一些新的Aware采用了扩展的方式添加。

BeanPostProcessor的调用时机也能在这里体现，包围住invokeInitMethods方法，也就说明了在初始化阶段的前后执行。

关于Aware接口的执行顺序，其实只需要记住第一组在第二组执行之前就行了。每组中各个Aware方法的调用顺序其实没有必要记，有需要的时候点进源码一看便知。

简单的两个生命周期接口

至于剩下的两个生命周期接口就很简单了，实例化和属性赋值都是Spring帮助我们做的，能够自己实现的有初始化和销毁两个生命周期阶段。

InitializingBean接口

InitializingBean顾名思义，是初始化Bean相关的接口。

接口定义：

```
1  public interface InitializingBean {
2
3      void afterPropertiesSet() throws Exception;
4
5  }
```

看方法名，是在读完Properties文件，之后执行的方法。afterPropertiesSet()方法是在初始化过程中被调用的。

InitializingBean 对应生命周期的初始化阶段，在上面源码的invokeInitMethods(beanName, wrappedBean, mbd);方法中调用。

有一点需要注意，因为Aware方法都是执行在初始化方法之前，所以可以在初始化方法中放心大胆的使用Aware接口获取的资源，这也是我们自定义扩展Spring的常用方式。

除了实现InitializingBean接口之外还能通过注解（@PostConstruct）或者xml配置的方式指定初始化方法（init-method），至于这几种定义方式的调用顺序其实没有必要记。因为这几个方法对应的都是同一个生命周期，只是实现方式不同，我们一般只采用其中一种方式。

三种实现指定初始化方法的方法：

- 使用@PostConstruct注解，该注解作用于void方法上
- 在配置文件中配置init-method方法

```
<bean id="student" class="com.demo.Student" init-method="init2">
        <property name="name" value="小明"></property>
        <property name="age" value="20"></property>
        <property name="school" ref="school"></property>
</bean>
```

- 将类实现InitializingBean接口

```
@Component("student")
public class Student implements InitializingBean{
    private String name;
    private int age;
            …
}
```

执行：

```
@Component("student")
public class Student implements InitializingBean{
    private String name;
    private int age;


    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }

    //1.使用postconstrtct注解
    @PostConstruct
    public void init(){
        System.out.println("执行 init方法");
    }

    //2.在xml配置文件中配置init-method方法
    public void init2(){
        System.out.println("执行init2方法 ");
    }

    //3.实现InitializingBean接口
    public void afterPropertiesSet() throws Exception {
        System.out.println("执行init3方法");
    }
}
```

通过测试我们可以得出结论，三种实现方式的执行顺序是：

Constructor > @PostConstruct > InitializingBean > init-method

DisposableBean接口

DisposableBean 类似于InitializingBean，对应生命周期的销毁阶段，**以ConfigurableApplicationContext#close()方法作为入口**，实现是通过循环获取所有实现了DisposableBean接口的Bean然后调用其destroy()方法 。

接口定义：

```
public interface DisposableBean {
    void destroy() throws Exception;
}
```

定义一个实现了DisposableBean接口的Bean：

```
public class IndexBean implements InitializingBean,DisposableBean {
    public void destroy() throws Exception {
        System.out.println("destroy");
    }
    public void afterPropertiesSet() throws Exception {
        System.out.println("init-afterPropertiesSet()");
    }
    public void test(){
        System.out.println("init-test()");
    }
}
```

执行：

```
public class Main {
    public static void main(String[] args) {
        AbstractApplicationContext applicationContext=new
ClassPathXmlApplicationContext("classpath:application-usertag.xml");
        System.out.println("init-success");
        applicationContext.registerShutdownHook();
    }
}
```

执行结果：

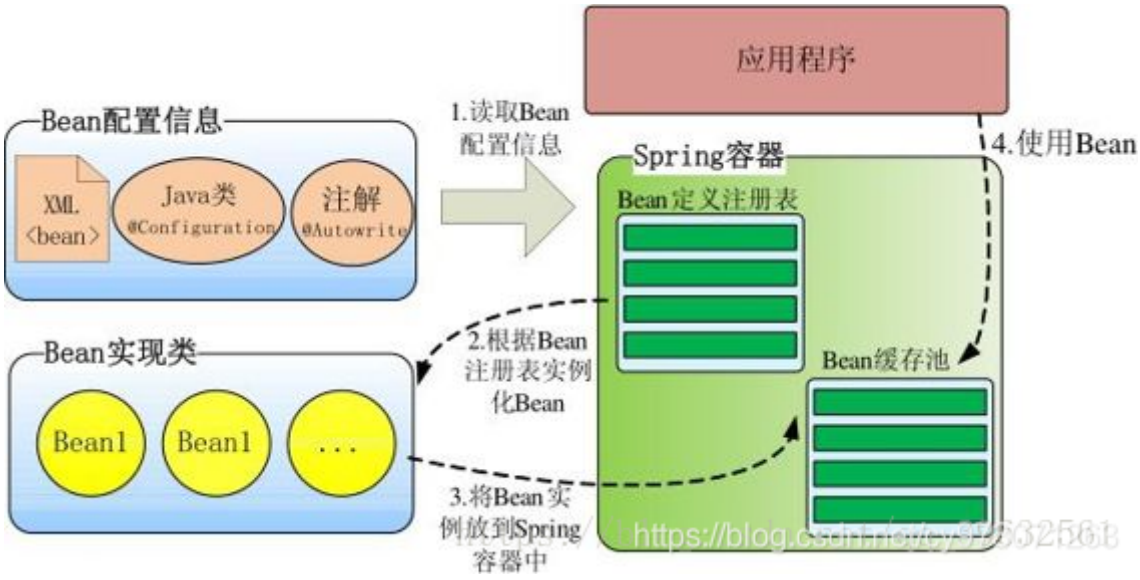
```
init-afterPropertiesSet()
init-test()
init-success
destroy
```

也就是说，在对象销毁的时候，会去调用DisposableBean的destroy方法。在进入到销毁过程时先去调用一下DisposableBean的destroy方法，然后后执行 destroy-method声明的方法（用来销毁Bean中的各项数据）。

扩展阅读: BeanPostProcessor注册时机与执行顺序

首先要明确一个概念，在spring中一切皆bean

所有的组件都会被作为一个bean装配到spring容器中，过程如下图：



所以我们前面所讲的那些拓展点，也都会被作为一个个bean装配到spring容器中

注册时机

我们知道BeanPostProcessor也会注册为Bean，那么Spring是如何保证BeanPostProcessor在我们的业务Bean之前初始化完成呢?

请看我们熟悉的refresh()方法的源码，省略部分无关代码（refresh的详细注解见refresh()）：

```
@Override
public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        try {
            // Allows post-processing of the bean factory in context subclasses.
            postProcessBeanFactory(beanFactory);

            // Invoke factory processors registered as beans in the context.
            invokeBeanFactoryPostProcessors(beanFactory);

            // Register bean processors that intercept bean creation.
            // 注册所有BeanPostProcesser的方法
            registerBeanPostProcessors(beanFactory);

            // Initialize message source for this context.
            initMessageSource();

```

```
            // Initialize event multicaster for this context.
            initApplicationEventMulticaster();

            // Initialize other special beans in specific context
subclasses.
            onRefresh();

            // Check for listener beans and register them.
            registerListeners();

            // Instantiate all remaining (non-lazy-init) singletons.
            // 所有单例非懒加载Bean的创建方法
            finishBeanFactoryInitialization(beanFactory);

            // Last step: publish corresponding event.
            finishRefresh();
        }
}
```

可以看出，Spring是先执行registerBeanPostProcessors()进行BeanPostProcessors的注册，然后再执行finishBeanFactoryInitialization创建我们的单例非懒加载的Bean。

执行顺序

BeanPostProcessor有很多个，而且每个BeanPostProcessor都影响多个Bean，其执行顺序至关重要，必须能够控制其执行顺序才行。关于执行顺序这里需要引入两个排序相关的接口：PriorityOrdered、Ordered

- PriorityOrdered是一等公民，首先被执行，PriorityOrdered公民之间通过接口返回值排序
- Ordered是二等公民，然后执行，Ordered公民之间通过接口返回值排序
- 都没有实现是三等公民，最后执行

在以下源码中，可以很清晰的看到Spring注册各种类型BeanPostProcessor的逻辑，根据实现不同排序接口进行分组。优先级高的先加入，优先级低的后加入。

```
// First, invoke the BeanDefinitionRegistryPostProcessors that implement
PriorityOrdered.
// 首先，加入实现了PriorityOrdered接口的BeanPostProcessors，顺便根据
PriorityOrdered排了序
String[] postProcessorNames =
beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class,
true, false);
for (String ppName : postProcessorNames) {
    if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
        currentRegistryProcessors.add(beanFactory.getBean(ppName,
BeanDefinitionRegistryPostProcessor.class));
        processedBeans.add(ppName);
    }
}

sortPostProcessors(currentRegistryProcessors, beanFactory);
registryProcessors.addAll(currentRegistryProcessors);
invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors,
registry);
currentRegistryProcessors.clear();
```

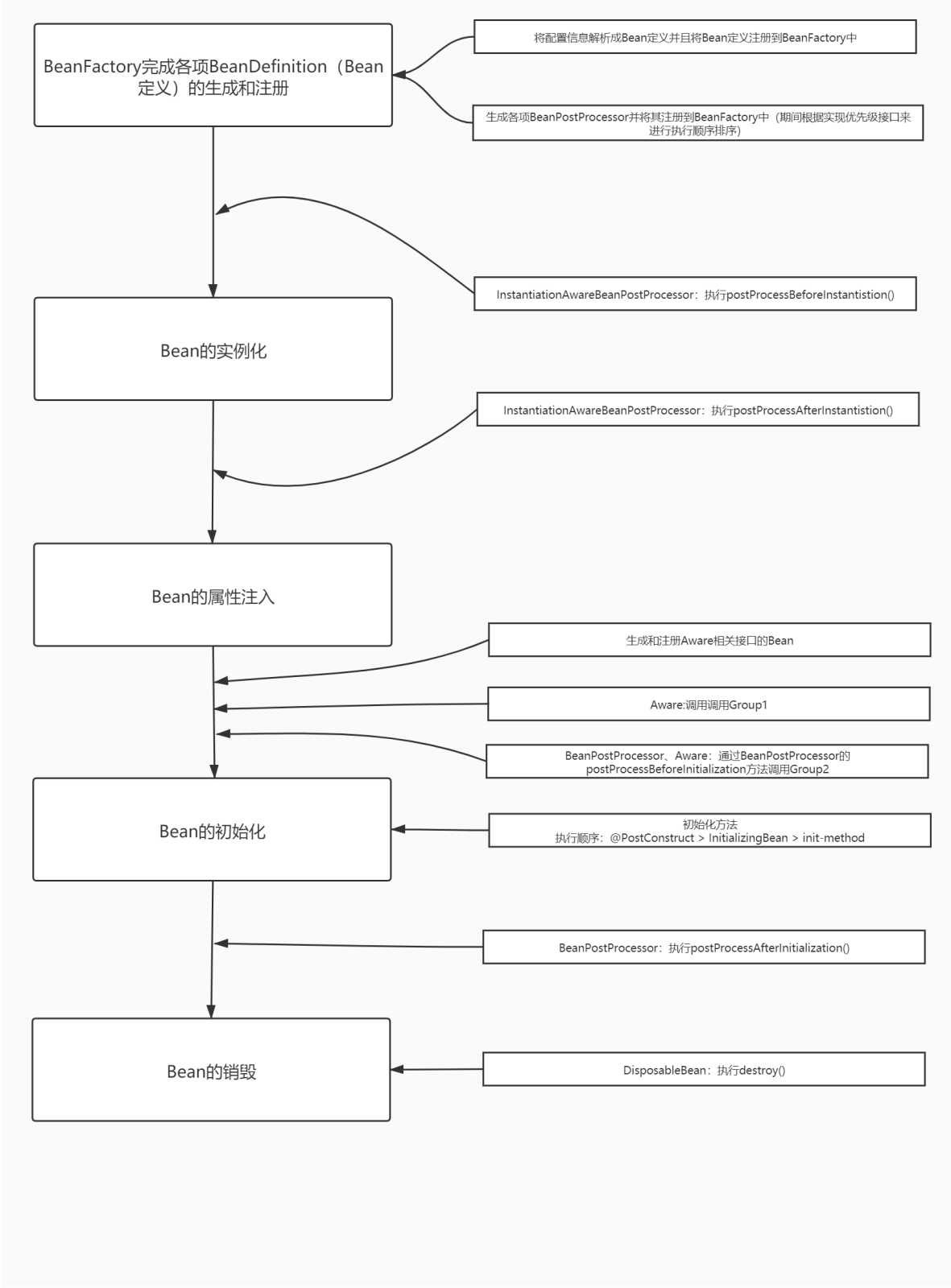
```
16
17  // Next, invoke the BeanDefinitionRegistryPostProcessors that implement
    Ordered.
18  // 然后，加入实现了Ordered接口的BeanPostProcessors，顺便根据Ordered排了序
19  postProcessorNames =
    beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class,
    true, false);
20  for (String ppName : postProcessorNames) {
21      if (!processedBeans.contains(ppName) && beanFactory.isTypeMatch(ppName,
    Ordered.class)) {
22          currentRegistryProcessors.add(beanFactory.getBean(ppName,
    BeanDefinitionRegistryPostProcessor.class));
23          processedBeans.add(ppName);
24      }
25  }
26  sortPostProcessors(currentRegistryProcessors, beanFactory);
27  registryProcessors.addAll(currentRegistryProcessors);
28  invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors,
    registry);
29  currentRegistryProcessors.clear();
30  // Finally, invoke all other BeanDefinitionRegistryPostProcessors until no
    further ones appear.
31
32  // 最后加入其他常规的BeanPostProcessors
33  boolean reiterate = true;
34  while (reiterate) {
35      reiterate = false;
36      postProcessorNames =
    beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class,
    true, false);
37      for (String ppName : postProcessorNames) {
38          if (!processedBeans.contains(ppName)) {
39              currentRegistryProcessors.add(beanFactory.getBean(ppName,
    BeanDefinitionRegistryPostProcessor.class));
40              processedBeans.add(ppName);
41              reiterate = true;
42          }
43      }
44      sortPostProcessors(currentRegistryProcessors, beanFactory);
45      registryProcessors.addAll(currentRegistryProcessors);
46      invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors,
    registry);
47      currentRegistryProcessors.clear();
48  }
```

根据排序接口返回值排序，默认升序排序，返回值越低优先级越高。

```
1  /**
2   * Useful constant for the highest precedence value.
3   * @see java.lang.Integer#MIN_VALUE
4   */
5  int HIGHEST_PRECEDENCE = Integer.MIN_VALUE;
6  /**
7   * Useful constant for the lowest precedence value.
8   * @see java.lang.Integer#MAX_VALUE
9   */
10 int LOWEST_PRECEDENCE = Integer.MAX_VALUE;
```

PriorityOrdered、Ordered接口作为Spring整个框架通用的排序接口，在Spring中应用广泛，也是非常重要的接口。

Bean的生命周期流程图



总结

Spring Bean的生命周期分为`四个阶段`和`多个扩展点`。扩展点又可以分为`影响多个Bean`和`影响单个Bean`。整理如下：

四个阶段

- 实例化 Instantiation
- 属性赋值 Populate
- 初始化 Initialization
- 销毁 Destruction

多个扩展点

- 影响多个Bean
    - BeanPostProcessor
    - InstantiationAwareBeanPostProcessor
- 影响单个Bean
    - Aware
        - Aware Group1
            - BeanNameAware
            - BeanClassLoaderAware
            - BeanFactoryAware
        - Aware Group2
            - EnvironmentAware
            - EmbeddedValueResolverAware
            - ApplicationContextAware(ResourceLoaderAware\ApplicationEventPublisherAware\MessageSourceAware)
    - 生命周期
        - InitializingBean
        - DisposableBean

哪些是重要的bean生命周期方法？ 你能重载它们吗?

有两个重要的bean 生命周期方法，第一个是setup ， 它是在容器加载bean的时候被调用。第二个方法是 teardown 它是在容器卸载类的时候被调用。

bean 标签有两个重要的属性（init-method和destroy-method）。用它们你可以自己定制初始化和注销方法。它们也有相应的注解（@PostConstruct和@PreDestroy）。

什么是Spring的内部bean？什么是Spring inner beans?

在Spring框架中，当一个bean仅被用作另一个bean的属性时，它能被声明为一个内部bean。内部bean可以用setter注入“属性”和构造方法注入“构造参数”的方式来实现，内部bean通常是匿名的，它们的Scope一般是prototype。

在 Spring中如何注入一个java集合?

Spring提供以下几种集合的配置元素：

类型用于注入一列值，允许有相同的值。

类型用于注入一组值，不允许有相同的值。

类型用于注入一组键值对，键和值都可以为任意类型。

类型用于注入一组键值对，键和值都只能为String类型。

什么是bean装配?

装配，或bean 装配是指在Spring 容器中把bean组装到一起，前提是容器需要知道bean的依赖关系，如何通过依赖注入来把它们装配到一起。

什么是bean的自动装配?

在Spring框架中，在配置文件中设定bean的依赖关系是一个很好的机制，Spring 容器能够自动装配相互合作的bean，这意味着容器不需要和配置，能通过Bean工厂自动处理bean之间的协作。这意味着Spring可以通过向Bean Factory中注入的方式自动搞定bean之间的依赖关系。自动装配可以设置在每个bean上，也可以设定在特定的bean上。

解释不同方式的自动装配，spring 自动装配 bean 有哪些方式?

在spring中，对象无需自己查找或创建与其关联的其他对象，由容器负责把需要相互协作的对象引用赋予各个对象，使用autowire来配置自动装载模式。

在Spring框架xml配置中共有5种自动装配：

- no：默认的方式是不进行自动装配的，通过手工设置ref属性来进行装配bean。
- byName：通过bean的名称进行自动装配，如果一个bean的 property 与另一bean 的name 相同，就进行自动装配。
- byType：通过参数的数据类型进行自动装配。
- constructor：利用构造函数进行装配，并且构造函数的参数通过byType进行装配。
- autodetect：自动探测，如果有构造方法，通过 construct的方式自动装配，否则使用 byType的方式自动装配。

使用@Autowired注解自动装配的过程是怎样的?

使用@Autowired注解来自动装配指定的bean。在使用@Autowired注解之前需要在Spring配置文件进行配置，<context:annotation-config />。

在启动spring IoC时，容器自动装载了一个AutowiredAnnotationBeanPostProcessor后置处理器，当容器扫描到@Autowied、@Resource或@Inject时，就会在IoC容器自动查找需要的bean，并装配给该对象的属性。在使用@Autowired时，首先在容器中查询对应类型的bean：

- 如果查询结果刚好为一个，就将该bean装配给@Autowired指定的数据；
- 如果查询的结果不止一个，那么@Autowired会根据名称来查找；
- 如果上述查找的结果为空，那么会抛出异常。解决方法时，使用required=false。

自动装配有哪些局限性?

自动装配的局限性是：

重写：你仍需用 和 配置来定义依赖，意味着总要重写自动装配。

基本数据类型：你不能自动装配简单的属性，如基本数据类型，String字符串，和类。

模糊特性：自动装配不如显式装配精确，如果有可能，建议使用显式装配。

你可以在Spring中注入一个null 和一个空字符串吗?

可以。

问题： FactoryBean 和 BeanFactory有什么区别?

简要的答案:

> BeanFactory 是 Bean 的工厂， ApplicationContext 的父类，IOC 容器的核心，负责生产和管理 Bean 对象。
>
> FactoryBean 是 Bean，可以通过实现 FactoryBean 接口定制实例化 Bean 的逻辑，通过代理一个Bean对象，对方法前后做一些操作。

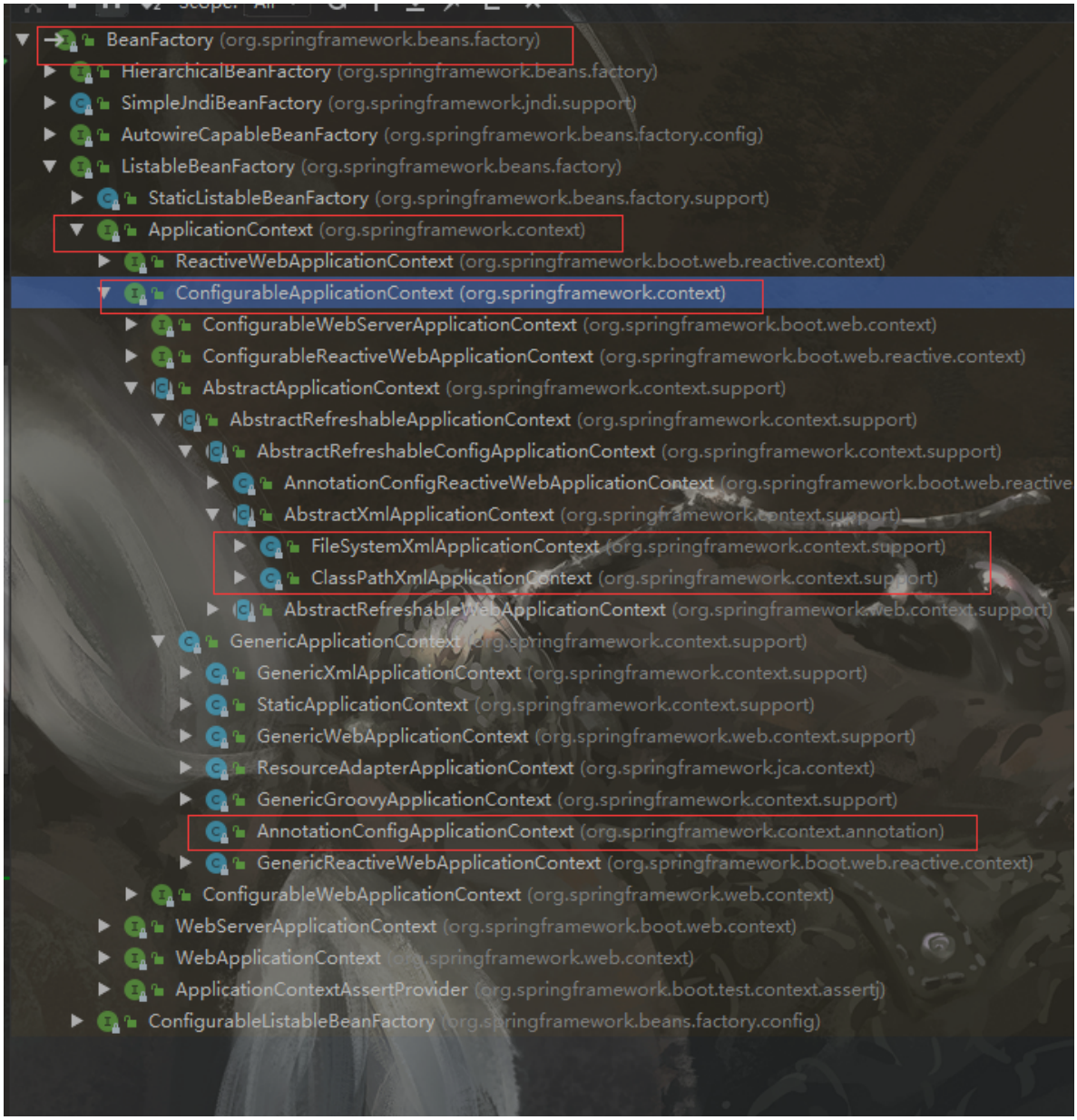
具体的介绍:

（1） BeanFactory 是ioc容器的底层实现接口，是ApplicationContext 顶级接口

spring不允许我们直接操作 BeanFactory bean工厂，所以为我们提供了ApplicationContext 这个接口 此接口集成BeanFactory 接口，ApplicationContext包含BeanFactory的所有功能,同时还进行更多的扩展。

BeanFactory 接口又衍生出以下接口，其中我们经常用到的是ApplicationContext 接口

ApplicationContext 继承图



ConfiguableApplicationContext 中添加了一些方法：

```
... 其他省略

    //刷新ioc容器上下文
    void refresh() throws BeansException, IllegalStateException;

// 关闭此应用程序上下文，释放所有资源并锁定，销毁所有缓存的单例bean。
    @Override
    void close();

//确定此应用程序上下文是否处于活动状态，即，是否至少刷新一次且尚未关闭。
    boolean isActive();

    ... 其他省略
```

主要作用在ioc容器进行相应的刷新，关闭等操作！

```
FileSystemXmlApplicationContext 和ClassPathXmlApplicationContext 是用来读取xml文件创建bean对象
ClassPathXmlApplicationContext ： 读取类路径下xml 创建bean
FileSystemXmlApplicationContext ：读取文件系统下xml创建bean
AnnotationConfigApplicationContext 主要是注解开发获取ioc中的bean实例
```

(2) FactoryBean 是spirng提供的工厂bean的一个接口

FactoryBean 接口提供三个方法，用来创建对象，
FactoryBean 具体返回的对象是由getObject 方法决定的。

```
*/
public interface FactoryBean<T> {

//创建的具体bean对象的类型
    @Nullable
    T getObject() throws Exception;

 //工厂bean 具体创建具体对象是由此getObject()方法来返回的
    @Nullable
    Class<?> getObjectType();

  //是否单例
    default boolean isSingleton() {
        return true;
    }

}
```

创建一个FactoryBean 用来生产User对象

```
@Component
public class FactoryBeanTest implements FactoryBean<User> {


    //创建的具体bean对象的类型
    @Override
    public Class<?> getObjectType() {
        return User.class;
    }


    //是否单例
    @Override
    public boolean isSingleton() {
        return true;
    }

    //工厂bean 具体创建具体对象是由此getObject()方法来返回的
    @Override
    public User getObject() throws Exception {
```

```
        return new User();
    }
}
```

Junit测试

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = {FactoryBeanTest.class})
@WebAppConfiguration
public class SpringBootDemoApplicationTests {
    @Autowired
    private ApplicationContext applicationContext;

    @Test
    public void tesst() {
        FactoryBeanTest bean1 =
applicationContext.getBean(FactoryBeanTest.class);
        try {
            User object = bean1.getObject();
            System.out.println(object==object);
            System.out.println(object);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

结果

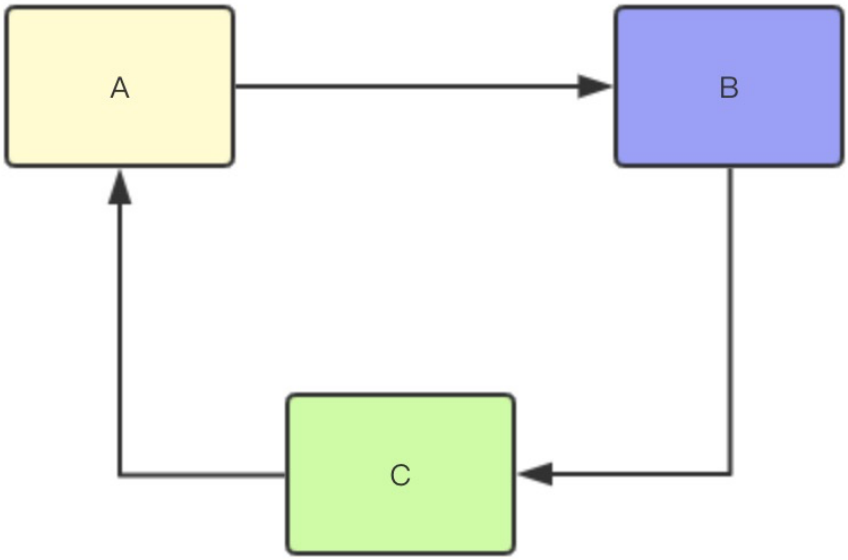
```
true
User [id=null, name=null, age=0]
```

简单的总结:

```
BeanFactory是个bean 工厂，是一个工厂类(接口)， 它负责生产和管理bean的一个工厂
是ioc 容器最底层的接口，是个ioc容器，是spring用来管理和装配普通bean的ioc容器（这些bean
成为普通bean）。

FactoryBean是个bean，在IOC容器的基础上给Bean的实现加上了一个简单工厂模式和装饰模式，是一
个可以生产对象和装饰对象的工厂bean，由spring管理后，生产的对象是由getObject()方法决定的
（从容器中获取到的对象不是
“ FactoryBeanTest ”对象）。
```

尼恩独家解读：循环依赖与三级缓存

循环依赖，其实就是循环引用，就是两个或者两个以上的 bean 互相引用对方，最终形成一个闭环，如 A 依赖 B，B 依赖 C，C 依赖 A。如下图所示:



Spring中的循环依赖，其实就是一个死循环的过程，

在初始化 A 的时候发现依赖了 B，这时就会去初始化 B，然后又发现 B 依赖 C，跑去初始化 C，初始化 C 的时候发现依赖了 A，则又会去初始化 A，依次循环永不退出，除非有终结条件。

一般来说，Spring 循环依赖的情况有两种：

- 构造器的循环依赖。
- field 属性的循环依赖。

对于构造器的循环依赖，Spring 是无法解决的，只能抛出 `BeanCurrentlyInCreationException` 异常表示循环依赖，

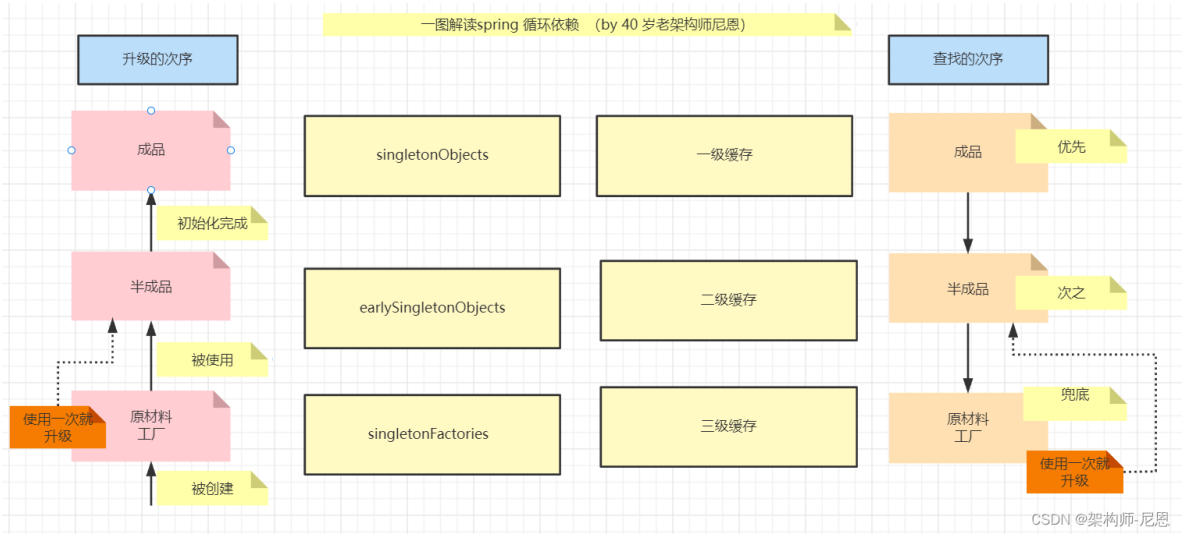
基于 field 属性的循环依赖，也要分两种细分的情况来对待：

- scope 为 singleton 的循环依赖。
- scope 为 prototype 的循环依赖。Spring 无法解决，直接抛出 BeanCurrentlyInCreationException 异常。

三级缓存解决方案

spring内部有三级缓存：

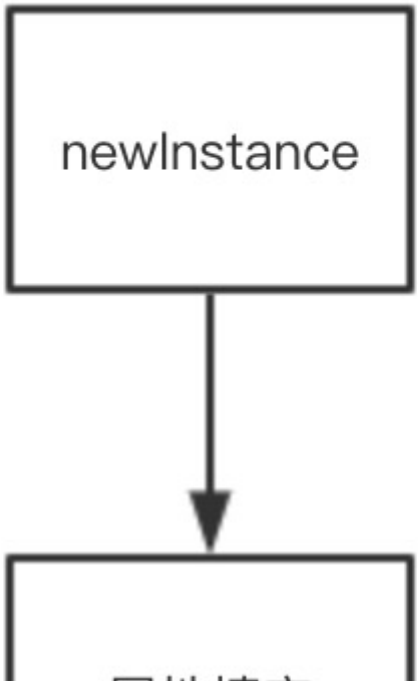
- **成品**：一级缓存 singletonObjects ， 用于保存实例化、注入、初始化完成的bean实例 ：
- **半成品**：二级缓存 earlySingletonObjects ， 用于保存实例化完成的bean实例
- **原材料工厂**： 三级缓存 singletonFactories ， 用于保存bean的创建工厂，以便于后面扩展有机会创建代理对象。

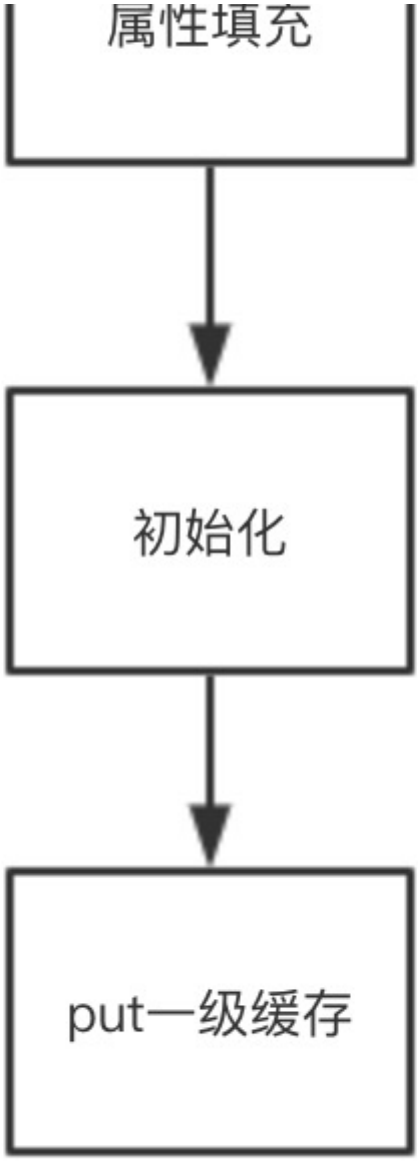


- 三级缓存是singletonFactories ， 但是是不完整的Bean的工厂Factory,是当一个Bean在new之后（没有属性填充、初始化），就put进去。所以，是原材料工厂
- 二级缓存是对三级缓存的过渡性处理，只要通过 `getObject()` 方法从三级缓存的BeanFactory中取出Bean一次，原材料就变成变成品，就put到二级缓存 ， 所以，二级缓存里边的bean，都是半成品
- 一级缓存里面是完整的Bean,是当一个Bean完全创建后(完成属性填充、彻底的完成初始化）才put进去， 所以，是成品

singleton类型bean的创建过程

Spring处理不同 scope 时，如果是 singleton调用的，创建过程如下图所示：





也就是说，一级缓存里面是完整的Bean，是产品

从源码解读，doGetBean的查找bean的次序

在AbstractBeanFactory 的 `doGetBean()` 方法中，我们根据BeanName去获取Singleton Bean的时候，会先从缓存获取。
代码如下：

```
//DefaultSingletonBeanRegistry.java

@Nullable
protected Object getSingleton(String beanName, boolean allowEarlyReference)
{
    // 从一级缓存缓存 singletonObjects 中加载 bean
    Object singletonObject = this.singletonObjects.get(beanName);
    // 缓存中的 bean 为空，且当前 bean 正在创建
    if (singletonObject == null &&
isSingletonCurrentlyInCreation(beanName)) {
        // 加锁
```

```
        synchronized (this.singletonObjects) {
            // 从 二级缓存 earlySingletonObjects 中获取
            singletonObject = this.earlySingletonObjects.get(beanName);
            // earlySingletonObjects 中没有，且允许提前创建
            if (singletonObject == null && allowEarlyReference) {
                // 从 三级缓存 singletonFactories 中获取对应的 ObjectFactory
                ObjectFactory<?> singletonFactory =
this.singletonFactories.get(beanName);
                if (singletonFactory != null) {
                    //从单例工厂中获取bean
                    singletonObject = singletonFactory.getObject();
                    // 添加到二级缓存
                    this.earlySingletonObjects.put(beanName,
singletonObject);
                    // 从三级缓存中删除
                    this.singletonFactories.remove(beanName);
                }
            }
        }
    }
    return singletonObject;
}
```

getSingleton()的查找bean的次序比较清晰：

首先，尝试从一级缓存singletonObjects （成品仓库）中获取单例Bean。

如果获取不到，则从二级缓存earlySingletonObjects （半成品仓库）中获取单例Bean。

如果仍然获取不到，则从三级缓存singletonFactories（原材料工厂 仓库）中获取单例BeanFactory原材料工厂。

最后，如果从三级缓存中拿到了BeanFactory，则通过getObject()生产一个最原始的bean，可以理解为最原始的bean实例（原材料），没有进行属性填充，也没有完成初始化

但是，由于此处是提取bean，所以bean原材料已经被使用了一次，所以，顺手把Bean存入二级缓存中，并把该Bean的（原材料工厂）从三级缓存中删除

三级缓存（原材料工厂 仓库）里边的（原材料工厂 ）在哪里设置呢

在 **AbstractAutowireCapableBeanFactory** 的 `doCreateBean()` 方法中，有这么一段代码：

```java
// AbstractAutowireCapableBeanFactory.java

boolean earlySingletonExposure = (mbd.isSingleton() // 单例模式
        && this.allowCircularReferences // 允许循环依赖
        && isSingletonCurrentlyInCreation(beanName)); // 当前单例 bean 是否正在被创建
if (earlySingletonExposure) {
    if (logger.isTraceEnabled()) {
        logger.trace("Eagerly caching bean '" + beanName +
                "' to allow for resolving potential circular references");
    }
    // 为了后期避免循环依赖，提前将创建的 bean 实例，加入到三级缓存 singletonFactories 中
    addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));
}
```

这段代码就是put三级缓存`singletonFactories`的地方，其核心逻辑是，当满足以下3个条件时，把bean加入三级缓存中：

- 单例
- 允许循环依赖
- 当前单例Bean正在创建

`addSingletonFactory(String beanName, ObjectFactory<?> singletonFactory)`方法，将原始的bean匿名工厂 （原材料工厂）加入到三级缓存，代码如下：

```java
// DefaultSingletonBeanRegistry.java

protected void addSingletonFactory(String beanName, ObjectFactory<?> singletonFactory) {
    Assert.notNull(singletonFactory, "Singleton factory must not be null");
    synchronized (this.singletonObjects) {
        if (!this.singletonObjects.containsKey(beanName)) {
            this.singletonFactories.put(beanName, singletonFactory);
            this.earlySingletonObjects.remove(beanName);
            this.registeredSingletons.add(beanName);
        }
    }
}
```

从这段代码我们可以看出，singletonFactories 这个三级缓存才是解决 Spring Bean 循环依赖的关键。

注意：这段代码发生在`createBeanInstance(...)`方法之后，也就是说这个 bean 其实已经被创建出来了，但是它还没有完善（没有进行属性填充和初始化），

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

```
6
7       if (instanceWrapper == null) {
8           instanceWrapper = this.createBeanInstance(beanName, mbd, args); //这
    个 bean 其实已经被创建出来了
9       }
10
11      Object bean = instanceWrapper.getWrappedInstance();
12      Class<?> beanType = instanceWrapper.getWrappedClass();
13      if (beanType != NullBean.class) {
14          mbd.resolvedTargetType = beanType;
15      }
16
17      synchronized(mbd.postProcessingLock) {
18          if (!mbd.postProcessed) {
19              try {
20                  this.applyMergedBeanDefinitionPostProcessors(mbd, beanType,
    beanName);
21              } catch (Throwable var17) {
22                  throw new
    BeanCreationException(mbd.getResourceDescription(), beanName, "Post-
    processing of merged bean definition failed", var17);
23              }
24
25              mbd.postProcessed = true;
26          }
27      }
28
29      boolean earlySingletonExposure = mbd.isSingleton() &&
    this.allowCircularReferences &&
    this.isSingletonCurrentlyInCreation(beanName);
30      if (earlySingletonExposure) {
31          if (this.logger.isTraceEnabled()) {
32              this.logger.trace("Eagerly caching bean '" + beanName + "' to
    allow for resolving potential circular references");
33          }
34
35          this.addSingletonFactory(beanName, () -> {
36              return this.getEarlyBeanReference(beanName, mbd, bean);
37          });
38      }
39  }
```

设置到三级缓存之后，对于其他依赖它的对象而言已经足够了（已经有内存地址了，可以根据对象引用定位到堆中对象），能够被认出来了。

addSingletonFactory方法的第二参数，通过匿名对象的方式，创建了一个 ObjectFactory 工厂，这个工厂只有一个方法，源码如下

```
1  @FunctionalInterface
2  public interface ObjectFactory<T> {
3      T getObject() throws BeansException;
4  }
```

一级缓存在哪里设置呢?

到这里我们发现三级缓存 singletonFactories 和 二级缓存 earlySingletonObjects 中的值都有出处了，那一级缓存在哪里设置的呢?

在类 DefaultSingletonBeanRegistry 中，可以发现这个 `addSingleton(String beanName, Object singletonObject)` 方法，代码如下：

```
// DefaultSingletonBeanRegistry.java

protected void addSingleton(String beanName, Object singletonObject) {
    synchronized (this.singletonObjects) {
            //添加至一级缓存，同时从二级、三级缓存中删除。
        this.singletonObjects.put(beanName, singletonObject);
        this.singletonFactories.remove(beanName);
        this.earlySingletonObjects.remove(beanName);
        this.registeredSingletons.add(beanName);
    }
}
```

高频面试题：Spring 如何解决循环依赖?

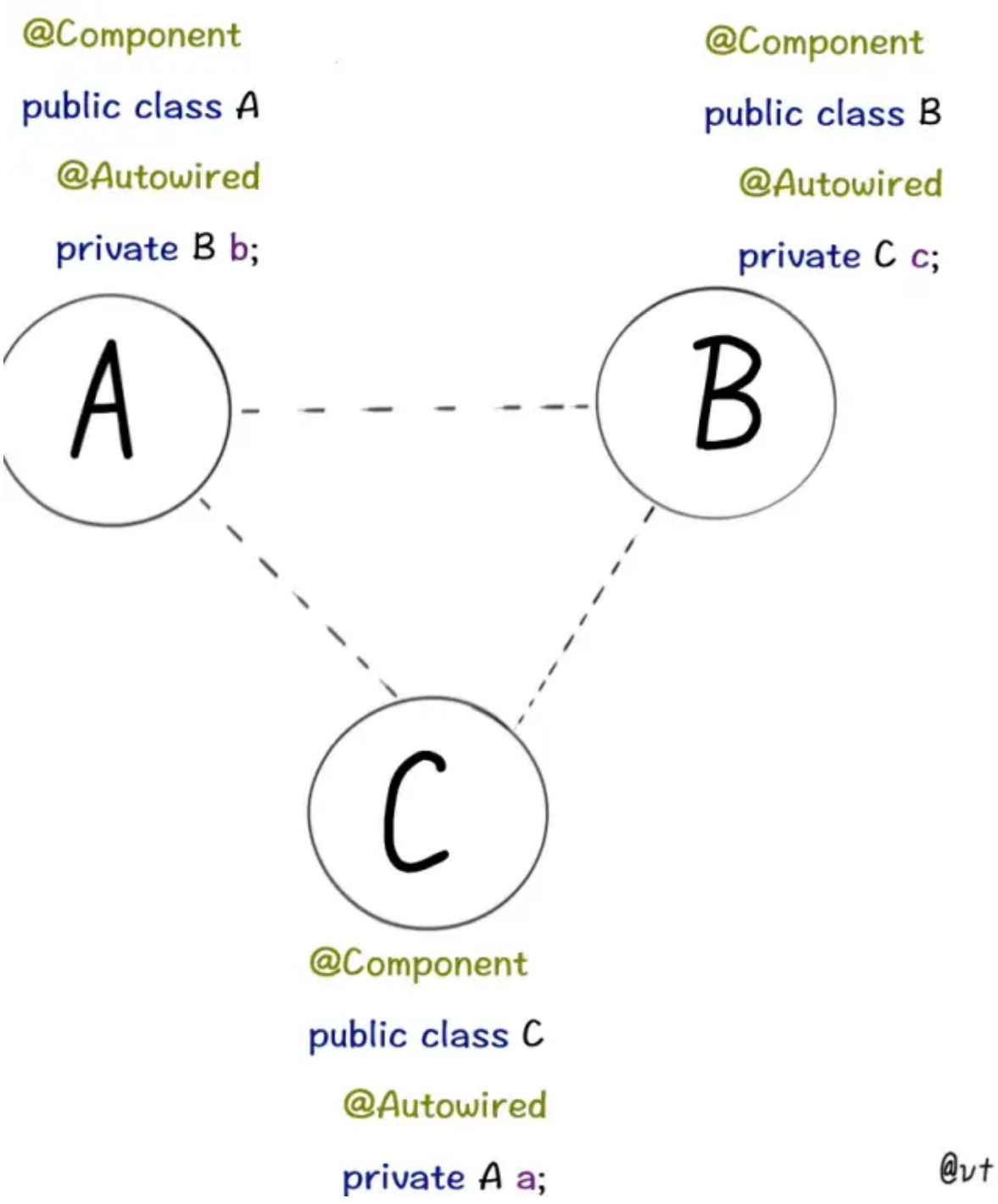
在关于Spring的面试中，我们经常会被问到一个问题：Spring是如何解决循环依赖的问题的。

这个问题算是关于Spring的一个高频面试题，因为如果不刻意研读，相信即使读过源码，面试者也不一定能够一下子思考出个中奥秘。

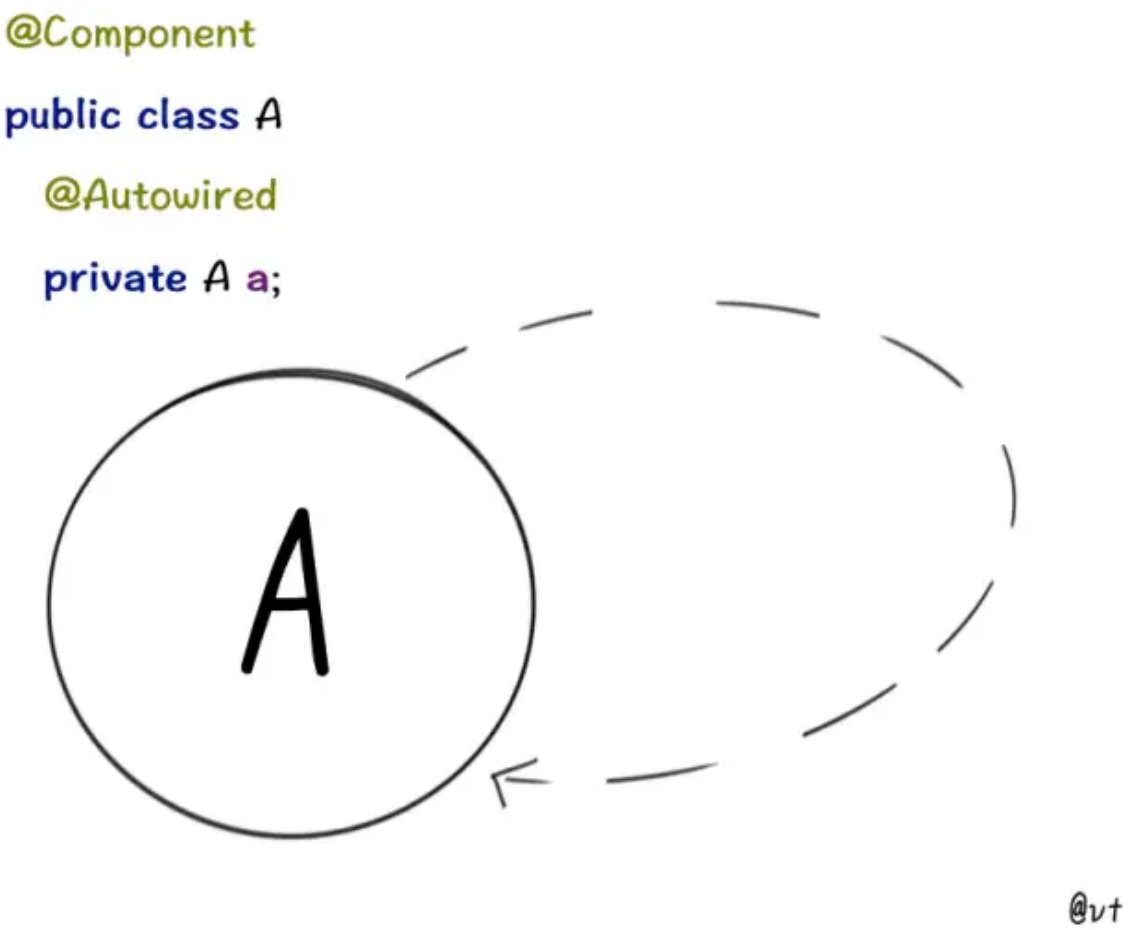
本文主要针对这个问题，从源码的角度对其实现原理进行讲解。

循环依赖的简单例子

比如几个Bean之间的互相引用：


@Component
public class A
@Autowired
private B b;
@Component
public class B
@Autowired
private C c;
A
B
C
@Component
public class C
@Autowired
private A a;
@vt

甚至自己“循环”依赖自己：



原型(Prototype)的场景是不支持循环依赖的

> 先说明前提：原型(Prototype)的场景是不支持循环依赖的.
>
> 单例的场景才能存在循环依赖

原型(Prototype)的场景通常会走到`AbstractBeanFactory`类中下面的判断，抛出异常。

```
if (isPrototypeCurrentlyInCreation(beanName)) {
    throw new BeanCurrentlyInCreationException(beanName);
}
```

原因很好理解，创建新的A时，发现要注入原型字段B，又创建新的B发现要注入原型字段A...

这就套娃了, Spring就先抛出了BeanCurrentlyInCreationException

什么是原型(Prototype)的场景?

通过如下方式，可以将该类的bean设置为原型模式

```
@Service
@Scope("prototype")
public class MyReportExporter extends AbstractReportExporter{
    ...
}
```

在Spring中，@Service默认都是单例的。

> 所谓单例，就是Spring的IOC机制只创建该类的一个实例，
>
> 每次请求，都会用这同一个实例进行处理，因此若存在全局变量，本次请求的值肯定会影响下一次请求时该变量的值。

若不想影响下次请求，就需要用到原型模式，即@Scope(“prototype”)

> 原型模式，指的是每次调用时，会重新创建该类的一个实例，比较类似于我们自己自己new的对象实例。

具体例子：循环依赖的代码片段

我们先看看当时出问题的代码片段：

```
@Service
publicclass TestService1 {

    @Autowired
    private TestService2 testService2;

    @Async
    public void test1() {
    }
}

@Service
publicclass TestService2 {

    @Autowired
    private TestService1 testService1;

    public void test2() {
    }
}
```

这两段代码中定义了两个Service类：`TestService1`和`TestService2`，

在TestService1中注入了TestService2的实例，同时在TestService2中注入了TestService1的实例，这里构成了`循环依赖`。

> 只不过，这不是普通的循环依赖，因为TestService1的test1方法上加了一个`@Async`注解。

大家猜猜程序启动后运行结果会怎样？

```
org.springframework.beans.factory.BeanCurrentlyInCreationException: Error creating bean with name 'testService1': Bean with name 'testService1' has been injected into other beans [testService2] in its raw version as part of a circular reference, but has eventually been wrapped. This means that said other beans do not use the final version of the bean. This is often the result of over-eager type matching - consider using 'getBeanNamesOfType' with the 'allowEagerInit' flag turned off, for example.
```

报错了。。。原因是出现了循环依赖。

「不科学呀，spring不是号称能解决循环依赖问题吗，怎么还会出现？」

如果把上面的代码稍微调整一下：

```
@Service
publicclass TestService1 {

    @Autowired
    private TestService2 testService2;

    public void test1() {
    }
}
```

把TestService1的test1方法上的`@Async`注解去掉，`TestService1`和`TestService2`都需要注入对方的实例，同样构成了循环依赖。

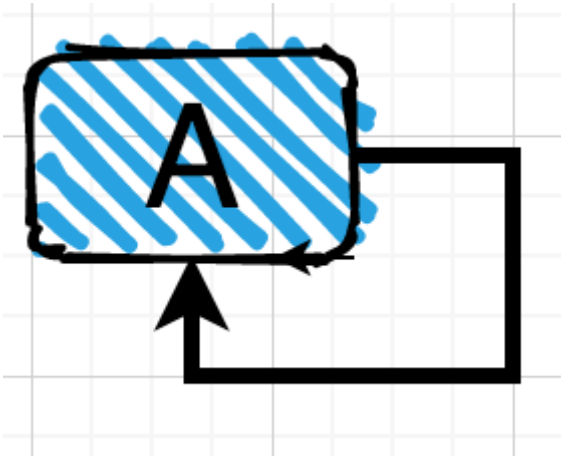
但是重新启动项目，发现它能够正常运行。这又是为什么？

带着这两个问题，让我们一起开始spring循环依赖的探秘之旅。

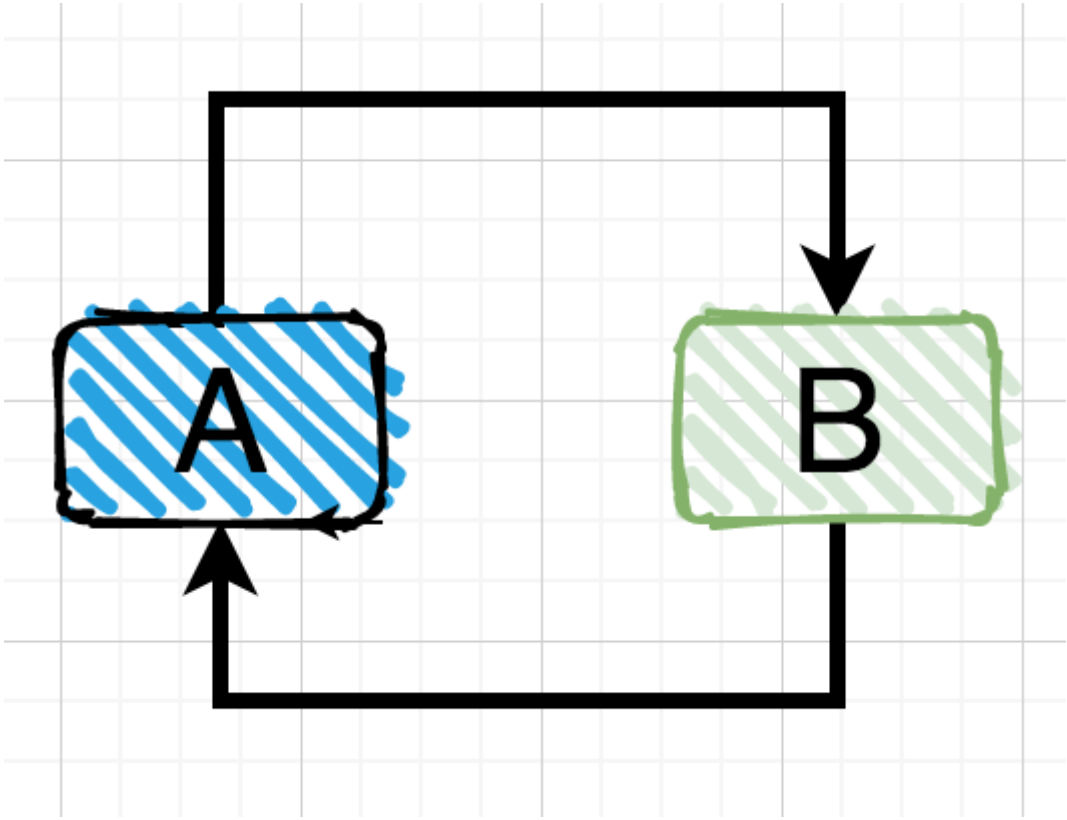
什么是循环依赖？

循环依赖：说白是一个或多个对象实例之间存在直接或间接的依赖关系，这种依赖关系构成了构成一个环形调用。

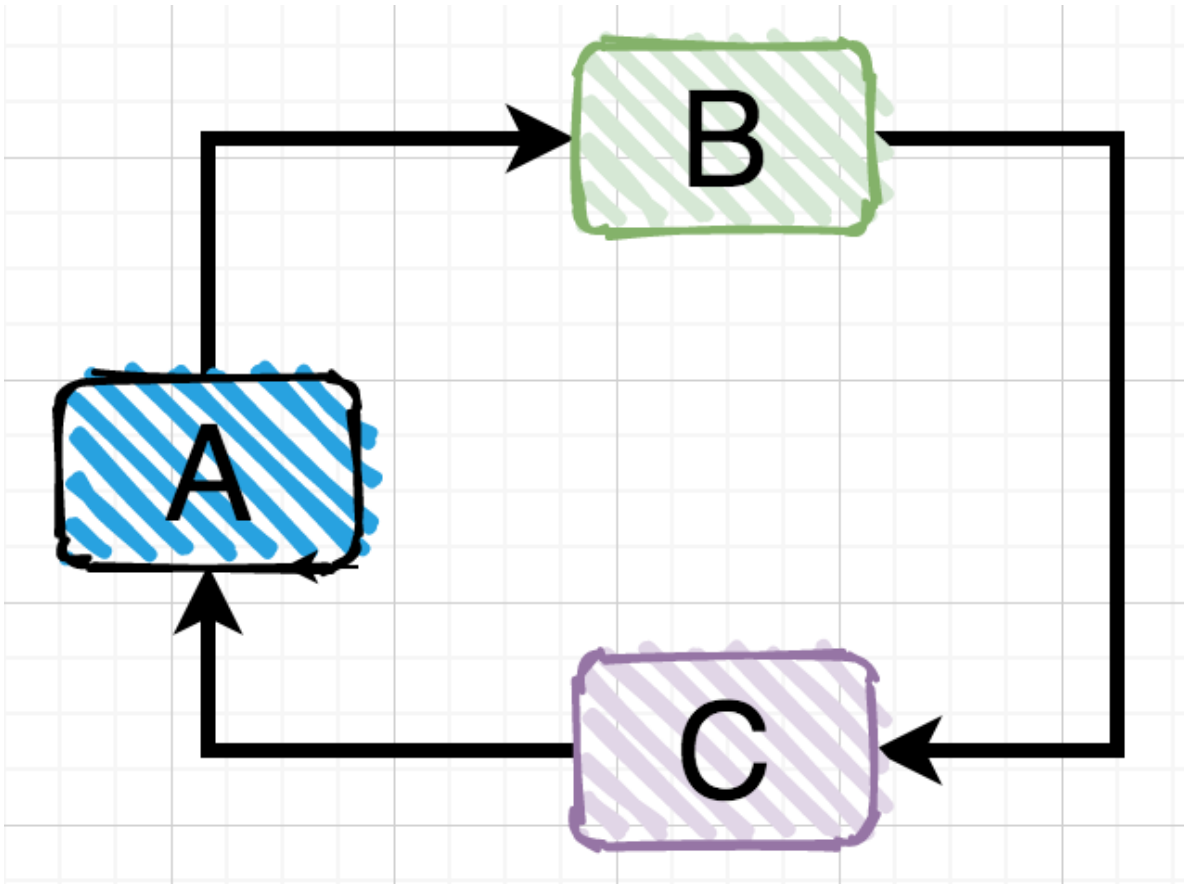
第一种情况：自己依赖自己的直接依赖



第二种情况：两个对象之间的直接依赖



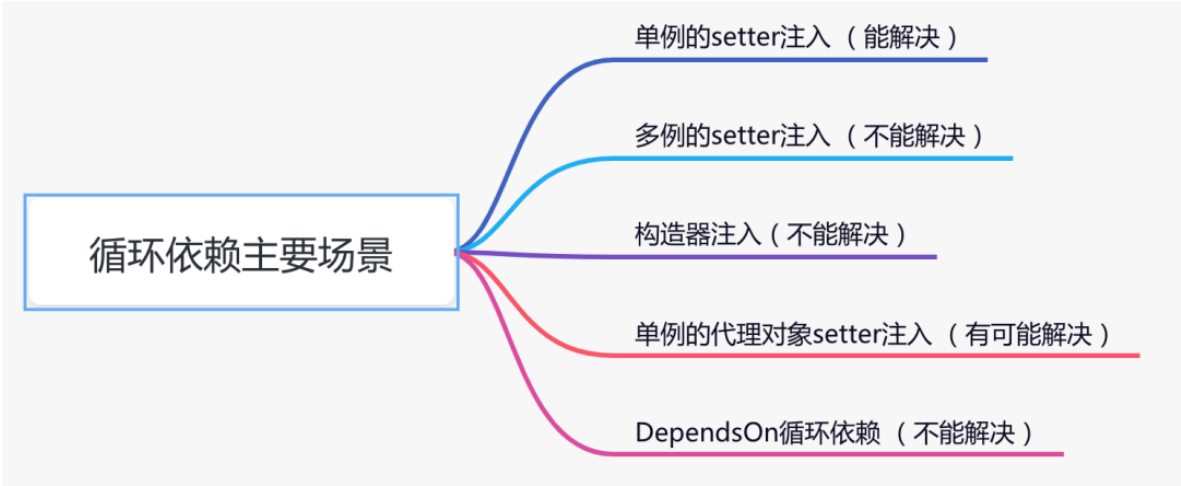
第三种情况：多个对象之间的间接依赖



前面两种情况的直接循环依赖比较直观，非常好识别，但是第三种间接循环依赖的情况有时候因为业务代码调用层级很深，不容易识别出来。

循环依赖的N种场景

spring中出现循环依赖主要有以下场景：



场景1：单例的setter注入

这种注入方式应该是spring用的最多的，代码如下：

```
@Service
publicclass TestService1 {

    @Autowired
    private TestService2 testService2;

    public void test1() {
    }
}


@Service
publicclass TestService2 {

    @Autowired
    private TestService1 testService1;

    public void test2() {
    }
}
```

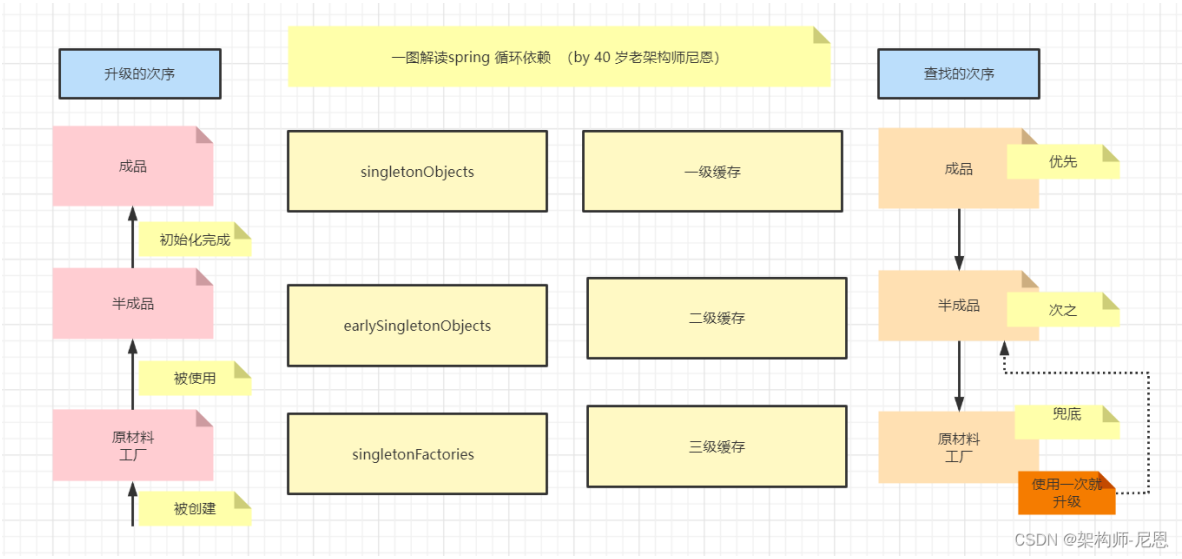
这是一个经典的循环依赖，但是它能正常运行，得益于spring的内部机制，

让我们根本无法感知它有问题，因为spring默默帮我们解决了。

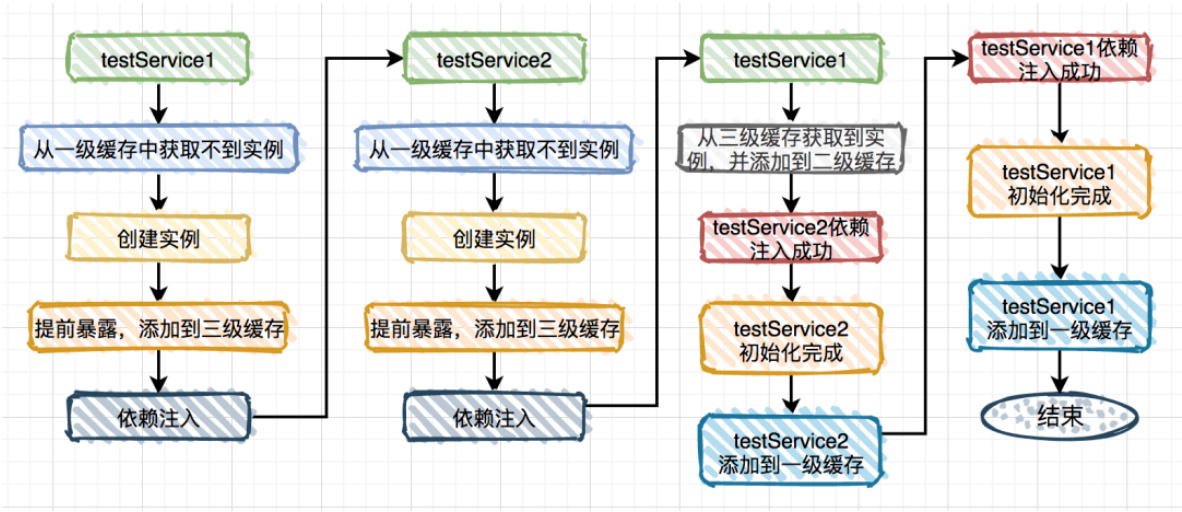
spring内部有三级缓存：

- **成品**：一级缓存 singletonObjects，用于保存实例化、注入、初始化完成的bean实例 ：
- **半成品**：二级缓存 earlySingletonObjects，用于保存实例化完成的bean实例
- **原材料工厂**： 三级缓存 singletonFactories，用于保存bean创建工厂，以便于后面扩展有机会创建代理对象。

一张图告诉你，spring 三级缓存之间的关系



一张图告诉你，spring是如何解决循环依赖的：



细心的朋友可能会发现在这种场景中**第二级缓存**作用不大。

那么问题来了，为什么要用第二级缓存呢?

试想一下，如果出现以下这种情况，我们要如何处理?

```
@Service
publicclass TestService1 {

    @Autowired
    private TestService2 testService2;
    @Autowired
    private TestService3 testService3;

    public void test1() {
    }
}


@Service
publicclass TestService2 {

```

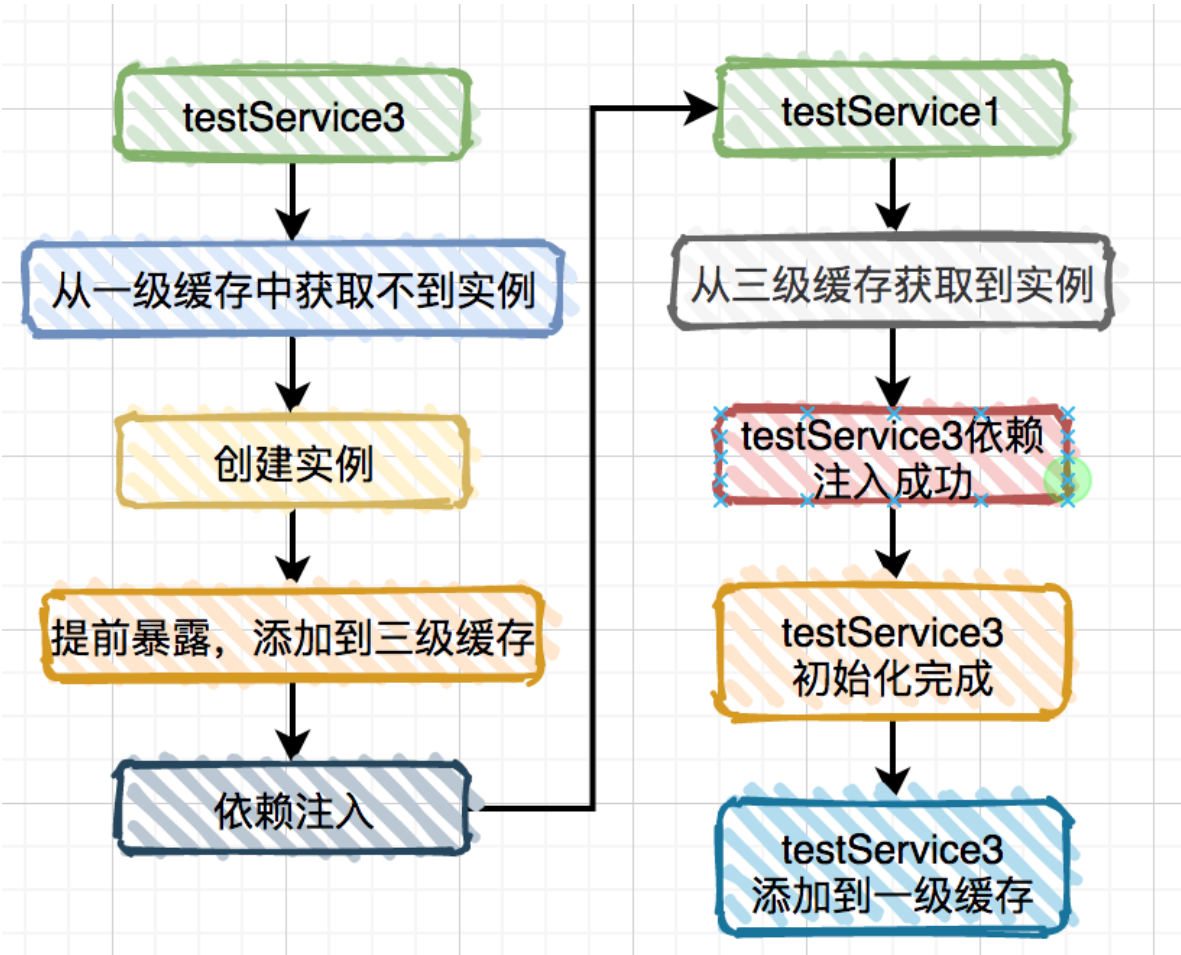
```
    @Autowired
    private TestService1 testService1;

    public void test2() {
    }
}


@Service
publicclass TestService3 {

    @Autowired
    private TestService1 testService1;

    public void test3() {
    }
}
```

TestService1依赖于TestService2和TestService3，

- 而TestService2依赖于TestService1，
- 同时TestService3也依赖于TestService1。

按照上图的流程可以把TestService1注入到TestService2，并且TestService1的实例是从第三级缓存中获取的。

假设不用第二级缓存，TestService1注入到TestService3的流程如图：



TestService1注入到TestService3又需要从第三级缓存中获取实例，而第三级缓存里保存的并非真正的实例对象，而是`ObjectFactory`对象。

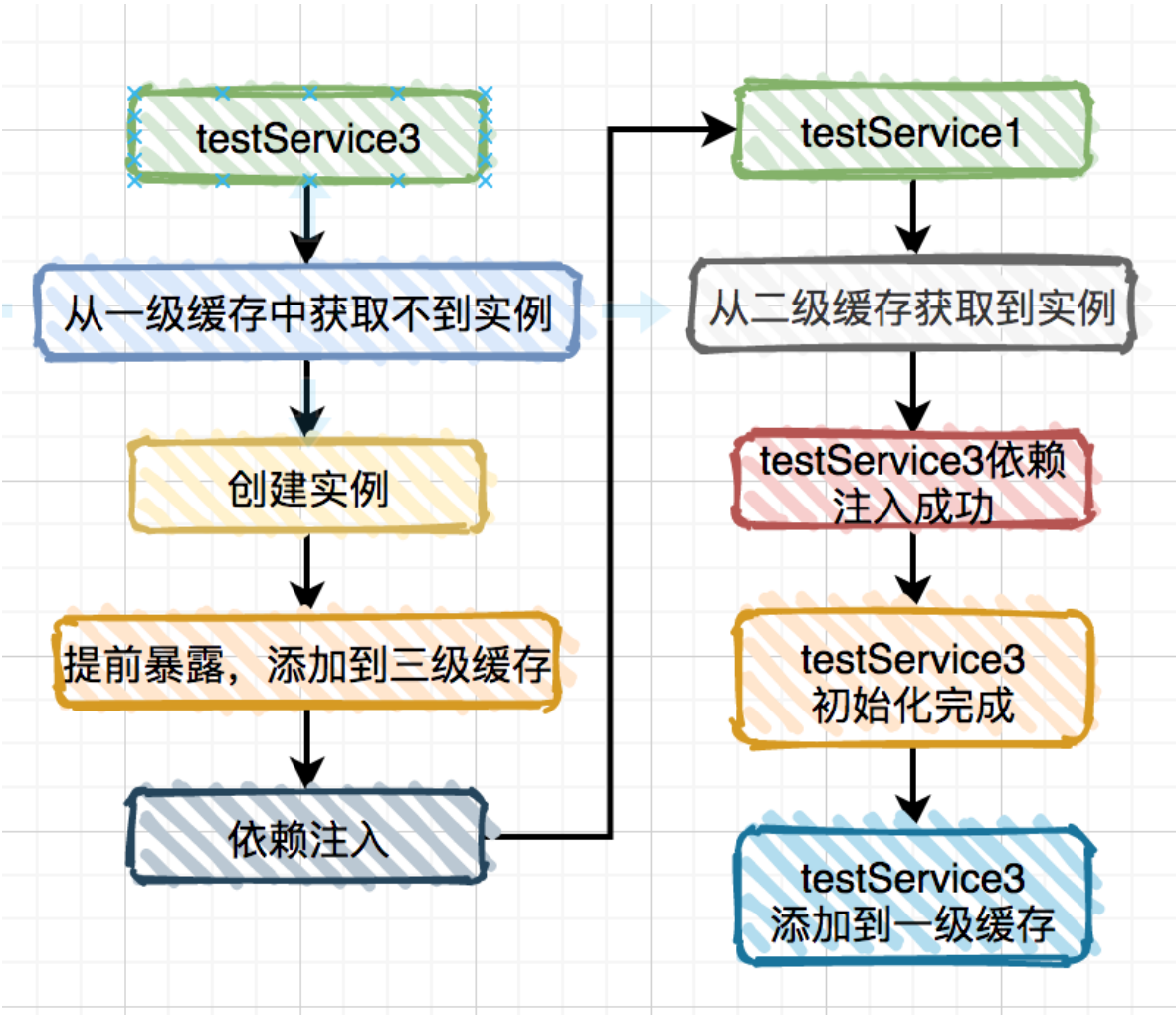
> 说白了，两次从三级缓存中获取都是`ObjectFactory`对象，而通过它创建的实例对象每次可能都不一样的。

这样不是有问题?

为了解决这个问题，spring引入的第二级缓存。

前一个图其实TestService1对象的实例已经被添加到第二级缓存中了，

而在TestService1注入到TestService3时，只用从第二级缓存中获取该对象即可。



还有个问题，第三级缓存中为什么要添加`ObjectFactory`对象，直接保存实例对象不行吗?

> 答：不行，因为假如你想对添加到三级缓存中的实例对象进行增强，直接用实例对象是行不通的。

针对这种场景spring是怎么做的呢?

答案就在`AbstractAutowireCapableBeanFactory`类`doCreateBean`方法的这段代码中：

```
if (earlySingletonExposure) {
    if (logger.isDebugEnabled()) {
        logger.debug( o: "Eagerly caching bean '" + beanName +
                "' to allow for resolving potential circular references");
    }
    addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));
}
```

它定义了一个匿名内部类，通过`getEarlyBeanReference`方法获取代理对象，其实底层是通过`AbstractAutoProxyCreator`类的`getEarlyBeanReference`生成代理对象。

场景二多例的setter注入

这种注入方法偶然会有，特别是在多线程的场景下，具体代码如下：

```
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
@Service
publicclass TestService1 {

    @Autowired
    private TestService2 testService2;

    public void test1() {
    }
}


@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
@Service
publicclass TestService2 {

    @Autowired
    private TestService1 testService1;

    public void test2() {
    }
}

```

很多人说这种情况spring容器启动会报错，其实是不对的，我非常负责任的告诉你程序能够正常启动。

为什么呢?

其实在`AbstractApplicationContext`类的`refresh`方法中告诉了我们答案，

它会调用`finishBeanFactoryInitialization`方法，该方法的作用是为了spring容器启动的时候提前初始化一些bean。该方法的内部又调用了`preInstantiateSingletons`方法

```
// Trigger initialization of all non-lazy singleton beans...
for (String beanName : beanNames) {
    RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);
    if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
        if (isFactoryBean(beanName)) {
            Object bean = getBean( name: FACTORY_BEAN_PREFIX + beanName);
            if (bean instanceof FactoryBean) {
                final FactoryBean<?> factory = (FactoryBean<?>) bean;
                boolean isEagerInit;
                if (System.getSecurityManager() != null && factory instanceof SmartFactoryBean) {
                    isEagerInit = AccessController.doPrivileged((PrivilegedAction<Boolean>)
                                    ((SmartFactoryBean<?>) factory)::isEagerInit,
                            getAccessControlContext());
```

标红的地方明显能够看出：**非抽象、单例** 并且**非懒加载的类**才能被提前初始bean。

而多例即`SCOPE_PROTOTYPE`类型的类，非单例，不会被提前初始化bean，所以程序能够正常启动。

如何让他提前初始化bean呢?

只需要再定义一个单例的类，在它里面注入TestService1

```
@Service
publicclass TestService3 {

    @Autowired
    private TestService1 testService1;
}
```

重新启动程序，执行结果：

```
Requested bean is currently in creation: Is there an unresolvable circular reference?
```

果然出现了循环依赖。

注意：这种循环依赖问题是无法解决的，因为它没有用缓存，每次都会生成一个新对象。

场景三：构造器注入

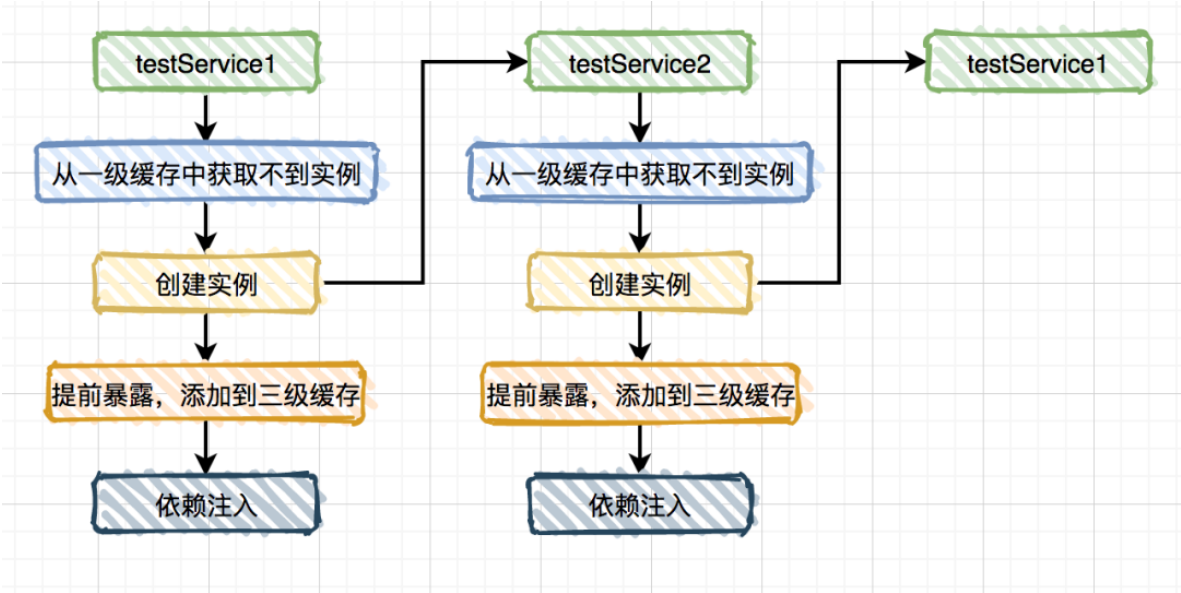
这种注入方式现在其实用的已经非常少了，但是我们还是有必要了解一下，看看如下代码：

```
@Service
publicclass TestService1 {

    public TestService1(TestService2 testService2) {
    }
}


@Service
publicclass TestService2 {

    public TestService2(TestService1 testService1) {
    }
}
```

运行结果：

```
Requested bean is currently in creation: Is there an unresolvable circular reference?
```

出现了循环依赖，为什么呢?



从图中的流程看出构造器注入没能添加到三级缓存，也没有使用缓存，所以也无法解决循环依赖问题。

场景四：单例的代理对象setter注入

这种注入方式其实也比较常用，比如平时使用：`@Async`注解的场景，会通过`AOP`自动生成代理对象。

我那位同事的问题也是这种情况。

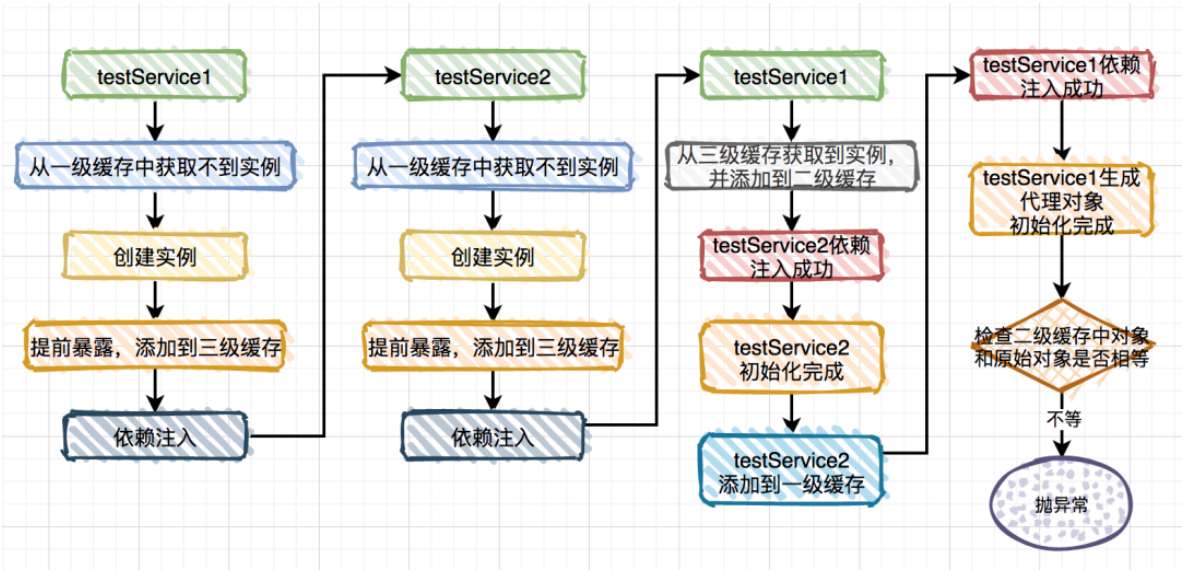
```
@Service
publicclass TestService1 {

    @Autowired
    private TestService2 testService2;

    @Async
    public void test1() {
    }
}
@Service
publicclass TestService2 {

    @Autowired
    private TestService1 testService1;

    public void test2() {
    }
}
```

从前面得知程序启动会报错，出现了循环依赖：

```
org.springframework.beans.factory.BeanCurrentlyInCreationException: Error creating bean with name 'testService1': Bean with name 'testService1' has been injected into other beans [testService2] in its raw version as part of a circular reference, but has eventually been wrapped. This means that said other beans do not use the final version of the bean. This is often the result of over-eager type matching - consider using 'getBeanNamesOfType' with the 'allowEagerInit' flag turned off, for example.
```

为什么会循环依赖呢?

答案就在下面这张图中：



说白了，bean初始化完成之后，后面还有一步去检查：第二级缓存 和 原始对象 是否相等。由于它对前面流程来说无关紧要，所以前面的流程图中省略了，但是在这里是关键点，我们重点说说：

```
if (earlySingletonExposure) {
    Object earlySingletonReference = getSingleton(beanName,  allowEarlyReference: false);
    if (earlySingletonReference != null) {
        if (exposedObject == bean) {
            exposedObject = earlySingletonReference;
        }
        else if (!this.allowRawInjectionDespiteWrapping && hasDependentBean(beanName)) {
            String[] dependentBeans = getDependentBeans(beanName);
            Set<String> actualDependentBeans = new LinkedHashSet<>(dependentBeans.length);
            for (String dependentBean : dependentBeans) {
                if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {
                    actualDependentBeans.add(dependentBean);
                }
            }
            if (!actualDependentBeans.isEmpty()) {
                throw new BeanCurrentlyInCreationException(beanName,
                        "Bean with name '" + beanName + "' has been injected into other beans [" +
                        StringUtils.collectionToCommaDelimitedString(actualDependentBeans) +
                        "] in its raw version as part of a circular reference, but has eventually been " +
                        "wrapped. This means that said other beans do not use the final version of the " +
                        "bean. This is often the result of over-eager type matching - consider using " +
                        "'getBeanNamesOfType' with the 'allowEagerInit' flag turned off, for example.");
            }
        }
    }
```

那位同事的问题正好是走到这段代码，发现第二级缓存 和 原始对象不相等，所以抛出了循环依赖的异常。

如果这时候把TestService1改个名字，改成：TestService6，其他的都不变。

```
@Service
publicclass TestService6 {

    @Autowired
    private TestService2 testService2;

    @Async
    public void test1() {
    }
}
```

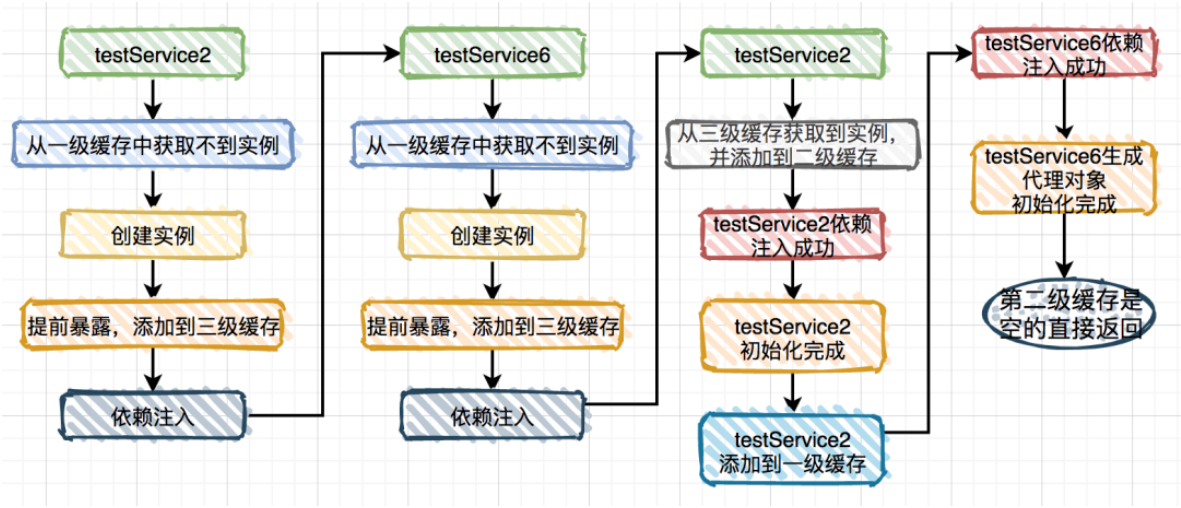
再重新启动一下程序，神奇般的好了。

what？ 这又是为什么？

这就要从spring的bean加载顺序说起了，默认情况下，spring是按照文件完整路径递归查找的，按路径+文件名排序，排在前面的先加载。所以TestService1比TestService2先加载，而改了文件名称之后，TestService2比TestService6先加载。

为什么TestService2比TestService6先加载就没问题呢？

答案在下面这张图中：



这种情况testService6中其实第二级缓存是空的，不需要跟原始对象判断，所以不会抛出循环依赖。

场景5：DependsOn循环依赖

还有一种有些特殊的场景，比如我们需要在实例化Bean A之前，先实例化Bean B，这个时候就可以使用`@DependsOn`注解。

```
@DependsOn(value = "testService2")
@Service
publicclass TestService1 {

    @Autowired
    private TestService2 testService2;

    public void test1() {
    }
}
@DependsOn(value = "testService1")
@Service
publicclass TestService2 {

    @Autowired
    private TestService1 testService1;

    public void test2() {
    }
}
```

程序启动之后，执行结果：

```
Circular depends-on relationship between 'testService2' and 'testService1'
```

这个例子中本来如果TestService1和TestService2都没有加`@DependsOn`注解是没问题的，反而加了这个注解会出现循环依赖问题。

这又是为什么?

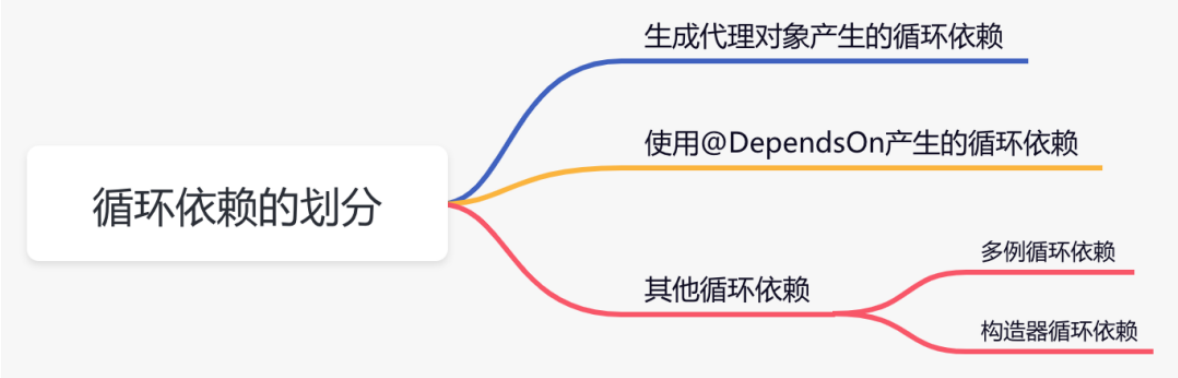
答案在`AbstractBeanFactory`类的`doGetBean`方法的这段代码中：

```
String[] dependsOn = mbd.getDependsOn();
if (dependsOn != null) {
    for (String dep : dependsOn) {
        if (isDependent(beanName, dep)) {
            throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                    "Circular depends-on relationship between '" + beanName + "' and '" + dep + "'");
        }
        registerDependentBean(dep, beanName);
        try {
            getBean(dep);
        }
        catch (NoSuchBeanDefinitionException ex) {
            throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                    "'" + beanName + "' depends on missing bean '" + dep + "'", ex);
        }
    }
}
```

它会检查dependsOn的实例有没有循环依赖，如果有循环依赖则抛异常。

总体策略：出现循环依赖如何解决?

项目中如果出现循环依赖问题，说明是spring默认无法解决的循环依赖，要看项目的打印日志，属于哪种循环依赖。目前包含下面几种情况：



生成代理对象产生的循环依赖 的解决方案：

这类循环依赖问题解决方法很多，主要有：

1. 使用`@Lazy`注解，延迟加载
2. 使用`@DependsOn`注解，指定加载先后关系
3. 修改文件名称，改变循环依赖类的加载顺序

使用@DependsOn产生的循环依赖 的解决方案：

这类循环依赖问题要找到`@DependsOn`注解循环依赖的地方，迫使它不循环依赖就可以解决问题。

多例循环依赖 的解决方案：

这类循环依赖问题可以通过把bean改成单例的解决。

构造器循环依赖 的解决方案：

这类循环依赖问题可以通过使用`@Lazy`注解解决

回答提要：

按照上面的方式回答， 起码120分。

> 但是答案太复杂， 如果上面的答案，记不住，就用下面的答案吧，至少也是100分。

问题： Spring是怎么解决循环依赖的?

首先，Spring 解决循环依赖有两个前提条件：

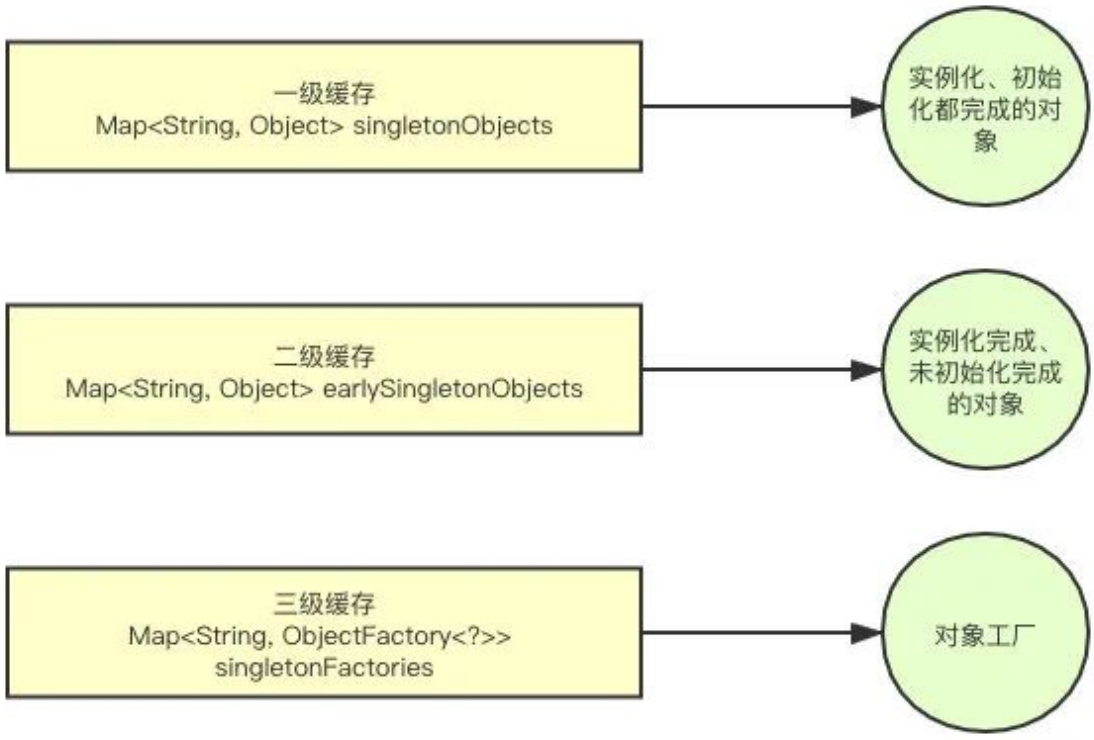
1. 不全是构造器方式的循环依赖
2. 必须是单例

基于上面的问题，我们知道Bean的生命周期，本质上解决循环依赖的问题就是三级缓存，通过三级缓存提前拿到未初始化的对象。

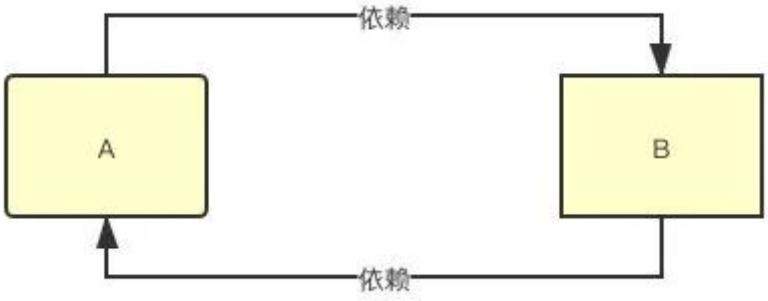
第三级缓存：用来保存一个对象工厂，提供一个匿名内部类，用于创建二级缓存中的对象

第二级缓存：用来保存实例化完成，但是未初始化完成的对象

第一级缓存：用来保存实例化、初始化都完成的对象

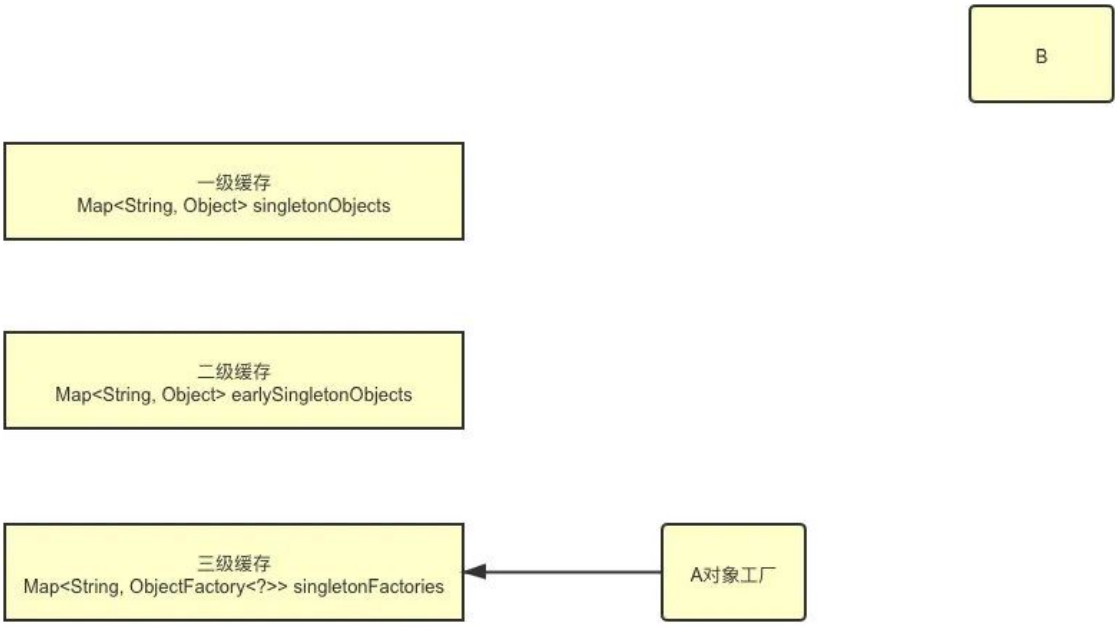


假设一个简单的循环依赖场景，A、B互相依赖。



A对象的创建过程：

1. 创建对象A，实例化的时候把A对象工厂放入三级缓存



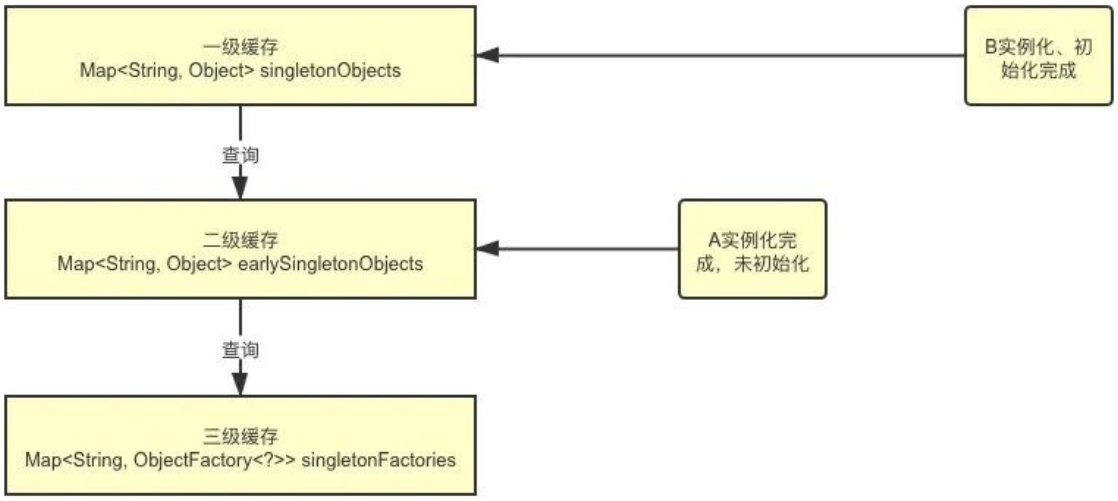
2 A注入属性时，发现依赖B，转而去实例化B

3 同样创建对象B，

注入属性时发现依赖A，一次从一级到三级缓存查询A，

从三级缓存通过对象工厂拿到A，把A放入二级缓存，同时删除三级缓存中的A，

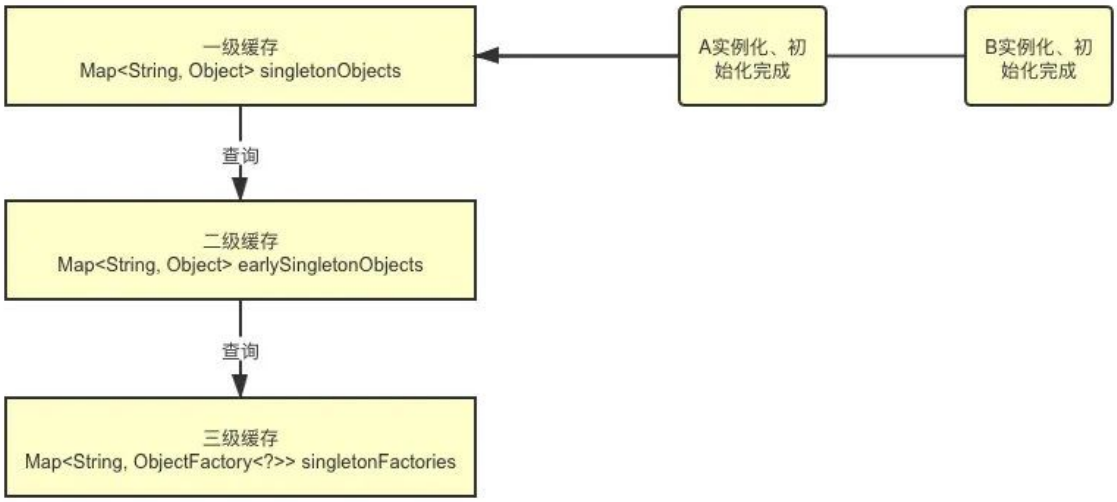
然后，B已经实例化并且初始化完成，把B放入一级缓存。



3 接着继续创建A，顺利从一级缓存拿到实例化且初始化完成的B对象，A对象创建也完成

删除二级缓存中的A，同时把A放入一级缓存

4 最后，一级缓存中保存着实例化、初始化都完成的A、B对象



因此，由于把实例化和初始化的流程分开了，所以如果都是用构造器的话，就没法分离这个操作，

所以都是构造器的话就无法解决循环依赖的问题了。

问题:为什么要三级缓存？二级不行吗？（重要)

不可以，主要是为了生成代理对象。

因为三级缓存中放的是生成具体对象的匿名内部类，他可以生成代理对象，也可以是普通的实例对象。

使用三级缓存主要是为了保证不管什么时候使用的都是一个对象。

假设只有二级缓存的情况，往二级缓存中放的显示一个普通的Bean对象，在`BeanPostProcessor`去生成代理对象之后，覆盖掉二级缓存中的普通Bean对象，那么多线程环境下可能取到的对象就不一致了。

Spring注解（8题目）

什么是基于Java的Spring注解配置? 给一些注解的例子

基于Java的配置，允许你在少量的Java注解的帮助下，进行你的大部分Spring配置而非通过XML文件。

以@Configuration 注解为例，它用来标记类可以当做一个bean的定义，被Spring IOC容器使用。

另一个例子是@Bean注解，它表示此方法将要返回一个对象，作为一个bean注册进Spring应用上下文。

```
@Configuration
public class StudentConfig {
    @Bean
    public StudentBean myStudent() {
        return new StudentBean();
    }
}
```

怎样开启注解装配?

注解装配在默认情况下是不开启的，为了使用注解装配，我们必须在Spring配置文件中配置`<context:annotation-config/>`元素。

@Component, @Controller, @Repository, @Service 有何区别?

@Component：这将 java 类标记为 bean。它是任何 Spring 管理组件的通用构造型。spring 的组件扫描机制现在可以将其拾取并将其拉入应用程序环境中。

@Controller：这将一个类标记为 Spring Web MVC 控制器。标有它的 Bean 会自动导入到 IoC 容器中。

@Service：此注解是组件注解的特化。它不会对 @Component 注解提供任何其他行为。您可以在服务层类中使用 @Service 而不是 @Component，因为它以更好的方式指定了意图。

@Repository：这个注解是具有类似用途和功能的 @Component 注解的特化。它为 DAO 提供了额外的好处。它将 DAO 导入 IoC 容器，并使未经检查的异常有资格转换为 Spring DataAccessException。

@Required 注解有什么作用

这个注解表明bean的属性必须在配置的时候设置，通过一个bean定义的显式的属性值或通过自动装配，若@Required注解的bean属性未被设置，容器将抛出BeanInitializationException。示例：

```
public class Employee {
    private String name;
    @Required
    public void setName(String name){
        this.name=name;
    }
    public string getName(){
        return name;
    }
}
```

@Autowired 注解有什么作用

@Autowired默认是按照类型装配注入的，默认情况下它要求依赖对象必须存在（可以设置它required属性为false）。@Autowired 注解提供了更细粒度的控制，包括在何处以及如何完成自动装配。它的用法和@Required一样，修饰setter方法、构造器、属性或者具有任意名称和/或多个参数的PN方法。

```java
public class Employee {
    private String name;
    @Autowired
    public void setName(String name) {
        this.name=name;
    }
    public string getName(){
        return name;
    }
}
```

@Autowired和@Resource之间的区别

@Autowired可用于：构造函数、成员变量、Setter方法

@Autowired和@Resource之间的区别

- @Autowired默认是按照类型装配注入的，默认情况下它要求依赖对象必须存在（可以设置它required属性为false）。
- @Resource默认是按照名称来装配注入的，只有当找不到与名称匹配的bean才会按照类型来装配注入。

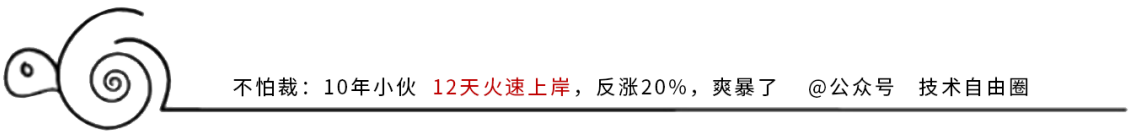
@Qualifier 注解有什么作用

当您创建多个相同类型的 bean 并希望仅使用属性装配其中一个 bean 时，您可以使用@Qualifier 注解和 @Autowired 通过指定应该装配哪个确切的 bean 来消除歧义。

@RequestMapping 注解有什么用?

@RequestMapping 注解用于将特定 HTTP 请求方法映射到将处理相应请求的控制器中的特定类/方法。此注释可应用于两个级别：

- 类级别：映射请求的 URL
- 方法级别：映射 URL 以及 HTTP 请求方法



Spring数据访问（14）

解释对象/关系映射集成模块

Spring 通过提供ORM模块，支持我们在直接JDBC之上使用一个对象/关系映射映射(ORM)工具，Spring支持集成主流的ORM框架，如Hiberate，JDO和 iBATIS，JPA，TopLink，JDO，OJB 。Spring的事务管理同样支持以上所有ORM框架及JDBC。

在Spring框架中如何更有效地使用JDBC?

使用Spring JDBC 框架，资源管理和错误处理的代价都会被减轻。所以开发者只需写statements 和 queries从数据存取数据，JDBC也可以在Spring框架提供的模板类的帮助下更有效地被使用，这个模板叫JdbcTemplate

解释JDBC抽象和DAO模块

通过使用JDBC抽象和DAO模块，保证数据库代码的简洁，并能避免数据库资源错误关闭导致的问题，它在各种不同的数据库的错误信息之上，提供了一个统一的异常访问层。它还利用Spring的AOP 模块给Spring应用中的对象提供事务管理服务。

spring DAO 有什么用?

Spring DAO（数据访问对象） 使得 JDBC，Hibernate 或 JDO 这样的数据访问技术更容易以一种统一的方式工作。这使得用户容易在持久性技术之间切换。它还允许您在编写代码时，无需考虑捕获每种技术不同的异常。

spring JDBC API 中存在哪些类?

JdbcTemplate

SimpleJdbcTemplate

NamedParameterJdbcTemplate

SimpleJdbcInsert

SimpleJdbcCall

JdbcTemplate是什么

JdbcTemplate 类提供了很多便利的方法解决诸如把数据库数据转变成基本数据类型或对象，执行写好的或可调用的数据库操作语句，提供自定义的数据错误处理。

使用Spring通过什么方式访问Hibernate？使用 Spring 访问 Hibernate 的方法有哪些?

在Spring中有两种方式访问Hibernate：

- 使用 Hibernate 模板和回调进行控制反转
- 扩展 HibernateDAOSupport 并应用 AOP 拦截器节点

如何通过HibernateDaoSupport将Spring和Hibernate结合起来?

用Spring的 SessionFactory 调用 LocalSessionFactory。集成过程分三步：

- 配置the Hibernate SessionFactory
- 继承HibernateDaoSupport实现一个DAO
- 在AOP支持的事务中装配

Spring支持的事务管理类型， spring 事务实现方式有哪些?

Spring支持两种类型的事务管理：

编程式事务管理：这意味你通过编程的方式管理事务，给你带来极大的灵活性，但是难维护。

声明式事务管理：这意味着你可以将业务代码和事务管理分离，你只需用注解和XML配置来管理事务。

Spring事务的实现方式和实现原理

Spring事务的本质其实就是数据库对事务的支持，没有数据库的事务支持，spring是无法提供事务功能的。真正的数据库层的事务提交和回滚是通过binlog或者redo log实现的。

说一下Spring的事务传播行为

spring事务的传播行为说的是，当多个事务同时存在的时候，spring如何处理这些事务的行为。

> ① PROPAGATION_REQUIRED：如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务，该设置是最常用的设置。
>
> ② PROPAGATION_SUPPORTS：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就以非事务执行。
>
> ③ PROPAGATION_MANDATORY：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就抛出异常。
>
> ④ PROPAGATION_REQUIRES_NEW：创建新事务，无论当前存不存在事务，都创建新事务。
>
> ⑤ PROPAGATION_NOT_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
>
> ⑥ PROPAGATION_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。
>
> 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，则按REQUIRED属性执行。

说一下 spring 的事务隔离?

spring 有五大隔离级别，默认值为 ISOLATION_DEFAULT（使用数据库的设置），其他四个隔离级别和数据库的隔离级别一致：

1. ISOLATION_DEFAULT：用底层数据库的设置隔离级别，数据库设置的是什么我就用什么；
2. ISOLATION_READ_UNCOMMITTED：未提交读，最低隔离级别、事务未提交前，就可被其他事务读取（会出现幻读、脏读、不可重复读）；
3. ISOLATION_READ_COMMITTED：提交读，一个事务提交后才能被其他事务读取到（会造成幻读、不可重复读），SQL server 的默认级别；
4. ISOLATION_REPEATABLE_READ：可重复读，保证多次读取同一个数据时，其值都和事务开始时候的内容是一致，禁止读取到别的事务未提交的数据（会造成幻读），MySQL 的默认级别；
5. ISOLATION_SERIALIZABLE：序列化，代价最高最可靠的隔离级别，该隔离级别能防止脏读、不可重复读、幻读。

脏读：表示一个事务能够读取另一个事务中还未提交的数据。比如，某个事务尝试插入记录 A，此时该事务还未提交，然后另一个事务尝试读取到了记录 A。

不可重复读：是指在一个事务内，多次读同一数据。

幻读：指同一个事务内多次查询返回的结果集不一样。比如同一个事务 A 第一次查询时候有 n 条记录，但是第二次同等条件下查询却有 n+1 条记录，这就好像产生了幻觉。发生幻读的原因也是另外一个事务新增或者删除或者修改了第一个事务结果集里面的数据，同一个记录的数据内容被修改了，所有数据行的记录就变多或者变少了。

Spring框架的事务管理有哪些优点?

- 为不同的事务API 如 JTA，JDBC，Hibernate，JPA 和JDO，提供一个不变的编程模式。
- 为编程式事务管理提供了一套简单的API而不是一些复杂的事务API
- 支持声明式事务管理。
- 和Spring各种数据访问抽象层很好得集成。

你更倾向用那种事务管理类型?

大多数Spring框架的用户选择声明式事务管理，因为它对应用代码的影响最小，因此更符合一个无侵入的轻量级容器的思想。声明式事务管理要优于编程式事务管理，虽然比编程式事务管理（这种方式允许你通过代码控制事务）少了一点灵活性。唯一不足地方是，最细粒度只能作用到方法级别，无法做到像编程式事务那样可以作用到代码块级别。

Spring面向切面编程(AOP)（13）

什么是AOP

OOP(Object-Oriented Programming)面向对象编程，允许开发者定义纵向的关系，但并适用于定义横向的关系，导致了大量代码的重复，而不利于各个模块的重用。

AOP(Aspect-Oriented Programming)，一般称为面向切面编程，作为面向对象的一种补充，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。可用于权限认证、日志、事务处理等。

Spring AOP and AspectJ AOP 有什么区别？AOP 有哪些实现方式?

AOP实现的关键在于 代理模式，AOP代理主要分为静态代理和动态代理。静态代理的代表为AspectJ；动态代理则以Spring AOP为代表。

（1）AspectJ是静态代理的增强，所谓静态代理，就是AOP框架会在编译阶段生成AOP代理类，因此也称为编译时增强，他会在编译阶段将AspectJ(切面)织入到Java字节码中，运行的时候就是增强之后的AOP对象。

（2）Spring AOP使用的动态代理，所谓的动态代理就是说AOP框架不会去修改字节码，而是每次运行时在内存中临时为方法生成一个AOP对象，这个AOP对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。

JDK动态代理和CGLIB动态代理的区别

Spring AOP中的动态代理主要有两种方式，JDK动态代理和CGLIB动态代理：

- JDK动态代理只提供接口的代理，不支持类的代理。核心InvocationHandler接口和Proxy类，InvocationHandler 通过invoke()方法反射来调用目标类中的代码，动态地将横切逻辑和业务编织在一起；接着，Proxy利用 InvocationHandler动态创建一个符合某一接口的的实例, 生成目标类的代理对象。
- 如果代理类没有实现 InvocationHandler 接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成指定类的一个子类对象，并覆盖其中特定方法并添加增强代码，从而实现AOP。CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动态代理的。

静态代理与动态代理区别在于生成AOP代理对象的时机不同，相对来说AspectJ的静态代理方式具有更好的性能，但是AspectJ需要特定的编译器进行处理，而Spring AOP则无需特定的编译器处理。

> InvocationHandler 的 invoke(Object proxy,Method method,Object[] args)：proxy是最终生成的代理实例; method 是被代理目标实例的某个具体方法; args 是被代理目标实例某个方法的具体入参，在方法反射调用时使用。

如何理解 Spring 中的代理?

将 Advice 应用于目标对象后创建的对象称为代理。在客户端对象的情况下，目标对象和代理对象是相同的。

Advice + Target Object = Proxy

解释一下Spring AOP里面的几个名词

（1）切面（Aspect）：切面是通知和切点的结合。通知和切点共同定义了切面的全部内容。 在Spring AOP中，切面可以使用通用类（基于模式的风格） 或者在普通类中以 @AspectJ 注解来实现。

（2）连接点（Join point）：指方法，在Spring AOP中，一个连接点 总是 代表一个方法的执行。 应用可能有数以千计的时机应用通知。这些时机被称为连接点。连接点是在应用执行过程中能够插入切面的一个点。这个点可以是调用方法时、抛出异常时、甚至修改一个字段时。切面代码可以利用这些点插入到应用的正常流程之中，并添加新的行为。

（3）通知（Advice）：在AOP术语中，切面的工作被称为通知。

（4）切入点（Pointcut）：切点的定义会匹配通知所要织入的一个或多个连接点。我们通常使用明确的类和方法名称，或是利用正则表达式定义所匹配的类和方法名称来指定这些切点。

（5）引入（Introduction）：引入允许我们向现有类添加新方法或属性。

（6）目标对象（Target Object）： 被一个或者多个切面（aspect）所通知（advise）的对象。它通常是一个代理对象。也有人把它叫做 被通知（adviced） 对象。 既然Spring AOP是通过运行时代理实现的，这个对象永远是一个 被代理（proxied） 对象。

（7）织入（Weaving）：织入是把切面应用到目标对象并创建新的代理对象的过程。在目标对象的生命周期里有多少个点可以进行织入：

- 编译期：切面在目标类编译时被织入。AspectJ的织入编译器是以这种方式织入切面的。
- 类加载期：切面在目标类加载到JVM时被织入。需要特殊的类加载器，它可以在目标类被引入应用之前增强该目标类的字节码。AspectJ5的加载时织入就支持以这种方式织入切面。
- 运行期：切面在应用运行的某个时刻被织入。一般情况下，在织入切面时，AOP容器会为目标对象动态地创建一个代理对象。SpringAOP就是以这种方式织入切面。

Spring在运行时通知对象

通过在代理类中包裹切面，Spring在运行期把切面织入到Spring管理的bean中。代理封装了目标类，并拦截被通知方法的调用，再把调用转发给真正的目标bean。当代理拦截到方法调用时，在调用目标bean方法之前，会执行切面逻辑。

直到应用需要被代理的bean时，Spring才创建代理对象。如果使用的是ApplicationContext的话，在ApplicationContext从BeanFactory中加载所有bean的时候，Spring才会创建被代理的对象。因为Spring运行时才创建代理对象，所以我们不需要特殊的编译器来织入SpringAOP的切面。

Spring只支持方法级别的连接点

因为Spring基于动态代理，所以Spring只支持方法连接点。Spring缺少对字段连接点的支持，而且它不支持构造器连接点。方法之外的连接点拦截功能，我们可以利用Aspect来补充。

在Spring AOP 中，关注点和横切关注的区别是什么？在 spring aop 中 concern 和 cross-cutting concern 的不同之处

关注点（concern）是应用中一个模块的行为，一个关注点可能会被定义成一个我们想实现的一个功能。

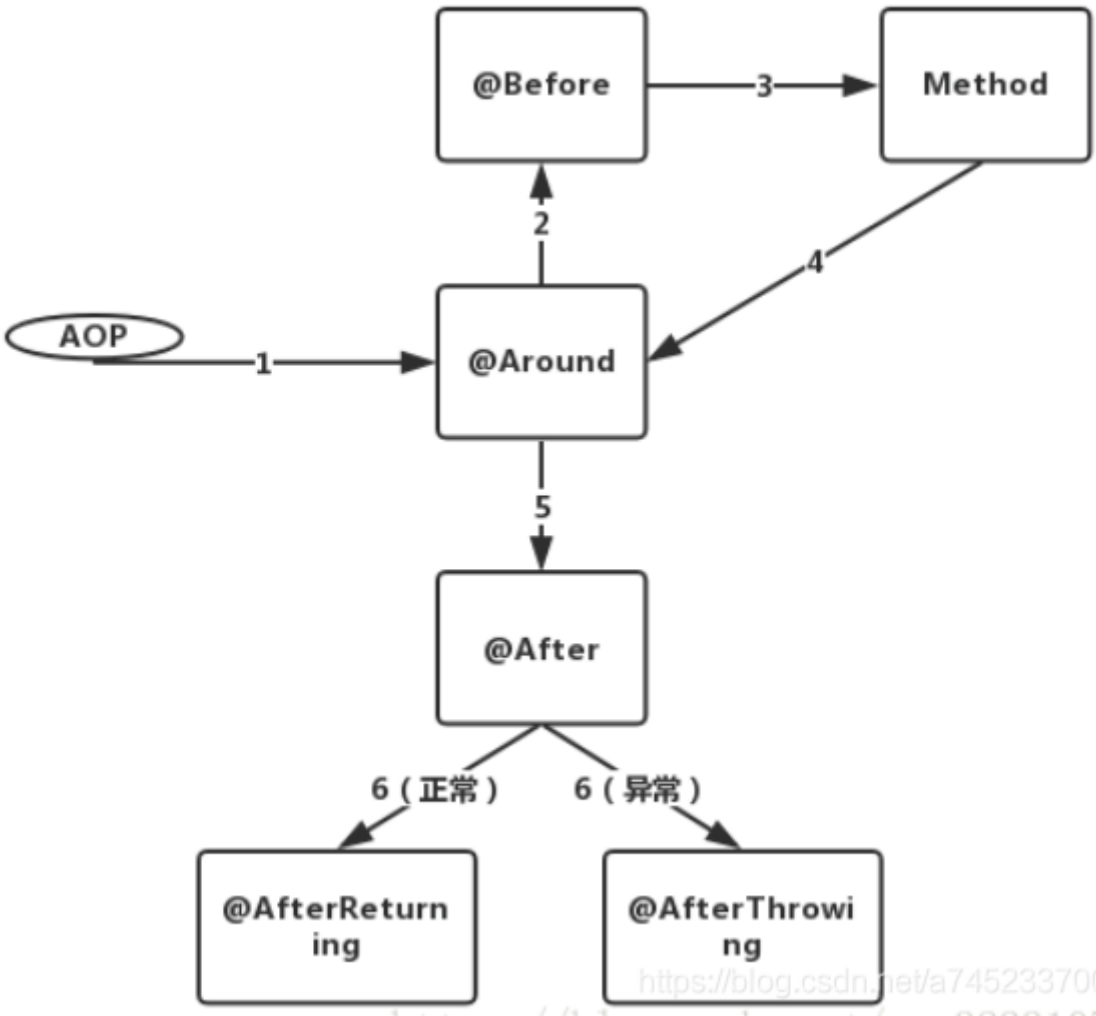
横切关注点（cross-cutting concern）是一个关注点，此关注点是整个应用都会使用的功能，并影响整个应用，比如日志，安全和数据传输，几乎应用的每个模块都需要的功能。因此这些都属于横切关注点。

Spring通知有哪些类型?

在AOP术语中，切面的工作被称为通知，实际上是程序执行时要通过SpringAOP框架触发的代码段。

Spring切面可以应用5种类型的通知：

1. 前置通知（Before）：在目标方法被调用之前调用通知功能；
2. 后置通知（After）：在目标方法完成之后调用通知，此时不会关心方法的输出是什么；
3. 返回通知（After-returning ）：在目标方法成功执行之后调用通知；
4. 异常通知（After-throwing）：在目标方法抛出异常后调用通知；
5. 环绕通知（Around）：通知包裹了被通知的方法，在被通知的方法调用之前和调用之后执行自定义的行为。



> 同一个aspect，不同advice的执行顺序：
>
> ①没有异常情况下的执行顺序：

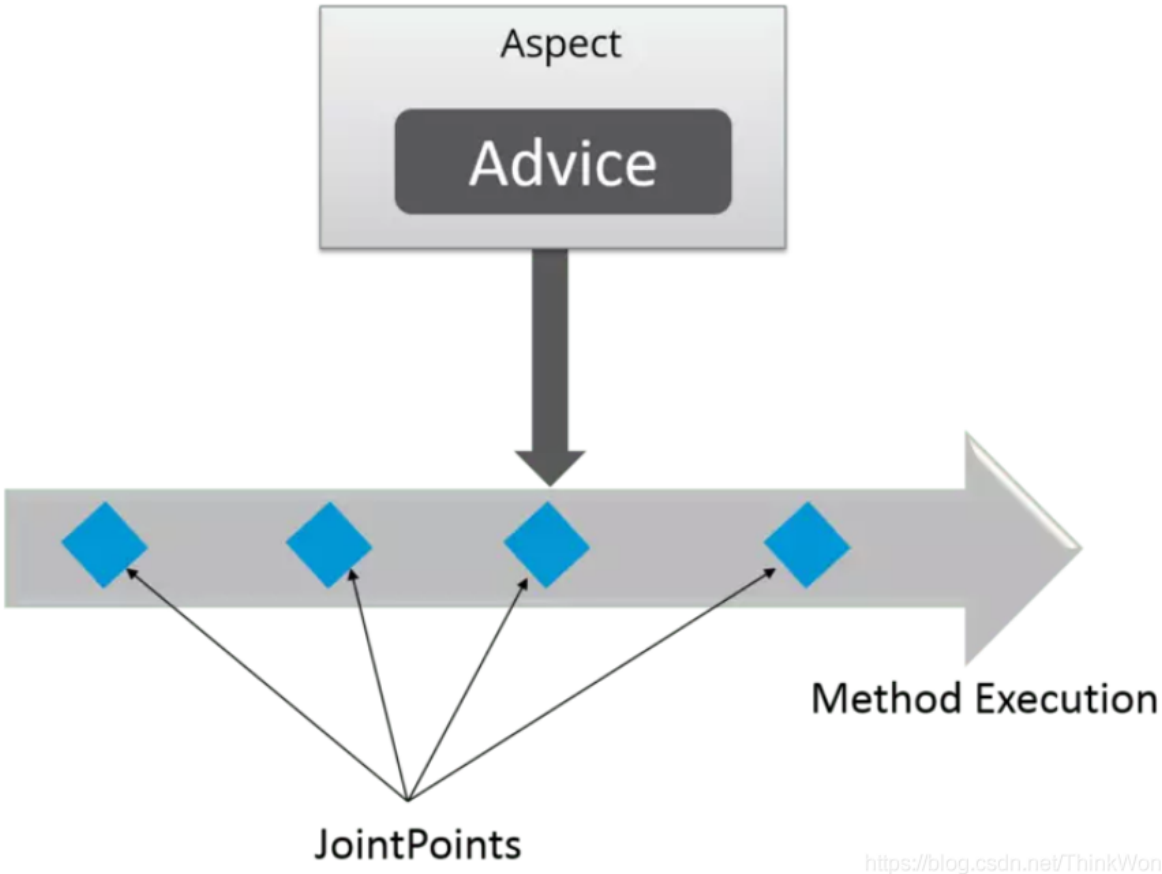
> around before advice
> before advice
> target method 执行
> around after advice
> after advice
> afterReturning
>
> ②有异常情况下的执行顺序：
>
> around before advice
> before advice
> target method 执行
> around after advice
> after advice
> afterThrowing:异常发生
> java.lang.RuntimeException: 异常发生

什么是切面 Aspect?

aspect 由 pointcount 和 advice 组成，切面是通知和切点的结合。 它既包含了横切逻辑的定义, 也包括了连接点的定义. Spring AOP 就是负责实施切面的框架, 它将切面所定义的横切逻辑编织到切面所指定的连接点中.
AOP 的工作重心在于如何将增强编织目标对象的连接点上, 这里包含两个工作:

- 如何通过 pointcut 和 advice 定位到特定的 joinpoint 上
- 如何在 advice 中编写切面代码.

可以简单地认为, 使用 @Aspect 注解的类就是切面.



解释基于XML Schema方式的切面实现

在这种情况下，切面由常规类以及基于XML的配置实现。

解释基于注解的切面实现

在这种情况下(基于@AspectJ的实现)，涉及到的切面声明的风格与带有java5标注的普通java类一致。

有几种不同类型的自动代理?

BeanNameAutoProxyCreator

DefaultAdvisorAutoProxyCreator

Metadata autoproxying

大厂面试题：读过Spring源码吗？说说Spring事务是怎么实现的?

说在前面

在40岁老架构师 尼恩的**读者交流群**(50+)中，最近有小伙伴拿到了一线互联网企业如阿里、美团、极兔、有赞、希音的面试资格，Spring事务源码的面试题，经常遇到：

> (1）spring什么情况下进行事务回滚?
>
> (2) spring 事务的传播行为有哪些 ?

与之类似的、其他小伙伴遇到过的问题还有：

> 读过Spring源码吗？说说Spring事务是怎么实现的?

Spring事务的面试题，绝对是面试的核心重点， 也是核心难点。

这里尼恩给大家做一下系统化、体系化梳理，使得大家可以充分展示一下大家雄厚的 “技术肌肉”，**让面试官爱到 “不能自已、口水直流”。**

也一并把这个题目以及参考答案，收入咱们的 《尼恩Java面试宝典》V60版本，供后面的小伙伴参考，提升大家的 3高 架构、设计、开发水平。

> 注：本文以 PDF 持续更新，最新尼恩 架构笔记、面试题 的PDF文件，请从这里获取：[码云]()

什么是数据库事务

数据库事务(Database Transaction) ，是指作为单个逻辑工作单元执行的一系列操作，要么完全地执行，要么完全地不执行。

简单来说： 事务是逻辑上的一组操作，要么都执行，要么都不执行。

通过事务，至少可以实现2点：

（1） 操作的原子性

（2）数据一致性。

我们系统的每个业务方法可能包括了多个原子性的数据库操作，比如下面的 `savePerson()` 方法中就有两个原子性的数据库操作。

这些原子性的数据库操作是有依赖的，它们要么都执行，要不就都不执行。

```
public void savePerson() {
        personDao.save(person);
        personDetailDao.save(personDetail);
    }
```

事务就是保证这两个关键操作要么都成功，要么都要失败。 最经典也经常被拿出来说例子就是**转账**了。

假如小明要给小红转账 1000 元，这个转账会涉及到两个关键操作就是：

> 1. 将小明的余额减少 1000 元。
> 2. 将小红的余额增加 1000 元。

万一在这两个操作之间突然出现错误比如银行系统崩溃或者网络故障，导致小明余额减少而小红的余额没有增加，这样就不对了。

```
public class OrdersService {
    private AccountDao accountDao;

    public void setOrdersDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }

  @Transactional(propagation = Propagation.REQUIRED,
                isolation = Isolation.DEFAULT, readOnly = false, timeout =
-1)
    public void accountMoney() {
    //小红账户多1000
        accountDao.addMoney(1000,xiaohong);
        //模拟突然出现的异常，比如银行中可能为突然停电等等
    //如果没有配置事务管理的话会造成，小红账户多了1000而小明账户没有少钱
        int i = 10 / 0;
        //小王账户少1000
        accountDao.reduceMoney(1000,xiaoming);
    }
}
```

数据库事务的 ACID 四大特性

数据库事务的 ACID 四大特性是事务的基础，下面简单来了解一下。

事务的执行具备四大特征：

1、Atomic 原子性

事务必须是一个原子的操作序列单元，事务中包含的各项操作在一次执行过程中，要么全部执行成功，要么全部不执行，任何一项失败，整个事务回滚，只有全部都执行成功，整个事务才算成功。

2、Consistency 一致性

事务的执行不能破坏数据库数据的完整性和一致性，事务在执行之前和之后，数据库都必须处于一致性状态。

3、Isolation 隔离性

在并发环境中，并发的事务是相互隔离的，一个事务的执行不能被其他事务干扰。

即不同的事务并发操纵相同的数据时，每个事务都有各自完整的数据空间，即一个事务内部的操作及使用的数据对其他并发事务是隔离的，并发执行的各个事务之间不能相互干扰。

4、Durability 持久性

持久性（durability）：持久性也称永久性（permanence），指一个事务一旦提交，它对数据库中对应数据的状态变更就应该是永久性的。

即使发生系统崩溃或机器宕机，只要数据库能够重新启动，那么一定能够将其恢复到事务成功结束时的状态。

> 比方说：一个人买东西的时候需要记录在账本上，即使老板忘记了那也有据可查。

以上的事务特性是由Innodb引擎提供，为实现这些特性，使用到了数据库锁、WAL、MVCC等技术。

事务的并发执行

并发场景、高并发场景下， 事务都是并发执行的。

多个事务，如果都是一个一个串行，想想数据库的性能会有多低下。

但是，事务的并发执行，可能会带来很多问题，比如 脏读、不可重复读、幻读等问题。

如果不对事务进行并发控制，我们看看数据库并发操作是会有那些异常情形

- （1）一类丢失更新：两个事物读同一数据，一个修改字段1，一个修改字段2，后提交的恢复了先提交修改的字段。
- （2）二类丢失更新：两个事物读同一数据，都修改同一字段，后提交的覆盖了先提交的修改。
- （3）脏读：读到了未提交的值，万一该事物回滚，则产生脏读。
- （4）不可重复读：两个查询之间，被另外一个事务修改（update）了数据的内容，产生内容的不一致。
- （5）幻读：两个查询之间，被另外一个事务插入或删除了（insert、delete）记录，产生结果集的不一致。

在数据库操作中，为了有效保证并发读取数据的正确性，提出的事务**隔离级别**。我们的数据库锁，也是为了构建这些隔离级别存在的。

隔离级别	**脏读（Dirty Read）**	**不可重复读（NonRepeatable Read）**	**幻读（Phantom Read）**
读未提交(Read uncommitted)	可能	可能	可能
读已提交（Read committed）	不可能	可能	可能
可重复读（Repeatable read）	**不可**能	**不可**能	**可能**
可串行化	不可能	不可能	不可能

隔离级别	脏读（Dirty Read）	不可重复读（NonRepeatable Read）	幻读（Phantom Read）
（Serializable）	不可能	不可能	不可能

（1）读未提交

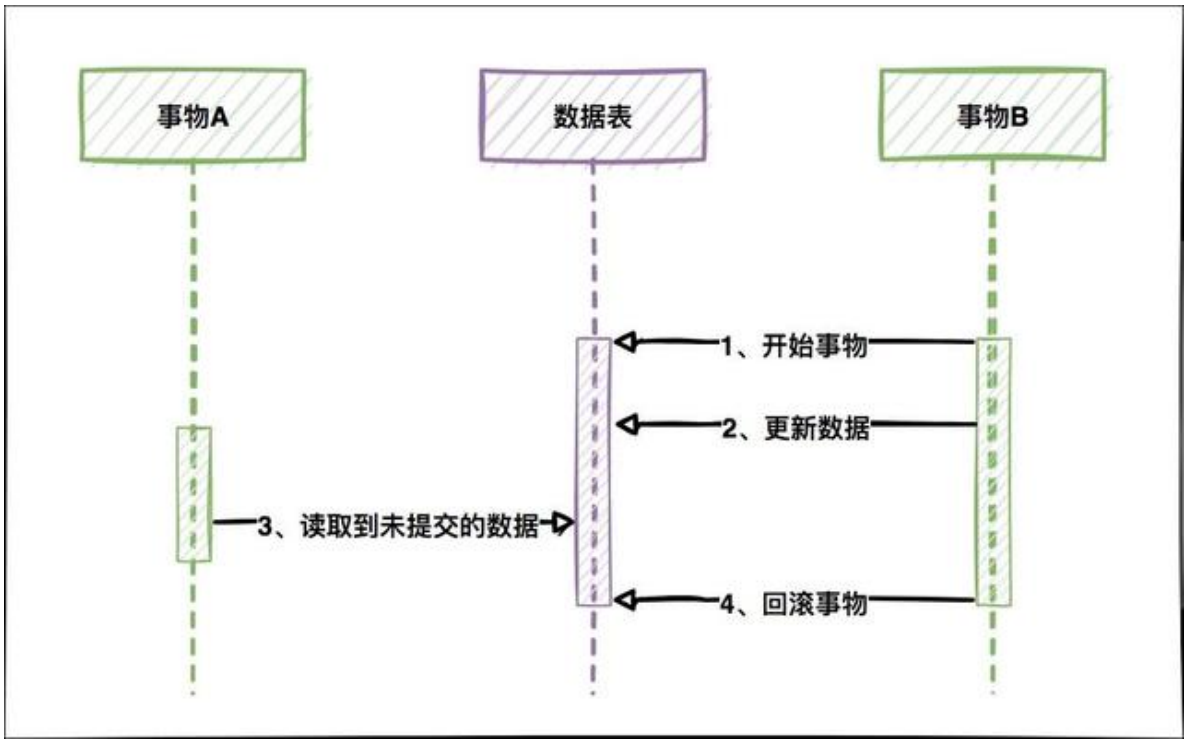
```
如果一个事务正在处理某一数据，并对其进行了更新，
但同时尚未完成事务，或者说事务没有提交，
与此同时，允许另一个事务也能够访问该数据。
例如A将变量n从0累加到10才提交事务，此时B可能读到n变量从0到10之间的所有中间值。
```

允许脏读。在 **读未提交** 隔离级别下，允许 **脏读** 的情况发生。

脏读指的是读到了其他事务未提交的数据，

未提交意味着这些数据可能会回滚，也就是可能最终不会存到数据库中，也就是不存在的数据。

读到了并一定最终存在的数据，这就是脏读。



脏读最大的问题就是可能会读到不存在的数据。

比如在上图中，事务B的更新数据被事务A读取，但是事务B回滚了，更新数据全部还原，也就是说事务A刚刚读到的数据并没有存在于数据库中。

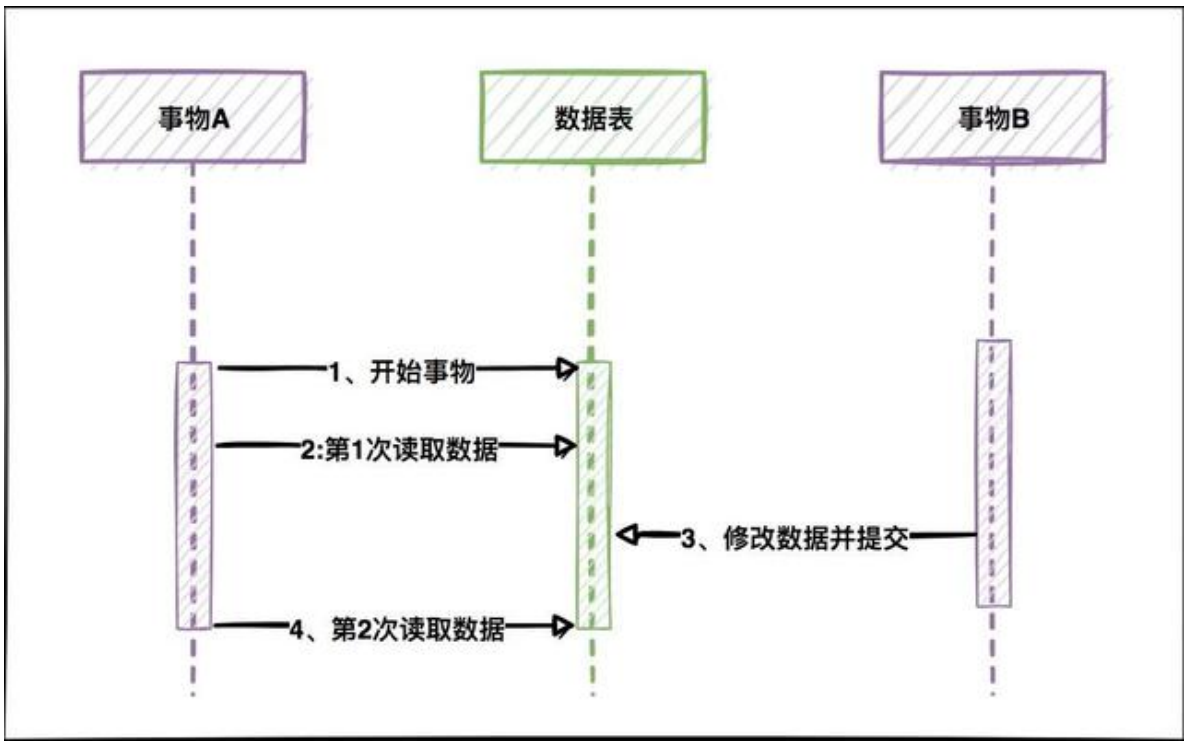
（2）读已提交

```
只允许读到已经提交的数据。
即事务A在将n从0累加到10的过程中，B无法看到n的中间值，之中只能看到10。
```

在 **读已提交** 隔离级别下，**禁止了 脏读**，但是 允许**不可重复读**的情况发生

```
事务A在将n从0累加到10的过程中，B无法看到n的中间值，之中只能看到10。
同时，
有事务C进行从10到20的累加，此时B在同一个事务内再次读时，读到的是20。
```

不可重复读指的是在一个事务内，最开始读到的数据和事务结束前的任意时刻读到的同一批数据出现不一致的情况。



事务 A 多次读取同一数据，但事务 B 在事务A多次读取的过程中，对数据作了更新并提交，导致事务A多次读取同一数据时，结果 不一致。

不可重复读 一词，有点反人类，不好记忆。是从 **Nonrepeatable read** 翻译过来的，感觉英文的，好记忆一点。

（3）可重复读

```
1 保证在事务处理过程中，多次读取同一个数据时，其值都和事务开始时刻时是一致的。
```

在**可重复读**隔离级别下，禁止了：**脏读、不可重复读。**

但是，允许**幻读。**

在可重复读中，该sql第一次读取到数据后，就将这些数据加锁（悲观锁），其它事务无法修改这些数据，就可以实现可重复读了。

但这种方法却无法锁住insert的数据，所以当事务A先前读取了数据，或者修改了全部数据，事务B还是可以insert数据提交，

这时事务A就会发现莫名其妙多了一条之前没有的数据，这就是幻读，不能通过行锁来避免。

（4）串行化

```
1 最严格的事务，要求所有事务被串行执行，不能并发执行。
```

事务的隔离级别总结

隔离级别有串行化读、可重复读、读已提交、读未提交四种级别。

- 串行化读级别下的事务并发度太低，原因是锁的粒度太大，基本没有场景可以被使用。
- 读未提交级别允许脏读，可以使用的场景并不多。
- 读已提交和可重复读是大多数数据库也是大多数项目会采用的数据库隔离级别。

读已提交隔离级别下由于读到其他事务已提交的数据，所以不会出现脏读，在普通读取时，使用到MVCC的快照读机制解决幻读问题；若在查询语句后增加for update，标识当前查询是当前读，当前读并不能解决幻读问题；允许不可重复读。

读已提交隔离级别并发度比较高，互联网行业需要数据库较高的事务并发度，一般会选择此种隔离级别。也是Oracle数据库的默认隔离级别。在使用锁方面，查询时对结果中数据的索引加共享锁，数据读取结束就会释放，更新时对操作的数据索引加排他锁，需要等到事务结束才会释放。

可重复读隔离级别是Mysql的默认隔离级别，**事务并发度仅次于读已提交**，普通读取的幻读问题与读已提交的解决方式一致，不过当前读可以使用 Gap Lock + Record Lock 解决。不可重复读的问题解决办法就是查询时对结果中数据的索引加锁共享锁，再事务结束时才会释放锁。解决不可重读的代价会牺牲掉并发度。

JDBC的转账事务案例

为了深入Spring 事务源码，先看一下在 JDBC 中对事务的操作处理.

还是以经典的 转账 事务为例。

```
1
2  import java.sql.Connection;
3  import java.sql.DriverManager;
4  import java.sql.PreparedStatement;
5  import java.sql.SQLException;
6
7  public class TransactionExample {
8
9      public static void main(String[] args) {
10         Connection conn = null;
11         PreparedStatement pstmt1 = null, pstmt2 = null;
12         try {
13
14             // 加载数据库驱动并建立连接
15             Class.forName("com.mysql.jdbc.Driver");
16             conn =
   DriverManager.getConnection("jdbc:mysql://localhost:3306/test", "root",
   "password");
17
18             // 关闭自动提交，开启事务
19             conn.setAutoCommit(false);
20
21             // 创建SQL语句
22             String sql1 = "UPDATE account SET balance = balance - ? WHERE
   id = ?";
23             String sql2 = "UPDATE account SET balance = balance + ? WHERE
   id = ?";
24
25             // 创建PreparedStatement对象
26             pstmt1 = conn.prepareStatement(sql1);
27             pstmt2 = conn.prepareStatement(sql2);
28
29             // 设置参数
30             pstmt1.setDouble(1, 1000);
31             pstmt1.setInt(2, 1);
32             pstmt2.setDouble(1, 1000);
33             pstmt2.setInt(2, 2);
```

```
            // 执行更新操作
            int count1 = pstmt1.executeUpdate();
            int count2 = pstmt2.executeUpdate();

            if (count1 > 0 && count2 > 0) {
                System.out.println("转账成功");
                // 提交事务
                conn.commit();
            } else {
                System.out.println("转账失败");
                // 回滚事务
                conn.rollback();
            }
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            try {
                if (conn != null) {
                    // 回滚事务
                    conn.rollback();
                }
            } catch (SQLException e1) {
                e1.printStackTrace();
            }
            e.printStackTrace();
        } finally {
            try {
                if (pstmt1 != null) {
                    pstmt1.close();
                }
                if (pstmt2 != null) {
                    pstmt2.close();
                }
                if (conn != null) {
                    conn.close();
                }
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

java 代码中要使用事务，就是 三个步骤：

```
            start transaction;-- 开启一个事务
            commit;-- 事务提交
            rollback;-- 事务回滚

```

mysql中事务默认是自动提交,一条sql语句就是一个事务.

所以，上面的代码，第一步，首先关闭自动提交，开启手动事务方式

```
// 关闭自动提交，开启事务
conn.setAutoCommit(false);


```

然后，第2步，如果顺利的话，就提交事务

```
 // 提交事务
  conn.commit();

```

如果发生异常的话，第3步，就回滚事务

```
  // 回滚事务
   conn.rollback();

```

这里总结一下，上面代码中的JDBC的转账事务案例几个重点步骤：

- 获取连接：

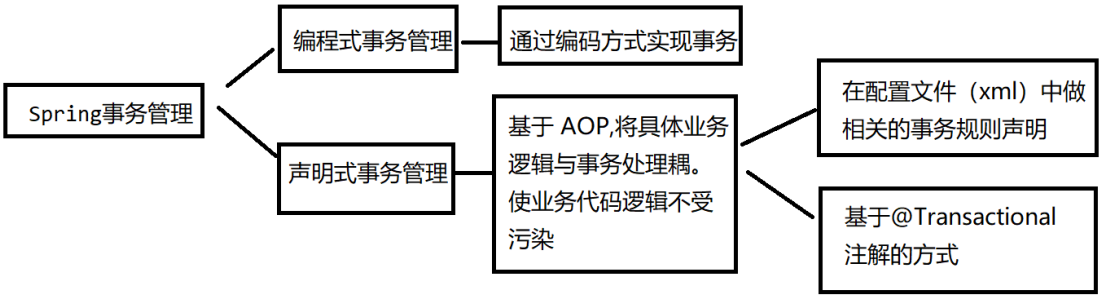
  ```
  Connection conn =
  DriverManager.getConnection("jdbc:mysql://localhost:3306/test", "root",
  "password")

  ```

- 关闭自动提交，开启事务：conn.setAutoCommit(false)
- 提交事务：conn.commit()
- 回滚事务：conn.rollback()

Spring 转账事务案例

Spring 两种事务管理方式

Spring 支持两种事务管理方式：编程式事务和声明式事务。



编程式事务是指在代码中显式地开启、提交或回滚事务。这种方式需要在代码中编写事务管理的相关逻辑，比较繁琐，但是灵活性较高，可以根据具体的业务需要进行定制。

声明式事务是通过配置来实现的，不需要在代码中显式地管理事务。这种方式需要在配置文件中声明事务的属性，比如事务的传播行为、隔离级别等。声明式事务的好处是可以将事务管理的逻辑与业务逻辑分离，使得代码更加简洁、清晰，同时也方便了事务管理的统一配置和维护。

在 Spring 中，声明式事务主要是通过 AOP 实现的。Spring 提供了两种声明式事务的方式：基于 XML 配置和基于注解配置。基于 XML 配置的方式需要在 Spring 配置文件中声明事务管理器和事务通知等相关信息，而基于注解配置的方式则可以在代码中通过注解来声明事务的属性，比如 @Transactional。

基于 XML 配置文件进行配置

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    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
        http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-4.3.xsd
        http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-4.1.xsd">
    <!-- 开启扫描 -->
    <context:component-scan base-package="com.dpb.*"></context:component-
scan>

    <!-- 配置数据源 -->
    <bean
class="org.springframework.jdbc.datasource.DriverManagerDataSource"
id="dataSource">
        <property name="url"
value="jdbc:oracle:thin:@localhost:1521:orcl"/>
        <property name="driverClassName"
value="oracle.jdbc.driver.OracleDriver"/>
        <property name="username" value="pms"/>
        <property name="password" value="pms"/>
    </bean>

    <!-- 配置JdbcTemplate -->
    <bean class="org.springframework.jdbc.core.JdbcTemplate" >
```

```
        <constructor-arg name="dataSource" ref="dataSource"/>
    </bean>

    <!--
    Spring中，使用XML配置事务三大步骤：
        1. 创建事务管理器
        2. 配置事务方法
        3. 配置AOP
     -->
     <bean
class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
id="transactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>
    <tx:advice id="advice" transaction-manager="transactionManager">
        <tx:attributes>
            <tx:method name="fun*" propagation="REQUIRED"/>
        </tx:attributes>
    </tx:advice>
    <!-- aop配置 -->
    <aop:config>
        <aop:pointcut expression="execution(* *..service.*.*(..))"
id="tx"/>
        <aop:advisor advice-ref="advice" pointcut-ref="tx"/>
    </aop:config>
</beans>

```

基于注解的声明式配置

一般来说，更加推荐声明式事务比编程式事务，因为它可以使代码更加简洁、清晰，同时也方便了事务管理的统一配置和维护。

所以，这里使用 声明式事务 进行演示，并且是使用 基于注解配置的 声明式事务。

首先必须要添加 @EnableTransactionManagement 注解，保证事务注解生效

```
@EnableTransactionManagement
public class AnnotationMain {
    public static void main(String[] args) {
    }
}

```

其次，在方法上添加 @Transactional 代表注解生效

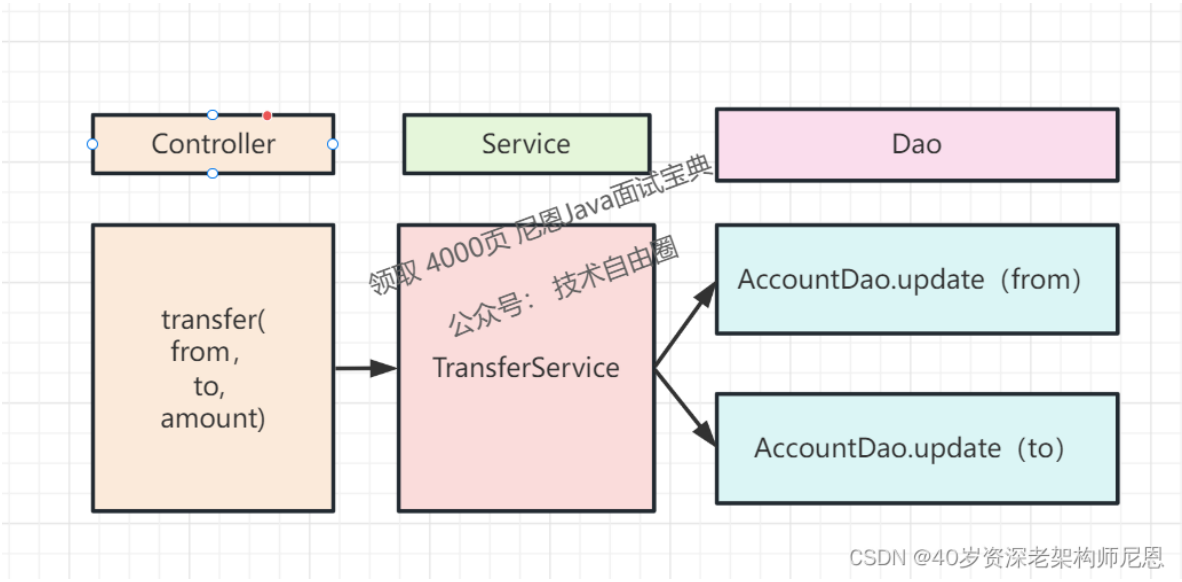
```
@Transactional
public int insertUser(User user) {
    userDao.insertUser();
    userDao.insertLog();
    return 1;
}

```

下面的案例，用到基于注解的声明式配置，具体的注解是 @Transactional。

转账案例的三层架构

首先，看看日常项目中，转账案例的三层架构。

我们在日常生产项目中，项目由 Controller、Serivce、Dao 三层进行构建。



对于服务层的transfer 方法，实际对于DAO 调用来说，分别调用了两次 DAO的upate 方法，更新了两个账号的amount金额，也就是说，对数据库的操作为两次。

所以，我们要保证 transfer 方法是符合事务定义的，具备事务的四大特性：ACID。

一个简单的 Spring 转账事务案例

好的，下面是一个简单的 Spring 转账事务案例， 一共5步骤：

- 1. 定义账户实体类
- 2. 定义转账服务接口
- 3. 实现转账服务接口
- 4. 定义账户数据访问对象（DAO）
- 5. 配置事务管理器

接下来，咱们step by step，一点点揭秘一下这个5步骤

1. 定义账户实体类

```
public class Account {

    private Long id;
    private String accountNumber;
    private double balance;

    // 省略 getter 和 setter 方法
}

```

2. 定义转账服务接口

```java
public interface TransferService {
    void transfer(String fromAccount, String toAccount, double amount);
}
```

3. 实现转账服务接口

```java
@Service
public class TransferServiceImpl implements TransferService {

    @Autowired
    private AccountDao accountDao;

    @Transactional
    public void transfer(String fromAccount, String toAccount, double amount) {
        Account from = accountDao.findByAccountNumber(fromAccount);
        Account to = accountDao.findByAccountNumber(toAccount);
        from.setBalance(from.getBalance() - amount);
        to.setBalance(to.getBalance() + amount);
        accountDao.update(from);
        accountDao.update(to);
    }
}
```

4. 实现账户数据访问对象（DAO）

```java
@Repository
public class AccountDaoImpl implements AccountDao {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    public Account findByAccountNumber(String accountNumber) {
        String sql = "SELECT * FROM account WHERE account_number = ?";
        return jdbcTemplate.queryForObject(sql, new Object[]{accountNumber}, new AccountRowMapper());
    }

    public void update(Account account) {
        String sql = "UPDATE account SET balance = ? WHERE id = ?";
        jdbcTemplate.update(sql, account.getBalance(), account.getId());
    }
}

class AccountRowMapper implements RowMapper<Account> {
    public Account mapRow(ResultSet rs, int rowNum) throws SQLException {
        Account account = new Account();
        account.setId(rs.getLong("id"));
        account.setAccountNumber(rs.getString("account_number"));
        account.setBalance(rs.getDouble("balance"));
        return account;
    }
}

```

5. 配置事务管理器

```
@Configuration
@EnableTransactionManagement
public class AppConfig {

    @Bean
    public DataSource dataSource() {
        // 配置数据源
    }

    @Bean
    public JdbcTemplate jdbcTemplate() {
        return new JdbcTemplate(dataSource());
    }

    @Bean
    public PlatformTransactionManager transactionManager() {
        return new DataSourceTransactionManager(dataSource());
    }
}
```

在上面的代码中，我们使用了 Spring 的声明式事务管理。

在 第3步，也就是服务层的实现方法上，添加 @Transactional 注解，告诉 Spring 这是一个需要进行事务管理的方法。

这里的事务方法，是transfer() 方法。

当 transfer() 方法被调用时，如果发生异常，事务管理器会自动回滚事务，保证数据库的一致性。

详解：事务注解 @Transactional

@Transactional 是 Spring Framework 中常用的注解之一，它可以被用于管理事务。通过使用这个注解，我们可以方便地管理事务，保证数据的一致性和完整性。

在 Spring 应用中，当我们需要对数据库进行操作时，通常需要使用事务来保证数据的一致性和完整性。

@Transactional 注解可以被用于类或方法上，用于指定事务的管理方式。当它被用于类上时，它表示该类中所有的方法都将被包含在同一个事务中。当它被用于方法上时，它表示该方法将被包含在一个新的事务中。

@Transactional 注解有多个属性，其中最常用的是 propagation 和 isolation。

- propagation 属性用于指定事务的传播行为，它决定了当前方法执行时，如何处理已经存在的事务
- isolation 属性用于指定事务的隔离级别，它决定了当前事务与其他事务之间的隔离程度。

除了 propagation 和 isolation 属性外，@Transactional 还支持其他属性，如 readOnly、timeout、rollbackFor、noRollbackFor 等，这些属性可以用于进一步细化事务的行为。

总之，@Transactional 注解是 Spring 应用中常用的事务管理注解。

@Tranasctional注解的使用注意事项

@Tranasctional注解是Spring 框架提供的声明式注解事务解决方案，在使用@Transactional注解时需要注意以下问题:

1. @Transactional 注解只能用在public 方法上，如果用在protected或者private的方法上，不会报错，但是该注解不会生效。
2. 默认情况下，@Transactional注解只能回滚 非检查型异常，具体为RuntimeException及其子类和Error子类

> 非检查型异常指（Unchecked Exception）的是程序在编译时不会提示需要处理该异常，而是在运行时才会出现异常。在 Java 中，非检查型异常指的是继承自 RuntimeException 类的异常，例如 NullPointerException、ArrayIndexOutOfBoundsException 等。这些异常通常是由程序员的代码错误引起的，因此应该尽可能避免它们的发生，但是在代码中并不需要显式地处理它们。
>
> 检查型异常（Checked Exception）是指在 Java 中，编译器会强制要求对可能会抛出这些异常的代码进行异常处理，否则代码将无法通过编译。这些异常包括 IOException、SQLException 等等，它们通常表示一些外部因素导致的异常情况，比如文件读写错误、数据库连接失败等等。
>
> 在编写代码时应该尽量避免抛出非检查型异常，因为这些异常的发生通常意味着程序存在严重的逻辑问题。

默认情况下，@Transactional注解只能回滚 非检查型异常，为啥呢?

可以从Spring源码的DefaultTransactionAttribute类里找到判断方法rollbackOn。

```
@Override
public boolean rollbackOn(Throwable ex) {
    return (ex instanceof RuntimeException || ex instanceof Error);
}
```

3. 如果需要对检查型异常（Checked Exception）进行回滚，可以使用rollbackFor 属性来定义回滚的异常类型，使用 propagation 属性定义事务的传播行为。

下面是一个例子:

```
 @Transactional(rollbackFor = Exception.class,propagation =
Propagation.REQUIRED)
```

上面的例子中: 指定了回滚Exception类的异常为 Exception 类型或者其子类型检查型异常（Checked Exception），另外，配置类事务的传播行为支持当前事务，当前如果没有事务，那么会创建一个事务。

4. @Transactional注解不能回滚被try{}catch() 捕获的异常。
5. @Transactional注解只能对在被Spring 容器扫描到的类下的方法生效。

其实Spring事务的创建也是有一定的规则，对于一个方法里已经存在的事务，Spring 也提供了解决方案去进一步处理存在事务，通过设置@Tranasctional的propagation 属性定义Spring 事务的传播规则。

Spring事务的传播规则

Spring 事务的传播规则是指在多个事务方法相互调用的情况下，事务应该如何进行传播和管理。

Spring事务的传播行为一共有7种，定义在spring-tx模块的Propagation枚举类里，对应的常量值定义在TransactionDefinition接口里,值为int类型的0-6。

PROPAGATION_REQUIRED	**支持当前事务，如果当前没有事务，则创建一个事务，这是最常见的选择。**
PROPAGATION_SUPPORTS	支持当前事务，如果当前没有事务，就以非事务来执行
PROPAGATION_MANDATORY	支持当前事务，如果没有当前事务，就抛出异常。
PROPAGATION_REQUIRES_NEW	新建事务，如果当前存在事务，就把当前事务挂起。
PROPAGATION_NOT_SUPPORTED	以非事务执行操作，如果当前存在事务，则当前事务挂起。
PROPAGATION_NEVER	以非事务方式执行，如果当前存在事务，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，则进行与PROPAGATION_REQUIRED 类似的操作。

稍后一点，结合源码介绍。

Spring事务总体架构

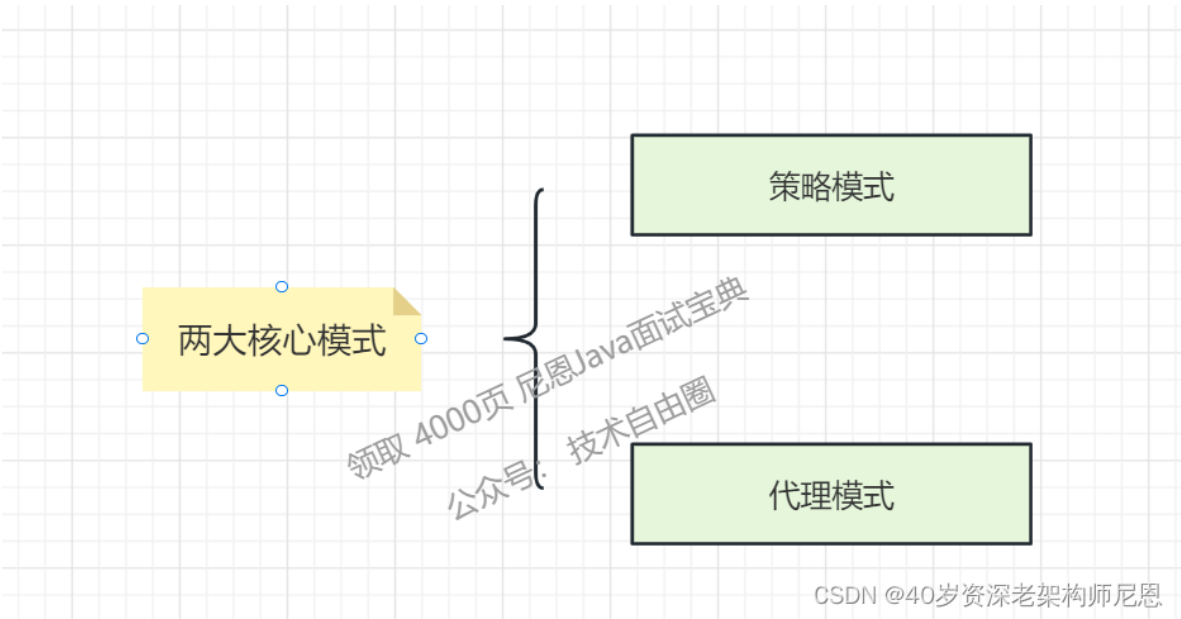
40岁老架构尼恩的建议： 要学源码，先梳理架构。

pring事务总体架构，尼恩给大家梳理了两个要点：

- 二大核心模式
- 三大核心流程

二大核心模式

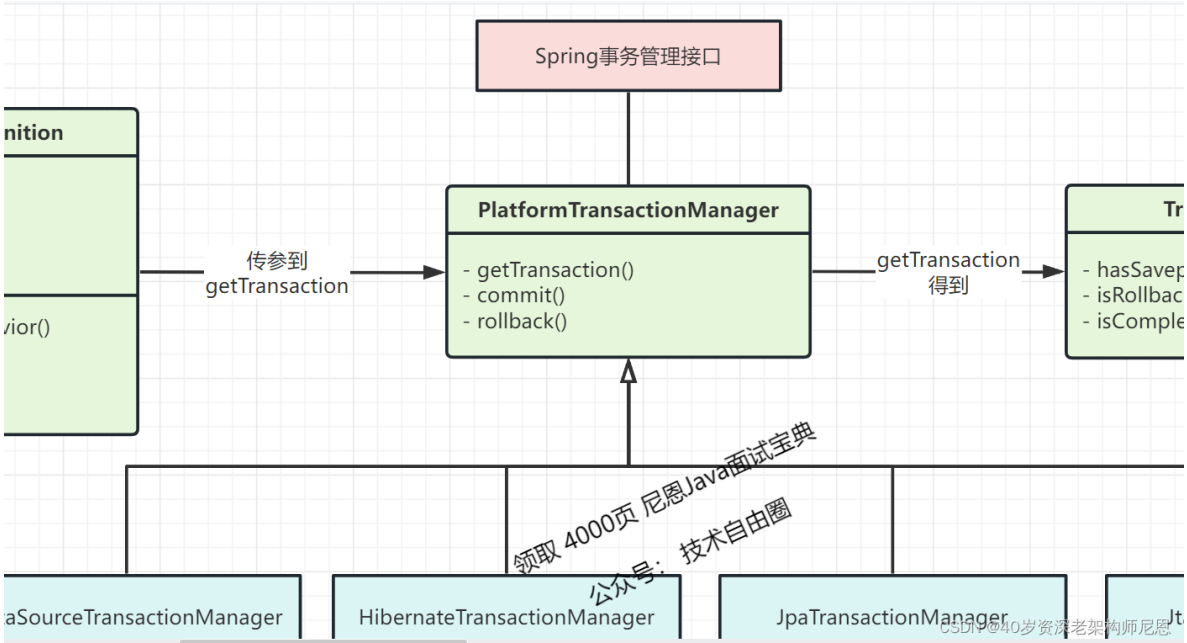
Spring事务架构， 用到了二大核心模式



模式一： 策略模式

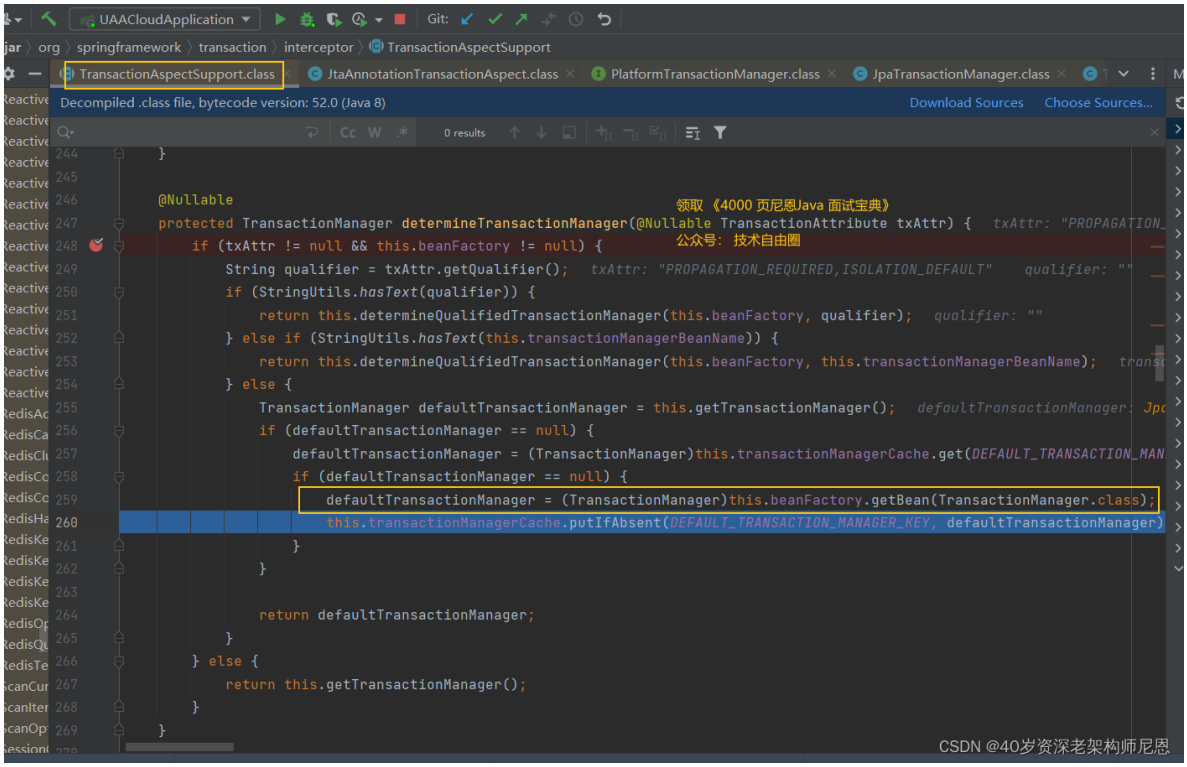
设计了统一的事务管理器超级接口， **PlatformTransactionManager** 是 Spring事务管理器设计的 基类。

PlatformTransactionManager 基类，实现了 事务执行设计到的三个核心方法：



不同的 数据管理组件JDBC、Hibernate等，为Spring都提供了对应的事务管理器 实现类。

既然是策略模式，如何进行动态的决策呢?



在动态代理模式的 切面的支撑 类 TransactionAspectSupport 中， 会有一个 determineTransactionManager 方法， 动态的从IOC容器，加载 注册了的 **PlatformTransactionManager** 实现类。

尼恩的脚手架，用到的orm框架是 jpa。在spring的bean factory里边 注册了的 **PlatformTransactionManager** 实现类 是 JpaTransactionManager 。

所以，这里的 JpaTransactionManager 作为事务管理器。

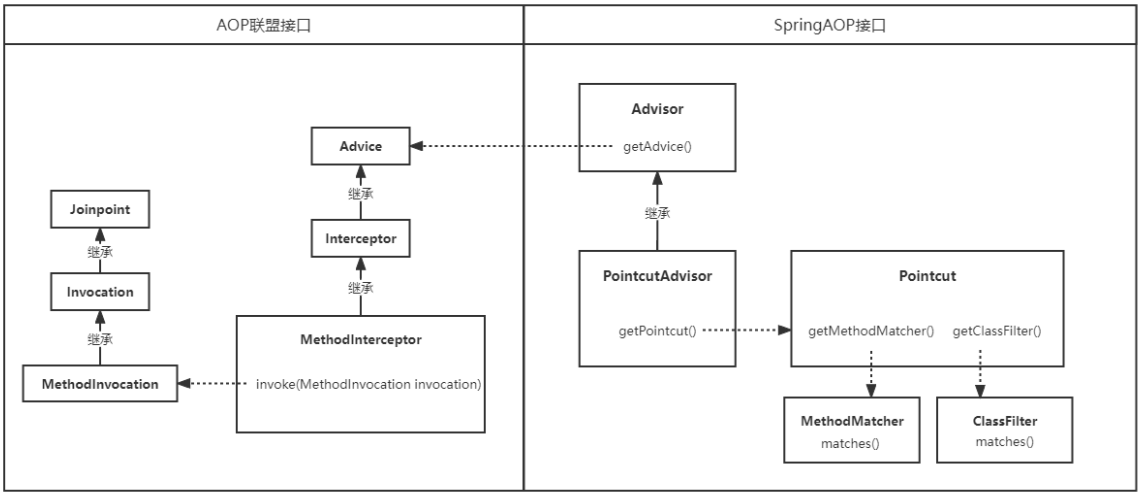
模式二： 动态代理模式

动态代理有JDK的动态代理、 Aspectj的动态代理，SpringAOP动态代理 ，很多种。

SpringAOP 和 Aspectj的动态代理 有和联系呢？ 本质上 Aspectj是一种静态代理，而SpringAOP是动态代理。但Aspectj的一套定义AOP的API非常好，直观易用。

但是，AOP联盟的关键词有Advice(顶级的通知类/拦截器)、MethodInvocation(方法连接点)、MethodInterceptor(方法拦截器)

SpringAOP与AOP联盟关系



SpringAOP在AOP联盟基础上又增加了几个类，丰富了AOP定义及使用概念，SpringAOP 的核心名称和概念包括：

- **Advisor**：包含通知(拦截器)，Spring内部使用的AOP顶级接口，还需要包含一个aop适用判断的过滤器，考虑到通用性，**过滤规则由其子接口定义，例如IntroductionAdvisor和PointcutAdvisor**，过滤器用于判断bean是否需要被代理
- **Pointcut**: 切点，属于过滤器的一种实现，匹配过滤哪些类哪些方法需要被切面处理，包含一个ClassFilter和一个MethodMatcher，使用PointcutAdvisor定义时需要
- **ClassFilter**：限制切入点或引入点与给定目标类集的匹配的筛选器，属于过滤器的一种实现。过滤筛选合适的类，有些类不需要被处理
- **MethodMatcher**：方法匹配器，定义方法匹配规则，属于过滤器的一种实现，哪些方法需要使用AOP

SpringAOP实现的大致思路：

1.配置获取Advisor (顾问)：拦截器+AOP匹配过滤器，生成Advisor
2.生成代理：根据Advisor生成代理对象，会生成JdkDynamicAopProxy或CglibAopProxy
3.执行代理：代理类执行代理时，从Advisor取出拦截器，生成MethodInvocation(连接点)并执行代理过程。

如果对 在 Spring AOP 中，Advisor 和 Interceptor 是两个重要的概念。

Advisor 是用于定义切面的对象，它包含了切点和通知两个部分。

- 切点指定了哪些方法需要被拦截， 使用@Pointcut 注解
- 通知则指定了拦截后需要执行的逻辑。

Interceptor 是用于拦截后，执行拦截逻辑的对象，它实现了 Spring 的 MethodInterceptor 接口，负责拦截方法的执行，并在方法执行前后执行一些操作。

下面是一个使用 Advisor 和 Interceptor 的示例：

```
@Aspect
@Component
public class MyInterceptor {

    //切点
    @Pointcut("execution(* com.example.service.*.*(..))")
    public void pointcut() {}

    @Around("pointcut()")
    public Object around(ProceedingJoinPoint pjp) throws Throwable {
        // 在方法执行前执行的逻辑
        System.out.println("before method");

        // 执行被拦截的方法
        Object result = pjp.proceed();

        // 在方法执行后执行的逻辑
        System.out.println("after method");

        return result;
    }
}
```

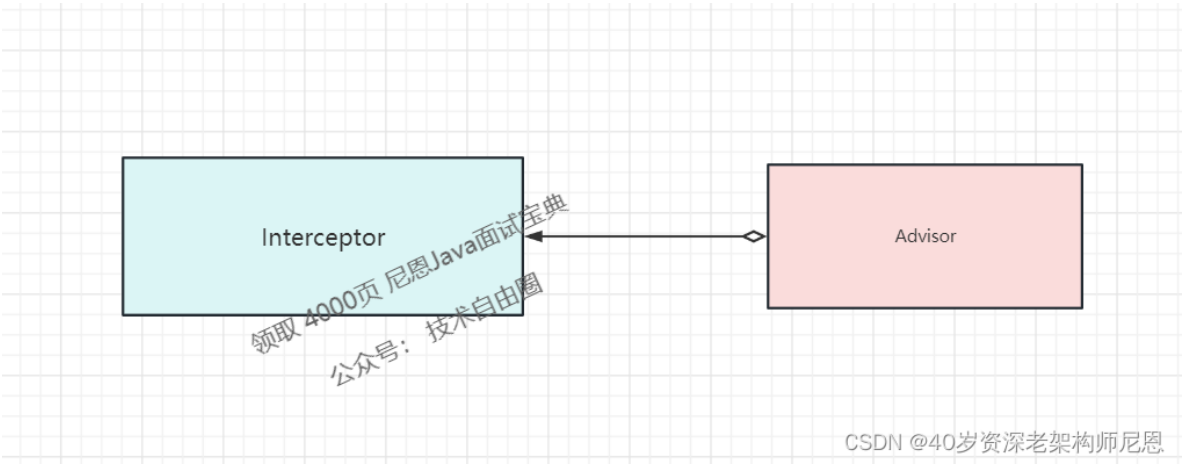
在上面的代码中，我们使用了 Advisor 和 Interceptor 来实现对 com.example.service 包中所有方法的拦截。

首先，我们定义了一个切点 pointcut()，它指定了需要拦截的方法。

然后，我们定义了一个 around() 方法，并使用 @Around 注解将它与切点关联起来。在 around() 方法中，我们实现了拦截逻辑，即在方法执行前后分别输出 "before method" 和 "after method"。最后，我们通过 pjp.proceed() 方法执行了被拦截的方法，并返回了它的执行结果。

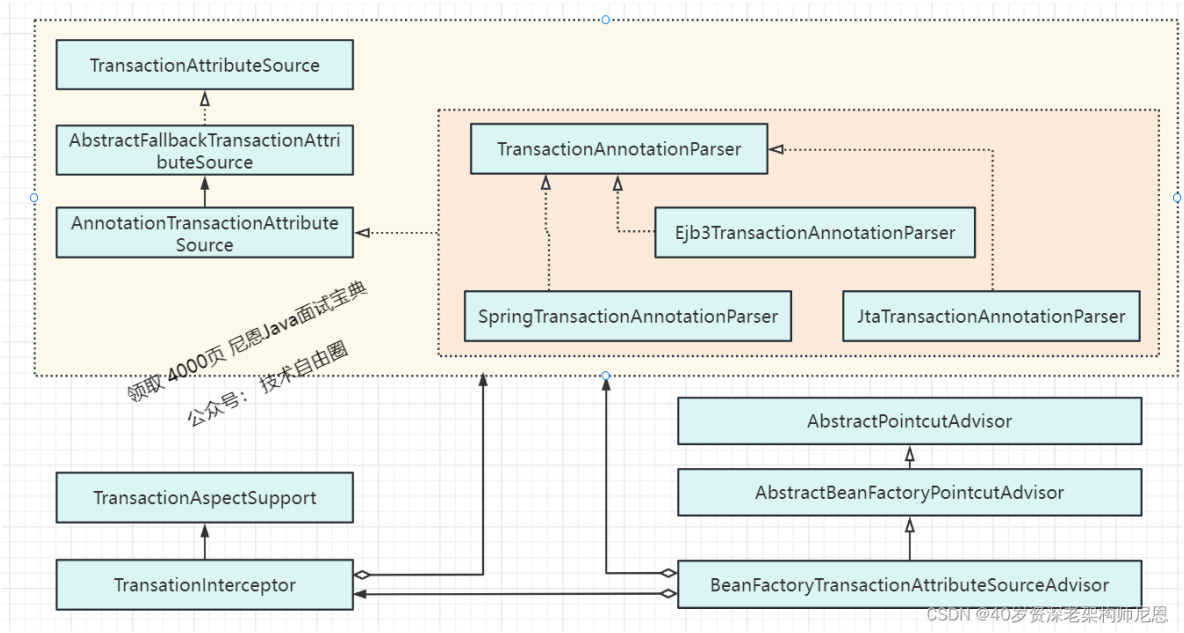
需要注意的是，为了让 Spring 能够识别 MyInterceptor 类并将它作为 Advisor 使用，我们需要在它上面加上 @Aspect 和 @Component 注解。

注意 Advisor 是动态生成的，但是Spring也会实现了一个类似于 `PointcutAdvisor` 的接口，可以根据配置的切入点和目标方法生成代理对象，调用咱们定义的MyInterceptor 实现拦截和AOP增强功能。



了解了SpringAOP 基本原理之后，我们来看看SpringAOP实现 事务的 动态代理模式的架构。

SpringAOP实现 事务的 动态代理模式的架构图如下：



动态代理模式的两个核心角色是：

- Advisor： `BeanFactoryTransactionAttributeSourceAdvisor`
- Interceptor： `TransactionInterceptor`

Spring事务进行了Advisor 的扩展， `BeanFactoryTransactionAttributeSourceAdvisor` 是 Spring AOP框架中一个用于事务管理的类。

该 Advisor它实现了 `PointcutAdvisor` 接口，可以根据配置的切入点和事务属性来为目标方法生成代理对象，从而实现事务管理的功能。

在 Spring 中，事务管理是通过 `TransactionInterceptor` 实现的。该 Advisor 的作用就是为 `Interceptor` 提供事务属性信息。

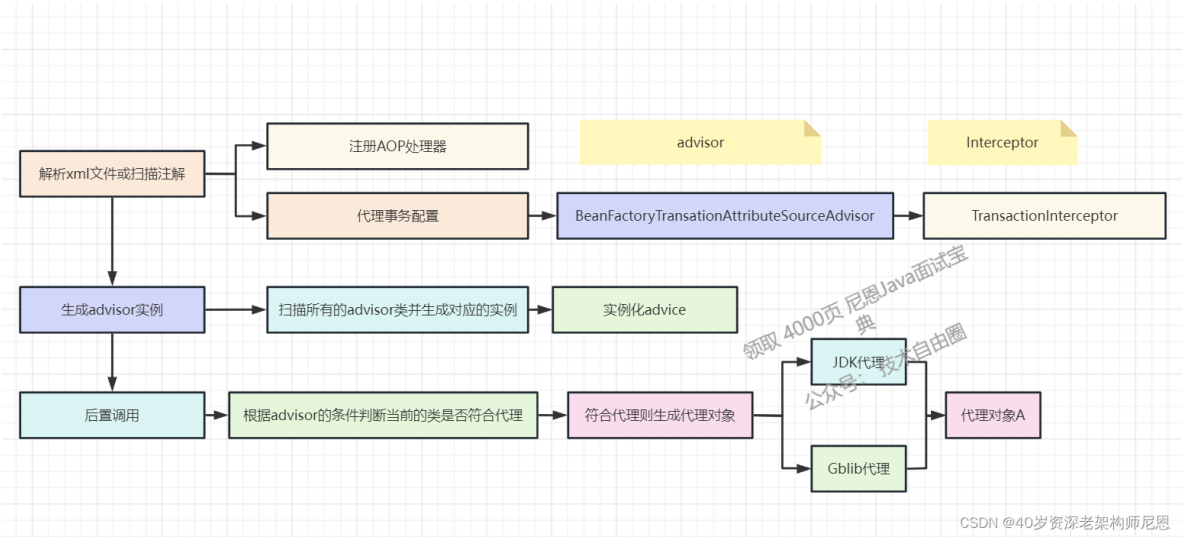
该 Advisor 通过 `TransactionAttributeSource` 接口来获取事务属性，而 `TransactionAttributeSource` 可以从配置文件、注解等多种途径获取事务属性信息。该 Advisor 是从 Spring 容器中获取 `TransactionAttributeSource` 的实现类，并将其设置给 `TransactionInterceptor`，从而实现事务管理的功能。

三大核心流程

- AOP 动态代理装配流程
- AOP 动态代理执行流程
- 事务执行流程

核心流程一：AOP 动态代理装配流程

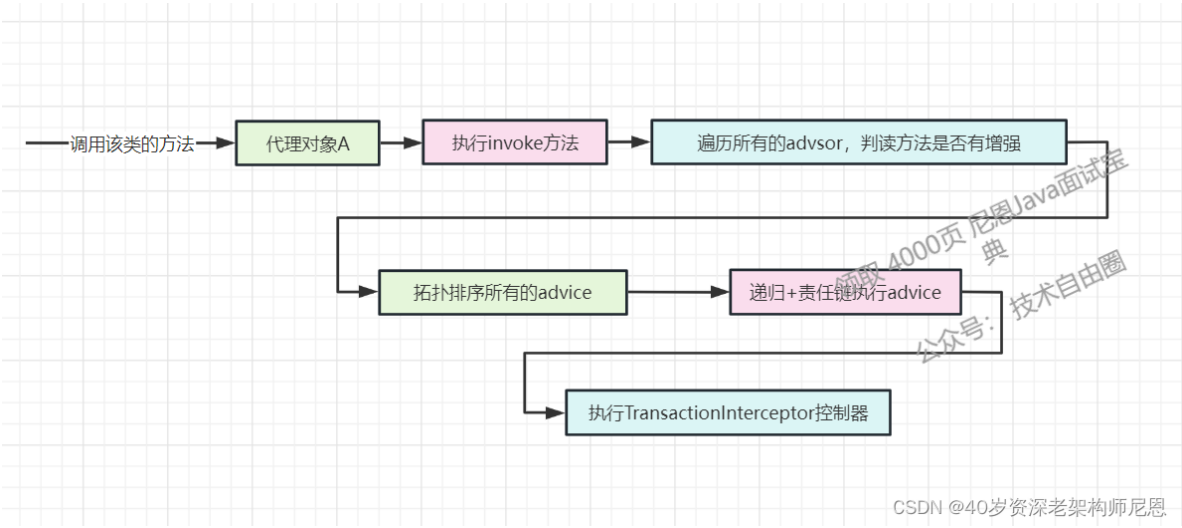
AOP 动态代理装配流程， 大致如下：



咱们关注的是 Advisor 和 Interceptor的装配。

核心流程二：AOP 动态代理执行流程

AOP 动态代理执行流程， 大致如下：

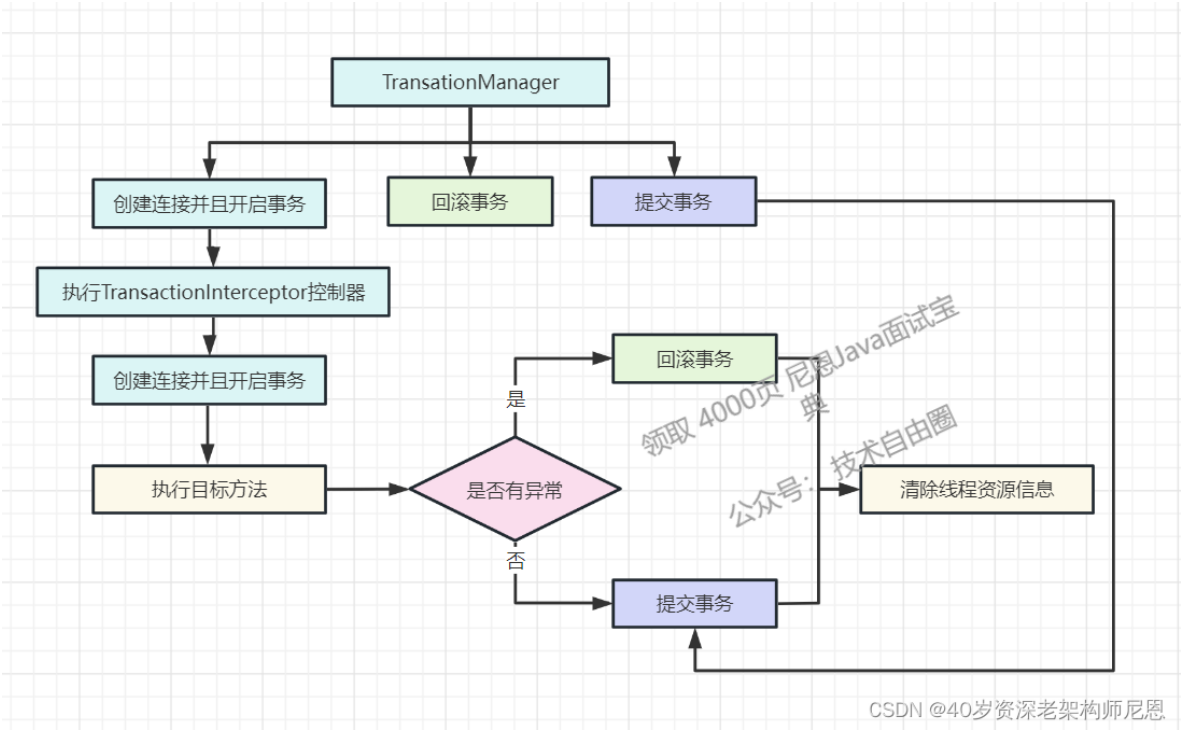


执行被@Transactional注解的方法时，首先会被AOP切面代理拦截， 先执行 事务的动态代理，再执行业务方法。

在事务的动态代理动态代理对象中，进行事务的 开启、提交、或者回滚。

核心流程三：事务执行流程

AOP 事务执行流程，大致如下：



这个使用 Spring事务管理器实现。

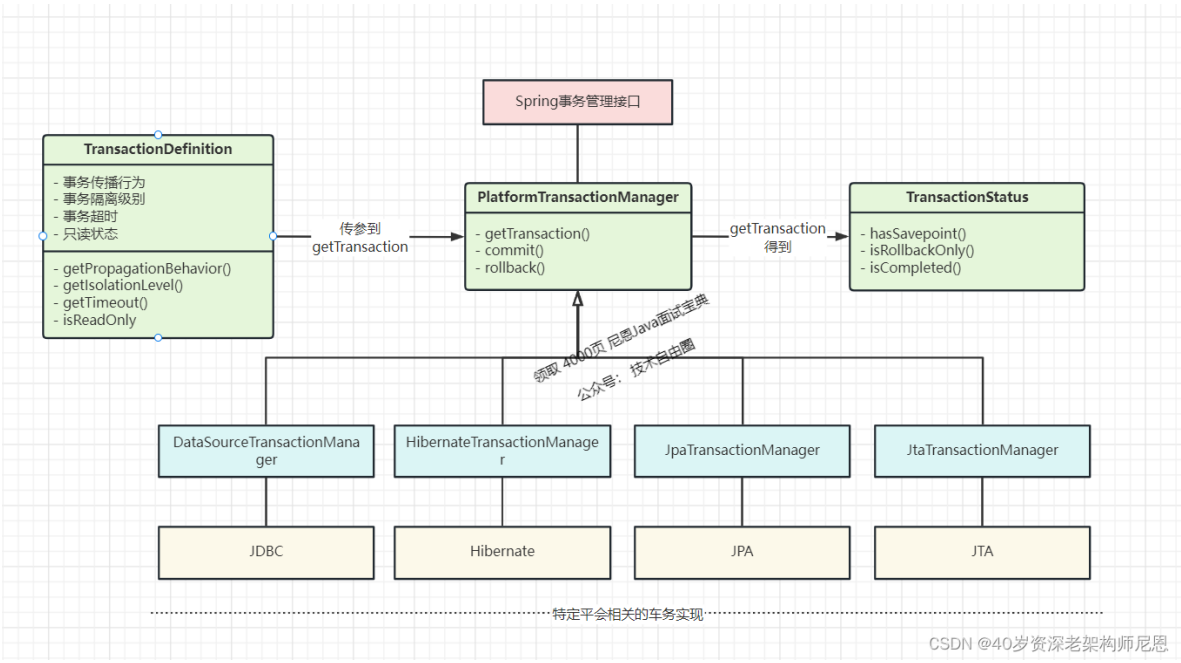
所以，学习Spring源码，咱们先从Spring事务管理器源码开始。

Spring事务管理器架构和源码学习

Spring事务管理的实现有许多细节，如果对整个模块架构有个大体了解，会非常有利于我们理解事务，

Spring事务管理模块架构：

Spring事务管理模块架构，如下：



Spring并不直接管理事务，而是提供了多种事务管理器。

通过事务管理器，Spring将事务管理的职责委托给ORM框架，比如Hibernate或者JTA等，由ORM框架的持久化机制所提供的相关平台框架的事务来实现。

事务管理的超级接口

Spring事务管理器的超级接口是 PlatformTransactionManager，

PlatformTransactionManager超级接口的内容如下：

```
Public interface PlatformTransactionManager()...{
    // 由TransactionDefinition得到TransactionStatus对象
    TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException;
    // 提交
    Void commit(TransactionStatus status) throws TransactionException;
    // 回滚
    Void rollback(TransactionStatus status) throws TransactionException;
    }

```

通过这个超级接口，Spring为各个平台如JDBC、Hibernate等都提供了对应的事务管理器。

从这里可知：PlatformTransactionManager 具体的实现就是各个平台自己的事情了。

所以说，具体的事务管理机制对Spring来说是透明的，Spring并不关心那些，那些是对应各个平台需要关心的。

所以，Spring事务管理模块架构 的一个优点就是：为不同的事务API提供一致的编程模型，如JTA、JDBC、Hibernate、JPA。

下面分别介绍各个平台框架实现事务管理的机制。

JDBC 事务管理器

如果应用程序中直接使用JDBC来进行ORM持久化，DataSourceTransactionManager会为你处理事务，提供事务管理的的具体实现。

`DataSourceTransactionManager` 是 Spring Framework 中用于管理数据库事务的类，它是基于 JDBC 的事务管理器。

为了使用DataSourceTransactionManager，如果使用XML定义Bean，需要使用如下的XML将其装配到应用程序的上下文定义中：

```
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
</bean>

```

在 Spring Boot 中，如果你使用了 Spring Data JPA 或者 MyBatis 等 ORM 框架，那么默认会使用 `DataSourceTransactionManager` 进行事务管理。

SpringBoot在使用事物Transactional的时候，要在main方法上加上 `@EnableTransactionManagement` 注解开发事物声明，在使用的service层的公共方法加上 `@Transactional` (spring)注解。

如果我们不想使用事物 `@Transactional` 注解，想自己进行事物控制(编程事物管理)，控制某一段的代码事务生效或者回滚，但是又不想自己去编写那么多的代码，怎么办呢?

可以使用springboot中的`DataSourceTransactionManager`和`TransactionDefinition`这两个类来结合使用，能够达到手动控制事物的提交回滚。

代码示例:

```
    @Autowired
    private DataSourceTransactionManager dataSourceTransactionManager;
    @Autowired
    private TransactionDefinition transactionDefinition;

    public boolean transferMoney(User user) {
        /*
         * 手动进行事物控制
         */
        TransactionStatus transactionStatus=null;
        boolean isCommit = false;
        try {
            //开启事务
            transactionStatus =

 dataSourceTransactionManager.getTransaction(transactionDefinition);
            System.out.println("查询的数据1:" + udao.findById(user.getId()));
            // 进行新增/修改
            udao.insert(user);
            System.out.println("查询的数据2:" + udao.findById(user.getId()));
            if(user.getAge()<20) {
                user.setAge(user.getAge()+2);
                udao.update(user);
                System.out.println("查询的数据3:" +
udao.findById(user.getId()));
            }else {
                throw new Exception("模拟一个异常!");
            }

            //手动提交
            dataSourceTransactionManager.commit(transactionStatus);

            isCommit= true;
            System.out.println("手动提交事物成功!");
            throw new Exception("模拟第二个异常!");

        } catch (Exception e) {
            //如果未提交就进行回滚
            if(!isCommit){
                System.out.println("发生异常,进行手动回滚! ");
                //手动回滚事物
                dataSourceTransactionManager.rollback(transactionStatus);
            }
            e.printStackTrace();
        }
        return false;
    }
}
```

特别说明：

> 在进行使用的时候，需要注意在回滚的时候，要确保开启了事物但是未提交，如果未开启或已提交的时候进行回滚是会在catch里面发生异常的！

实际上，DataSourceTransactionManager是通过调用java.sql.Connection来管理事务，而后者是通过DataSource获取到的。通过调用连接的commit()方法来提交事务，同样，事务失败则通过调用rollback()方法进行回滚。

Hibernate事务管理器

如果应用程序的持久化是通过Hibernate实现的，那么你需要使用HibernateTransactionManager。对于Hibernate3，需要在Spring上下文定义中添加如下的`<bean>`声明：

```
<bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

sessionFactory属性需要装配一个Hibernate的session工厂，HibernateTransactionManager的实现细节是它将事务管理的职责委托给org.hibernate.Transaction对象，而后者是从Hibernate Session中获取到的。

当事务成功完成时，HibernateTransactionManager将会调用Transaction对象的commit()方法，反之，将会调用rollback()方法。

Java持久化API（JPA）事务管理器

Hibernate多年来一直是事实上的Java持久化标准，但是现在Java持久化API作为真正的Java持久化标准进入大家的视野。

如果你计划使用JPA的话，那你需要使用Spring的JpaTransactionManager来处理事务。

你需要在Spring中这样配置JpaTransactionManager：

```
<bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

JpaTransactionManager只需要装配一个JPA实体管理工厂（javax.persistence.EntityManagerFactory接口的任意实现）。

JpaTransactionManager将与由工厂所产生的JPA EntityManager合作来构建事务。

Java原生API事务管理器

如果你没有使用以上所述的事务管理，或者是跨越了多个事务管理源（比如两个或者是多个不同的数据源），你就需要使用JtaTransactionManager：

```
<bean id="transactionManager"
class="org.springframework.transaction.jta.JtaTransactionManager">
        <property name="transactionManagerName"
value="java:/TransactionManager" />
</bean>
```

JtaTransactionManager将事务管理的责任委托给javax.transaction.UserTransaction和javax.transaction.TransactionManager对象，其中事务成功完成通过UserTransaction.commit()方法提交，事务失败通过UserTransaction.rollback()方法回滚。

TransactionDefinition基本事务属性定义

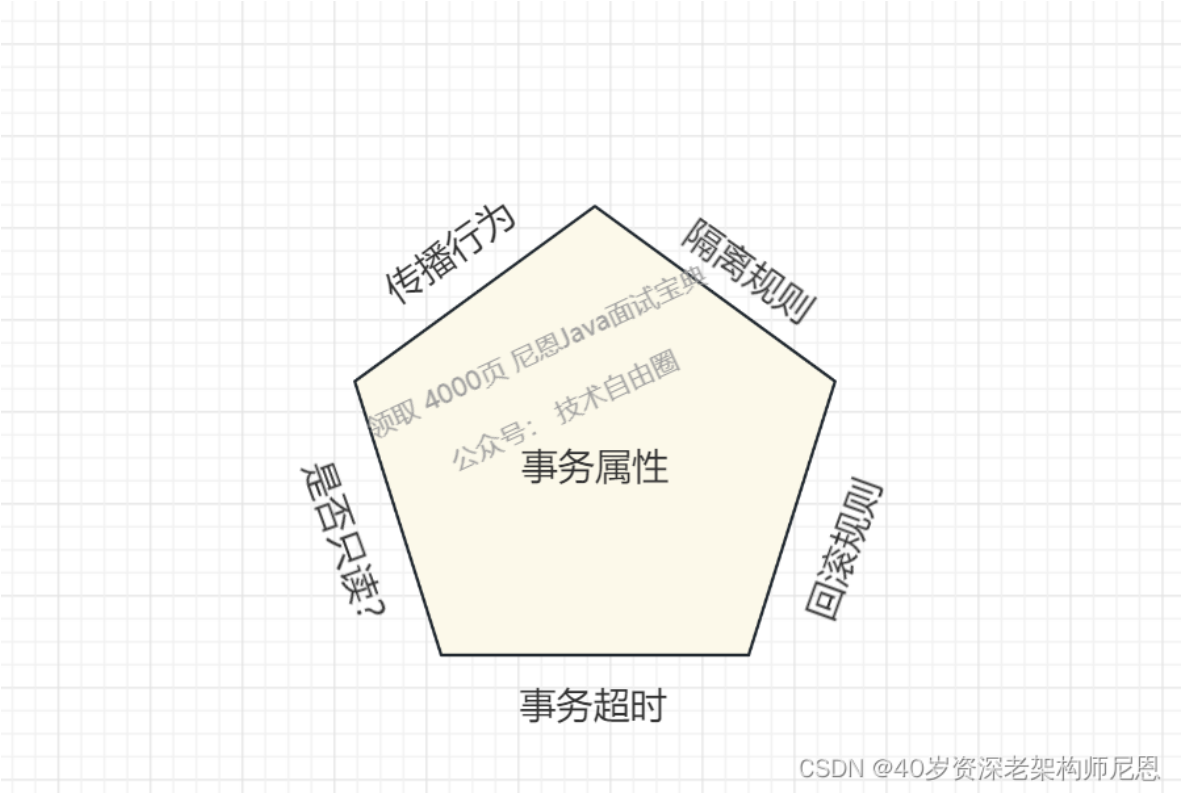
如何开启事务?

事务管理器接口PlatformTransactionManager通过getTransaction(TransactionDefinition definition)方法来开启一个事务，这个方法里面的参数是TransactionDefinition类，这个类就定义了一些基本的事务属性。

那么什么是事务属性呢?

事务属性可以理解成事务的一些基本配置，描述了事务策略如何应用到方法上。

总体来说，事务属性包含了5个方面，如图所示:



TransactionDefinition接口内容如下:

```
public interface TransactionDefinition {
    int getPropagationBehavior(); // 返回事务的传播行为
    int getIsolationLevel();     // 返回事务的隔离级别，事务管理器根据它来控制另外一个事务可以看到本事务内的哪些数据
    int getTimeout();  // 返回事务必须在多少秒内完成
    boolean isReadOnly(); // 事务是否只读，事务管理器能够根据这个返回值进行优化，确保事务是只读的
}
```

我们可以发现TransactionDefinition正好用来定义事务属性，下面详细介绍一下各个事务属性。

核心事务属性1：传播行为属性

事务的第一个方面是传播行为（propagation behavior）。

当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。

例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。

Spring定义了七种传播行为：

传播行为	含义
PROPAGATION_REQUIRED	表示当前方法必须运行在事务中。如果当前事务存在，方法将会在该事务中运行。否则，会启动一个新的事务
PROPAGATION_SUPPORTS	表示当前方法不需要事务上下文，但是如果存在当前事务的话，那么该方法会在这个事务中运行
PROPAGATION_MANDATORY	表示该方法必须在事务中运行，如果当前事务不存在，则会抛出一个异常
PROPAGATION_REQUIRED_NEW	表示当前方法必须运行在它自己的事务中。一个新的事务将被启动。如果存在当前事务，在该方法执行期间，当前事务会被挂起。如果使用JTATransactionManager的话，则需要访问TransactionManager
PROPAGATION_NOT_SUPPORTED	表示该方法不应该运行在事务中。如果存在当前事务，在该方法运行期间，当前事务将被挂起。如果使用JTATransactionManager的话，则需要访问TransactionManager
PROPAGATION_NEVER	表示当前方法不应该运行在事务上下文中。如果当前正有一个事务在运行，则会抛出异常
PROPAGATION_NESTED	表示如果当前已经存在一个事务，那么该方法将会在嵌套事务中运行。嵌套的事务可以独立于当前事务进行单独地提交或回滚。如果当前事务不存在，那么其行为与PROPAGATION_REQUIRED一样。注意各厂商对这种传播行为的支持是有所差异的。可以参考资源管理器的文档来确认它们是否支持嵌套事务

使用spring声明式事务，spring使用AOP来支持声明式事务，会根据事务属性，自动在方法调用之前决定是否开一个事务，并在方法执行之后决定事务提交或回滚事务。

（1）PROPAGATION_REQUIRED

PROPAGATION_REQUIRED 功能说明：

- 如果存在一个事务，则支持当前事务。
- 如果没有事务，则开启一个新的事务。

methodA 调用 methodB

```
//事务属性 PROPAGATION_REQUIRED
methodA{
    ……
    methodB();
    ……
}
```

methodB() 的代码

```
//事务属性 PROPAGATION_REQUIRED
methodB{
   ……
}
```

如果 单独调用methodB方法：

```
main{
    metodB();
}
```

Spring保证在methodB方法中所有的调用，都获得到一个相同的连接。在调用methodB时，没有一个存在的事务，所以获得一个新的连接，开启了一个新的事务。

相当于

```
Main{
    Connection con=null;
    try{
        con = getConnection();
        con.setAutoCommit(false);

        //方法调用
        methodB();

        //提交事务
        con.commit();
    } Catch(RuntimeException ex) {
        //回滚事务
        con.rollback();
    } finally {
        //释放资源
        closeCon();
    }
}
```

如果是 先调用MethodA时，在MethodA内又会调用MethodB. 执行效果相当于：

```
main{
    Connection con = null;
    try{
        con = getConnection();
        methodA();
        con.commit();
    } catch(RuntimeException ex) {
        con.rollback();
    } finally {
        closeCon();
    }
}
```

当在MethodA中调用MethodB时，环境中已经有了一个事务，所以methodB就加入当前事务。

（2）PROPAGATION_SUPPORTS

功能说明：

- 如果存在一个事务，支持当前事务。
- 如果没有事务，则非事务的执行。

但是对于事务同步的事务管理器，PROPAGATION_SUPPORTS与不使用事务有少许不同。

```
//事务属性 PROPAGATION_REQUIRED
methodA(){
  methodB();
}

//事务属性 PROPAGATION_SUPPORTS
methodB(){
  ……
}
```

单纯的调用methodB时，methodB方法是非事务的执行的。

当调用methdA时,methodB则加入了methodA的事务中,事务地执行。

（3）PROPAGATION_MANDATORY

功能说明：

- 如果已经存在一个事务，支持当前事务。
- 如果没有一个活动的事务，则抛出异常。

```
//事务属性 PROPAGATION_REQUIRED
methodA(){
    methodB();
}

//事务属性 PROPAGATION_MANDATORY
methodB(){
    ……
}
```

当单独调用methodB时，因为当前没有一个活动的事务，

则会抛出异常

```
throw new IllegalTransactionStateException("Transaction propagation 'mandatory' but no existing transaction found");
```

当调用methodA时，methodB则加入到methodA的事务中，事务地执行。

（4）PROPAGATION_REQUIRES_NEW

- 总是开启一个新的事务。
- 如果一个事务已经存在，则将这个存在的事务挂起。

下面是一个例子

```
//事务属性 PROPAGATION_REQUIRED
methodA(){
    doSomeThingA();
    methodB();
    doSomeThingB();
}

//事务属性 PROPAGATION_REQUIRES_NEW
methodB(){
    ……
}
```

调用A方法：

```
main(){
    methodA();
}
```

相当于

```
main(){
    TransactionManager tm = null;
    try{
        //获得一个JTA事务管理器
        tm = getTransactionManager();
        tm.begin();//开启一个新的事务
        Transaction ts1 = tm.getTransaction();
        doSomeThing();
        tm.suspend();//挂起当前事务
        try{
            tm.begin();//重新开启第二个事务
            Transaction ts2 = tm.getTransaction();
            methodB();
            ts2.commit();//提交第二个事务
        } Catch(RunTimeException ex) {
            ts2.rollback();//回滚第二个事务
        } finally {
            //释放资源
        }
        //methodB执行完后，恢复第一个事务
        tm.resume(ts1);
```

```
        doSomeThingB();
        ts1.commit();//提交第一个事务
    } catch(RunTimeException ex) {
        ts1.rollback();//回滚第一个事务
    } finally {
        //释放资源
    }
}
```

在这里，我把ts1称为外层事务，ts2称为内层事务。

从上面的代码可以看出，ts2与ts1是两个独立的事务，互不相干。

Ts2是否成功并不依赖于 ts1。

如果methodA方法在调用methodB方法后的doSomeThingB方法失败了，而methodB方法所做的结果依然被提交。而除了 methodB之外的其它代码导致的结果却被回滚了。

另外，使用PROPAGATION_REQUIRES_NEW，需要使用 JtaTransactionManager作为事务管理器。

（5）PROPAGATION_NOT_SUPPORTED

功能说明：

- 总是非事务地执行，并挂起任何存在的事务。

使用PROPAGATION_NOT_SUPPORTED,也需要使用JtaTransactionManager作为事务管理器。

（6）PROPAGATION_NEVER

功能说明：

- 总是非事务地执行
- 如果存在一个活动事务，则抛出异常。

（7）PROPAGATION_NESTED

功能说明：

- 如果一个活动的事务存在，则运行在一个嵌套的事务中.
- 如果没有活动事务, 则按TransactionDefinition.PROPAGATION_REQUIRED 属性执行。

这是一个嵌套事务,使用JDBC 3.0驱动时,仅仅支持DataSourceTransactionManager作为事务管理器。

需要JDBC 驱动的java.sql.Savepoint类。有一些JTA的事务管理器实现可能也提供了同样的功能。

使用PROPAGATION_NESTED，还需要把PlatformTransactionManager的nestedTransactionAllowed属性设为true;而 nestedTransactionAllowed属性值默认为false。

```
//事务属性 PROPAGATION_REQUIRED
methodA(){
    doSomeThingA();
    methodB();
    doSomeThingB();
}

//事务属性 PROPAGATION_NESTED
methodB(){
    ……
}
```

如果单独调用methodB方法，则按REQUIRED属性执行。如果调用methodA方法，相当于下面的效果：

```
main(){
    Connection con = null;
    Savepoint savepoint = null;
    try{
        con = getConnection();
        con.setAutoCommit(false);
        doSomeThingA();
        savepoint = con2.setSavepoint();
        try{
            methodB();
        } catch(RuntimeException ex) {
            con.rollback(savepoint);
        } finally {
            //释放资源
        }
        doSomeThingB();
        con.commit();
    } catch(RuntimeException ex) {
        con.rollback();
    } finally {
        //释放资源
    }
}
```

当methodB方法调用之前，调用setSavepoint方法，保存当前的状态到savepoint。

如果methodB方法调用失败，则恢复到之前保存的状态。

但是需要注意的是，这时的事务并没有进行提交，如果后续的代码(doSomeThingB()方法)调用失败，则回滚包括methodB方法的所有操作。

嵌套事务一个非常重要的概念就是内层事务依赖于外层事务。外层事务失败时，会回滚内层事务所做的动作。而内层事务操作失败并不会引起外层事务的回滚。

PROPAGATION_NESTED 与PROPAGATION_REQUIRES_NEW的区别:它们非常类似,都像一个嵌套事务，如果不存在一个活动的事务，都会开启一个新的事务。使用 PROPAGATION_REQUIRES_NEW时，内层事务与外层事务就像两个独立的事务一样，一旦内层事务进行了提交后，外层事务不能对其进行回滚。两个事务互不影响。两个事务不是一个真正的嵌套事务。同时它需要JTA事务管理器的支持。

使用PROPAGATION_NESTED时，外层事务的回滚可以引起内层事务的回滚。而内层事务的异常并不会导致外层事务的回滚，它是一个真正的嵌套事务。DataSourceTransactionManager使用savepoint支持PROPAGATION_NESTED时，需要JDBC 3.0以上驱动及1.4以上的JDK版本支持。其它的JTA TrasactionManager实现可能有不同的支持方式。

PROPAGATION_REQUIRES_NEW 启动一个新的, 不依赖于环境的 “内部” 事务. 这个事务将被完全 commited 或 rolled back 而不依赖于外部事务, 它拥有自己的隔离范围, 自己的锁, 等等. 当内部事务开始执行时, 外部事务将被挂起, 内务事务结束时, 外部事务将继续执行。

另一方面, PROPAGATION_NESTED 开始一个 “嵌套的” 事务, 它是已经存在事务的一个真正的子事务. 潜套事务开始执行时, 它将取得一个 savepoint. 如果这个嵌套事务失败, 我们将回滚到此 savepoint. 潜套事务是外部事务的一部分, 只有外部事务结束后它才会被提交。

由此可见, PROPAGATION_REQUIRES_NEW 和 PROPAGATION_NESTED 的最大区别在于,

> PROPAGATION_REQUIRES_NEW 完全是一个新的事务, 而 PROPAGATION_NESTED 则是外部事务的子事务, 如果外部事务 commit, 嵌套事务也会被 commit, 这个规则同样适用于 roll back.

事务的传播属性，当如何选择?

一般情况下，PROPAGATION_REQUIRED应该是我们首先的事务传播行为。它能够满足我们大多数的事务需求。

核心事务属性2：隔离级别属性

事务的第二个维度就是隔离级别（isolation level）。

隔离级别定义了一个事务可能受其他并发事务影响的程度。

（1）并发事务引起的问题

在典型的应用程序中，多个事务并发运行，经常会操作相同的数据来完成各自的任务。并发虽然是必须的，但可能会导致一下的问题。

> - 脏读（Dirty reads）——脏读发生在一个事务读取了另一个事务改写但尚未提交的数据时。如果改写在稍后被回滚了，那么第一个事务获取的数据就是无效的。
> - 不可重复读（Nonrepeatable read）——不可重复读发生在一个事务执行相同的查询两次或两次以上，但是每次都得到不同的数据时。这通常是因为另一个并发事务在两次查询期间进行了更新。
> - 幻读（Phantom read）——幻读与不可重复读类似。它发生在一个事务（T1）读取了几行数据，接着另一个并发事务（T2）插入了一些数据时。在随后的查询中，第一个事务（T1）就会发现多了一些原本不存在的记录。

不可重复读与幻读的区别

- （1）不可重复读的重点是修改
- （2）幻读的重点在于新增或者删除：

（1）不可重复读的重点是修改:

同样的条件, 你读取过的数据, 再次读取出来发现值不一样了
例如：在事务1中，Mary 读取了自己的工资为1000,操作并没有完成

```
con1 = getConnection();
select salary from employee empId ="Mary";
```

在事务2中，这时财务人员修改了Mary的工资为2000,并提交了事务.

```
con2 = getConnection();
update employee set salary = 2000;
con2.commit();
```

在事务1中，Mary 再次读取自己的工资时，工资变为了2000

```
//con1
select salary from employee empId ="Mary";
```

在一个事务中前后两次读取的结果并不一致，导致了不可重复读。

（2）幻读的重点在于新增或者删除：

同样的条件, 第1次和第2次读出来的记录数不一样
例如：目前工资为1000的员工有10人。

事务1,读取所有工资为1000的员工。

```
con1 = getConnection();
Select * from employee where salary =1000; 12
```

共读取10条记录

这时另一个事务向employee表插入了一条员工记录，工资也为1000

```
con2 = getConnection();
Insert into employee(empId,salary) values("Lili",1000);
con2.commit();
```

事务1再次读取所有工资为1000的员工

```
//con1
select * from employee where salary =1000;
```

共读取到了11条记录，这就产生了幻像读。

从总的结果来看, 似乎不可重复读和幻读都表现为两次读取的结果不一致。

但如果你从控制的角度来看, 两者的区别就比较大。

- 对于不可重复读, 只需要锁住满足条件的记录。
- 对于幻读, 要锁住满足条件及其相近的记录。

Spring 隔离级别的定义

数据库有自己的隔离级别的定义，Spring也有自己的 隔离级别的定义

隔离级别	含义
ISOLATION_DEFAULT	使用后端数据库默认的隔离级别
ISOLATION_READ_UNCOMMITTED	最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读
ISOLATION_READ_COMMITTED	允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生
ISOLATION_REPEATABLE_READ	对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生
ISOLATION_SERIALIZABLE	最高的隔离级别，完全服从ACID的隔离级别，确保阻止脏读、不可重复读以及幻读，也是最慢的事务隔离级别，因为它通常是通过完全锁定事务相关的数据库表来实现的

核心属性3： 事务只读属性

事务只读属性: 表示这个事务只读取数据但不更新数据， 这样可以帮助数据库引擎优化事务。

具体来说： 如果事务只对后端的数据库进行读操作，数据库可以利用事务的只读特性来进行一些特定的优化。通过将事务设置为只读，你就可以给数据库一个机会，让它应用它认为合适的优化措施。

当使用 @Transaction 注解时，可以通过设置 `readOnly=true` 来指定这是一个只读事务，这样在事务执行期间就不会对数据进行修改，只会进行查询操作。

以下是一个使用 @Transaction 只读示例的代码片段：

```
@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    @Transactional(readOnly = true)
    public User getUserById(Long id) {
        return userRepository.findById(id).orElse(null);
    }

    // 其他方法...
}
```

在上面的示例中，`getUserById` 方法被标记为只读事务，因此在执行期间只会进行查询操作。

如果在方法中尝试进行修改操作，将会抛出异常。

核心属性4：事务超时属性

事务超时属性：事务在强制回滚之前可以保持多久。这样可以防止长期运行的事务占用资源。

为了使应用程序很好地运行，事务不能运行太长的时间。

因为事务可能涉及对后端数据库的锁定，所以长时间的事务会不必要的占用数据库资源。

事务超时就是事务的一个定时器，在特定时间内事务如果没有执行完毕，那么就会自动回滚，而不是一直等待其结束。

`@Transactional` 注解中的 `timeout` 属性用于设置事务的超时时间，单位为秒。如果事务在超过指定时间后仍未完成，则会被强制回滚。

以下是一个示例：

```
@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;

    @Transactional(timeout = 5)
    public void updateUser(User user) {
        userRepository.save(user);
        // 执行一些其他操作
    }
}
```

在上面的示例中，`updateUser` 方法使用了 `@Transactional` 注解，并设置了 `timeout` 属性为 5 秒。如果该方法执行时间超过 5 秒，事务将会被强制回滚。

需要注意的是，如果在事务内部调用了其他带有事务注解的方法，那么这些方法的超时时间也会受到影响。

因此，在设置事务超时时间时，需要考虑到事务内部所涉及的所有方法的执行时间。

核心属性5：Spring事务的回滚规则

事务五边形的最后一个方面是一组规则，这些规则定义了哪些异常会导致事务回滚而哪些不会。

好的，下面是一个简单的 Java 代码示例，演示了 `@Transactional` 注解的使用以及如何在出现异常时回滚事务：

```
@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    @Transactional
    public void createUser(User user) {
        userRepository.save(user);
        if (user.getId() == null) {
            throw new RuntimeException("Failed to create user");
        }
    }
}
```

在上面的示例中，`createUser` 方法上使用了 `@Transactional` 注解，表示这个方法需要在事务管理下执行。如果在方法执行过程中发生异常，事务会自动回滚，保证数据的一致性。

在这个例子中，如果用户创建失败，`createUser` 方法会抛出一个 `RuntimeException` 异常，这会导致事务回滚，用户创建操作会被撤销。

需要注意的是，`@Transactional` 注解只能应用于公共方法，因为只有公共方法才能被代理，从而实现事务管理。

此外，`@Transactional` 注解默认只对受检查异常进行回滚，而对非受检查异常不进行回滚。

> 非检查型异常指（Unchecked Exception）的是程序在编译时不会提示需要处理该异常，而是在运行时才会出现异常。
>
> 检查型异常（Checked Exception）是指在 Java 中，编译器会强制要求对可能会抛出这些异常的代码进行异常处理，否则代码将无法通过编译。

一般来说，在编写代码时应该尽量避免抛出非检查型异常，因为这些异常的发生通常意味着程序存在严重的逻辑问题。

如果是**检查型异常（Checked Exception）**，可以声明事务在遇到特定的检查型异常时像遇到运行期异常那样回滚。同样，你还可以声明事务遇到特定的异常不回滚，即使这些异常是运行期异常。

如果需要对非受检查异常也进行回滚，可以在 `@Transactional` 注解中指定 `rollbackFor` 属性，例如

```
@Transactional(rollbackFor = Exception.class)
public void createUser(User user) {
      userRepository.save(user);
       if (user.getId() == null) {
            throw new RuntimeException("Failed to create user");
      }
}
```

核心流程一：AOP 动态代理装配流程源码剖析

注解@EnableTransactionManagement

我们知道了使用 AOP 技术实现，那到底是如何实现的呢?

我们从 @EnableTransactionManagement 注解聊起，我们点进该注解：

```
@Import(TransactionManagementConfigurationSelector.class)
public @interface EnableTransactionManagement {

```

很明显，TransactionManagementConfigurationSelector 类是我们主要关注的内容

```
public class TransactionManagementConfigurationSelector extends
AdviceModeImportSelector<EnableTransactionManagement> {

   /**
    * 此处是AdviceMode的作用，默认是用代理，另外一个是ASPECTJ
    */
   @Override
   protected String[] selectImports(AdviceMode adviceMode) {
      switch (adviceMode) {
         case PROXY:
            return new String[] {AutoProxyRegistrar.class.getName(),
                  ProxyTransactionManagementConfiguration.class.getName()};
         case ASPECTJ:
            return new String[] {determineTransactionAspectClass()};
         default:
            return null;
      }
   }
}
```

一共注册了两个：

- AutoProxyRegistrar.class：注册AOP处理器
- ProxyTransactionManagementConfiguration.class：代理事务配置，

注册事务需要用的一些类，而且Role=ROLE_INFRASTRUCTURE都是属于内部级别的

```
@Configuration(proxyBeanMethods = false)
@Role(BeanDefinition.ROLE_INFRASTRUCTURE)
public class ProxyTransactionManagementConfiguration extends
AbstractTransactionManagementConfiguration {

```

```
4
5     @Bean(name =
TransactionManagementConfigUtils.TRANSACTION_ADVISOR_BEAN_NAME)
6     @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
7     public BeanFactoryTransactionAttributeSourceAdvisor transactionAdvisor(
8           TransactionAttributeSource transactionAttributeSource,
TransactionInterceptor transactionInterceptor) {
9          // 【重点】注册了 BeanFactoryTransactionAttributeSourceAdvisor 的
advisor
10         // 其 advice 为 transactionInterceptor
11        BeanFactoryTransactionAttributeSourceAdvisor advisor = new
BeanFactoryTransactionAttributeSourceAdvisor();
12        advisor.setTransactionAttributeSource(transactionAttributeSource);
13        advisor.setAdvice(transactionInterceptor);
14        if (this.enableTx != null) {
15           advisor.setOrder(this.enableTx.<Integer>getNumber("order"));
16        }
17        return advisor;
18     }
19
20     @Bean
21     @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
22     public TransactionAttributeSource transactionAttributeSource() {
23        return new AnnotationTransactionAttributeSource();
24     }
25
26     @Bean
27     @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
28     public TransactionInterceptor
transactionInterceptor(TransactionAttributeSource
transactionAttributeSource) {
29        TransactionInterceptor interceptor = new TransactionInterceptor();
30
 interceptor.setTransactionAttributeSource(transactionAttributeSource);
31        if (this.txManager != null) {
32           interceptor.setTransactionManager(this.txManager);
33        }
34        return interceptor;
35     }
36 }
```

到这里，看到BeanFactoryTransactionAttributeSourceAdvisor 以 advisor 结尾的类，这就是进行创建AOP动态代理对象的核心类，其 Interceptor为 **transactionInterceptor**

到这里，我们思考一下，利用我们之前学习到的 AOP 的源码，猜测其运行逻辑：

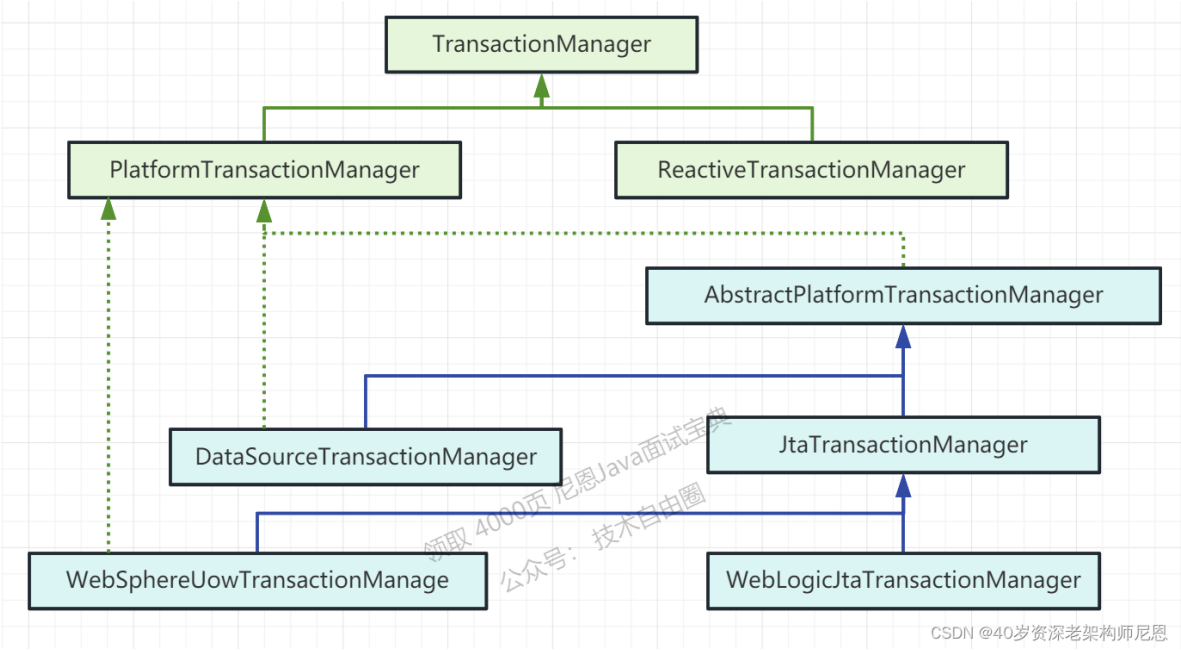
- 我们在方法上写上 @EnableTransactionManagement 注解，Spring 会注册一个 BeanFactoryTransactionAttributeSourceAdvisor 的类
- 创建对应的方法 Bean 时，会和 AOP 一样，利用该 Advisor 类生成对应的代理对象
- 最终调用方法时，会调用代理对象，并通过**环绕增强**来达到事务的功能

核心流程二：AOP 动态代理执行流程源码剖析

动态代理对象的代理方法被执行后，会执行到 Interceptor 拦截器。

我们从上面看到其 Interceptor正是 TransactionInterceptor。代理对象运行时，会拿到所有的Interceptor并进行排序，责任链递归运行

这里先看一下 TransactionInterceptor 类图



然后看其源码内容：

这里的 invoke 方法执行的时候，会执行invokeWithinTransaction ，具体看代码

```
public class TransactionInterceptor extends TransactionAspectSupport
implements MethodInterceptor {
   @Override
    @Nullable
    public Object invoke(MethodInvocation invocation) throws Throwable {
        // 获取我们的代理对象的class属性
        Class<?> targetClass = (invocation.getThis() != null ?
AopUtils.getTargetClass(invocation.getThis()) : null);

        // Adapt to TransactionAspectSupport's invokeWithinTransaction...
        /**
         * 以事务的方式调用目标方法
         * 在这埋了一个钩子函数 用来回调目标方法的
         */
        return invokeWithinTransaction(invocation.getMethod(), targetClass,
invocation::proceed);
    }
}

@Nullable
protected Object invokeWithinTransaction(Method method, @Nullable Class<?>
targetClass, final InvocationCallback invocation){
       // 获取我们的事务属性源对象
        TransactionAttributeSource tas = getTransactionAttributeSource();
        // 通过事务属性源对象获取到当前方法的事务属性信息
        final TransactionAttribute txAttr = (tas != null ?
tas.getTransactionAttribute(method, targetClass) : null);
```

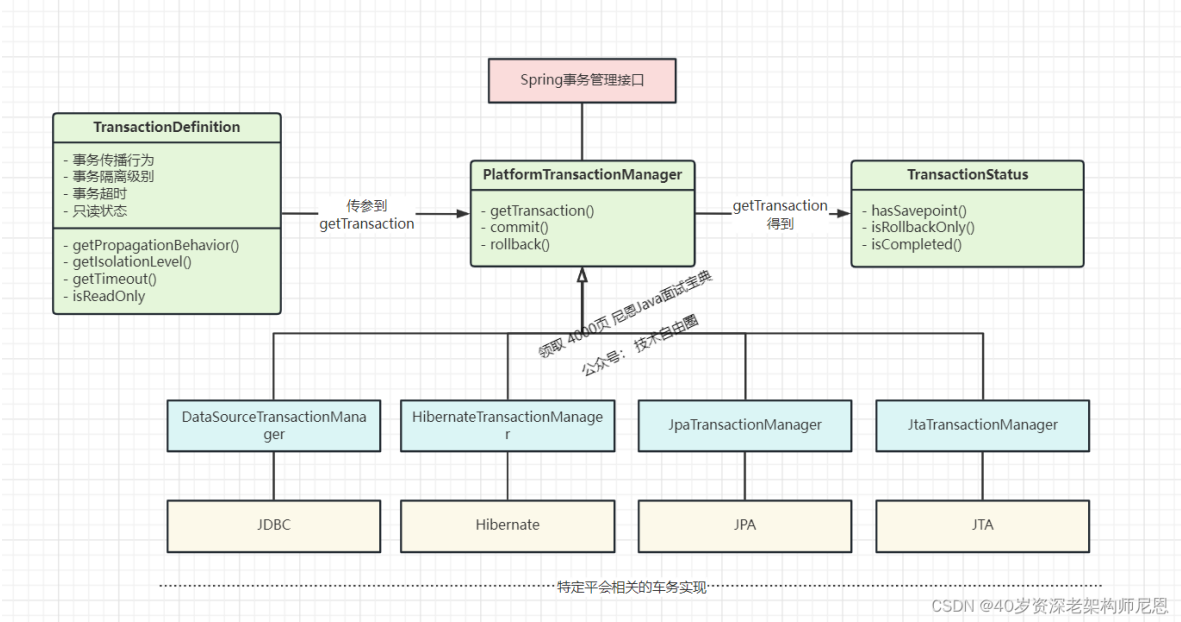
```
23
24        // 获取我们配置的事务管理器对象
25          final TransactionManager tm = determineTransactionManager(txAttr);
26
27      if (txAttr == null || !(ptm instanceof
CallbackPreferringPlatformTransactionManager)) {
28          // 【重点】创建TransactionInfo
29            TransactionInfo txInfo = createTransactionIfNecessary(ptm,
txAttr, joinpointIdentification);
30          try {
31                  // 执行被增强方法,调用具体的处理逻辑
32                  // 我们实际的业务方法
33                  retVal = invocation.proceedWithInvocation();
34              }
35              catch (Throwable ex) {
36                  // 异常回滚， 事务回滚
37                  completeTransactionAfterThrowing(txInfo, ex);
38                  throw ex;
39              }
40              finally {
41                  //清除事务信息，恢复线程私有的老的事务信息
42                  cleanupTransactionInfo(txInfo);
43              }
44            // 提交事务
45            //成功后提交，会进行资源储量，连接释放，恢复挂起事务等操作
46            commitTransactionAfterReturning(txInfo);
47          return retVal;
48      }
49  }
50
51  // 创建连接 + 开启事务
52  protected TransactionInfo createTransactionIfNecessary(@Nullable
PlatformTransactionManager tm,
53              @Nullable TransactionAttribute txAttr, final String
joinpointIdentification) {
54      // 获取TransactionStatus事务状态信息
55      status = tm.getTransaction(txAttr);
56
57      // 根据指定的属性与status准备一个TransactionInfo,
58      return prepareTransactionInfo(tm, txAttr, joinpointIdentification,
status);
59  }
60
61  // 存在异常时回滚事务
62  protected void completeTransactionAfterThrowing(@Nullable TransactionInfo
txInfo, Throwable ex) {
63      // 进行回滚
64      txInfo.getTransactionManager().rollback(txInfo.getTransactionStatus());
65  }
66
67  // 调用事务管理器的提交方法
68  protected void commitTransactionAfterReturning(@Nullable TransactionInfo
txInfo){
69      txInfo.getTransactionManager().commit(txInfo.getTransactionStatus());
70  }
71
```

从上面的拦截器可以看到

- 在业务方法前面，开启事务
- 业务方法后面，捕获异常，执行事务回滚
- 如果没有异常，提交事务

核心流程三：事务执行流程源码剖析

Spring 事务设计了统一的事务管理器超级接口， **PlatformTransactionManager** 是 Spring事务管理器设计的 基类。



先看看看PlatformTransactionManager 的实现，其实现一共三个方法：

- 开启事务：TransactionStatus getTransaction(@Nullable TransactionDefinition definition)
- 提交事务：void commit(TransactionStatus status)
- 回滚事务：void rollback(TransactionStatus status)

接下来，我们分别看一下其如何实现的

开启事务源码分析

我们想一下，在开启事务这一阶段，我们会做什么功能呢？ 参考DataSourceTransactionManager，这个阶段应该会 **创建连接并且开启事务**

```
1  public final TransactionStatus getTransaction(@Nullable
   TransactionDefinition definition){
2
3      // PROPAGATION_REQUIRED，PROPAGATION_REQUIRES_NEW，PROPAGATION_NESTED都需
   要新建事务
```

```
    if (def.getPropagationBehavior() ==
TransactionDefinition.PROPAGATION_REQUIRED ||
        def.getPropagationBehavior() ==
TransactionDefinition.PROPAGATION_REQUIRES_NEW ||
        def.getPropagationBehavior() ==
TransactionDefinition.PROPAGATION_NESTED) {
        //没有当前事务的话，REQUIRED，REQUIRES_NEW，NESTED挂起的是空事务，然后创建一
个新事务
        SuspendedResourcesHolder suspendedResources = suspend(null);
        try {
            // 看这里重点：开始事务
            return startTransaction(def, transaction, debugEnabled,
suspendedResources);
        }
        catch (RuntimeException | Error ex) {
            // 恢复挂起的事务
            resume(null, suspendedResources);
            throw ex;
        }
    }
}

private TransactionStatus startTransaction(TransactionDefinition
definition, Object transaction, boolean debugEnabled, @Nullable
SuspendedResourcesHolder suspendedResources) {
    // 是否需要新同步
    boolean newSynchronization = (getTransactionSynchronization() !=
SYNCHRONIZATION_NEVER);
    // 创建新的事务
    DefaultTransactionStatus status = newTransactionStatus( definition,
transaction, true, newSynchronization, debugEnabled, suspendedResources);
    // 【重点】开启事务和连接
    doBegin(transaction, definition);
    // 新同步事务的设置，针对于当前线程的设置
    prepareSynchronization(status, definition);
    return status;
}

protected void doBegin(Object transaction, TransactionDefinition
definition) {
    // 判断事务对象没有数据库连接持有器
    if (!txObject.hasConnectionHolder() ||
        txObject.getConnectionHolder().isSynchronizedWithTransaction()) {
        // 【重点】通过数据源获取一个数据库连接对象
        Connection newCon = obtainDataSource().getConnection();
        // 把我们的数据库连接包装成一个ConnectionHolder对象 然后设置到我们的txObject
对象中去
        // 再次进来时，该 txObject 就已经有事务配置了
        txObject.setConnectionHolder(new ConnectionHolder(newCon), true);
    }

    // 【重点】获取连接
    con = txObject.getConnectionHolder().getConnection();


    // 为当前的事务设置隔离级别【数据库的隔离级别】
    Integer previousIsolationLevel =
DataSourceUtils.prepareConnectionForTransaction(con, definition);
```

```
    // 设置先前隔离级别
    txObject.setPreviousIsolationLevel(previousIsolationLevel);
    // 设置是否只读
    txObject.setReadOnly(definition.isReadOnly());

    // 关闭自动提交
    if (con.getAutoCommit()) {
        //设置需要恢复自动提交
        txObject.setMustRestoreAutoCommit(true);
        // 【重点】关闭自动提交
        con.setAutoCommit(false);
    }

    // 判断事务是否需要设置为只读事务
    prepareTransactionalConnection(con, definition);
    // 标记激活事务
    txObject.getConnectionHolder().setTransactionActive(true);

    // 设置事务超时时间
    int timeout = determineTimeout(definition);
    if (timeout != TransactionDefinition.TIMEOUT_DEFAULT) {
        txObject.getConnectionHolder().setTimeoutInSeconds(timeout);
    }

    // 绑定我们的数据源和连接到我们的同步管理器上，把数据源作为key,数据库连接作为value设置到线程变量中
    if (txObject.isNewConnectionHolder()) {
        // 将当前获取到的连接绑定到当前线程
        TransactionSynchronizationManager.bindResource(obtainDataSource(), txObject.getConnectionHolder());
    }
}
}
```

到这里，我们的 获取事务 接口完成了 数据库连接的创建 和 关闭自动提交（开启事务），将 Connection 注册到了缓存（resources）当中，便于获取。

提交事务源码分析

```
public final void commit(TransactionStatus status) throws TransactionException {
        DefaultTransactionStatus defStatus = (DefaultTransactionStatus) status;
        // 如果在事务链中已经被标记回滚，那么不会尝试提交事务，直接回滚
        if (defStatus.isLocalRollbackOnly()) {
            // 不可预期的回滚
            processRollback(defStatus, false);
            return;
        }

        // 设置了全局回滚
        if (!shouldCommitOnGlobalRollbackOnly() && defStatus.isGlobalRollbackOnly()) {
            // 可预期的回滚，可能会报异常
            processRollback(defStatus, true);
            return;
        }
```

```
        // 【重点】处理事务提交
        processCommit(defStatus);
}

// 处理提交，先处理保存点，然后处理新事务，如果不是新事务不会真正提交，要等外层是新事务的才提交，
// 最后根据条件执行数据清除,线程的私有资源解绑，重置连接自动提交，隔离级别，是否只读，释放连接，恢复挂起事务等
private void processCommit(DefaultTransactionStatus status) throws TransactionException {;
     // 如果是独立的事务则直接提交
      doCommit(status);

  //根据条件，完成后数据清除,和线程的私有资源解绑，重置连接自动提交，隔离级别，是否只读，释放连接，恢复挂起事务等
      cleanupAfterCompletion(status);
}
```

这里比较重要的有两个步骤：

doCommit：提交事务（直接使用 JDBC 提交即可）

```
protected void doCommit(DefaultTransactionStatus status) {
   DataSourceTransactionObject txObject = (DataSourceTransactionObject) status.getTransaction();
   Connection con = txObject.getConnectionHolder().getConnection();
   try {
      // JDBC连接提交
      con.commit();
   }
   catch (SQLException ex) {
      throw new TransactionSystemException("Could not commit JDBC transaction", ex);
   }
}
```

cleanupAfterCompletion：数据清除，与线程中的私有资源解绑，方便释放

```
// 线程同步状态清除
TransactionSynchronizationManager.clear();

// 清除同步状态【这些都是线程的缓存，使用ThreadLocal的】
public static void clear() {
    synchronizations.remove();
    currentTransactionName.remove();
    currentTransactionReadOnly.remove();
    currentTransactionIsolationLevel.remove();
    actualTransactionActive.remove();
}
// 如果是新事务的话，进行数据清除，线程的私有资源解绑，重置连接自动提交，隔离级别，是否只读，释放连接等
doCleanupAfterCompletion(status.getTransaction());

// 此方法做清除连接相关操作，比如重置自动提交啊，只读属性啊，解绑数据源啊，释放连接啊，清除链接持有器属性
protected void doCleanupAfterCompletion(Object transaction) {
```

```
    DataSourceTransactionObject txObject = (DataSourceTransactionObject)
transaction;
    // 将数据库连接从当前线程中解除绑定
    TransactionSynchronizationManager.unbindResource(obtainDataSource());

    // 释放连接
    Connection con = txObject.getConnectionHolder().getConnection();

    // 恢复数据库连接的自动提交属性
    con.setAutoCommit(true);
    // 重置数据库连接
    DataSourceUtils.resetConnectionAfterTransaction(con,
txObject.getPreviousIsolationLevel(), txObject.isReadOnly());

    // 如果当前事务是独立的新创建的事务则在事务完成时释放数据库连接
    DataSourceUtils.releaseConnection(con, this.dataSource);


    // 连接持有器属性清除
    txObject.getConnectionHolder().clear();
}
```

这就是我们提交事务的操作了，总之来说，主要就是 调用JDBC的commit提交 和 清除一系列的线程内部数据和配置

回滚事务源码分析

```
public final void rollback(TransactionStatus status) throws
TransactionException {
   DefaultTransactionStatus defStatus = (DefaultTransactionStatus) status;
   processRollback(defStatus, false);
}

private void processRollback(DefaultTransactionStatus status, boolean
unexpected) {
     // 回滚的擦欧洲哦
    doRollback(status);

    // 回滚完成后回调
    triggerAfterCompletion(status,
TransactionSynchronization.STATUS_ROLLED_BACK);

     // 根据事务状态信息，完成后数据清除，和线程的私有资源解绑，重置连接自动提交，隔离级别，是否只读，释放连接，恢复挂起事务等
    cleanupAfterCompletion(status);
}

protected void doRollback(DefaultTransactionStatus status) {
        DataSourceTransactionObject txObject =
(DataSourceTransactionObject) status.getTransaction();
        Connection con = txObject.getConnectionHolder().getConnection();
        // jdbc的回滚
        con.rollback();
}
```

回滚事务，简单来说 调用JDBC的rollback 和 清除数据

尼恩总结

很多的小伙伴，当时对 Spring 懵懂无知，或者，看完源码，转眼间忘记了

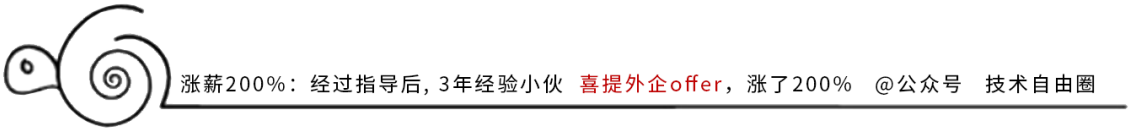
非常痛苦

看了此文，记住尼恩梳理的2大模式和三大核心流程

再也不会忘记Spring事务的核心源码了。

通过这篇顶奢好文，99.99% 的人应该都可以理解了 Spring 事务 的来龙去脉.

如果做不到，可以看10遍，一定就可以做到了。



专题6：Spring MVC面试题 专题部分

什么是Spring MVC？简单介绍下你对Spring MVC的理解?

Spring MVC是一个基于Java的实现了MVC设计模式的请求驱动类型的轻量级Web框架，通过把模型-视图-控制器分离，将web层进行职责解耦，把复杂的web应用分成逻辑清晰的几部分，简化开发，减少出错，方便组内开发人员之间的配合。

Spring MVC的优点

（1）可以支持各种视图技术,而不仅仅局限于JSP；

（2）与Spring框架集成（如IoC容器、AOP等）；

（3）清晰的角色分配：前端控制器(dispatcherServlet) , 请求到处理器映射（handlerMapping), 处理器适配器（HandlerAdapter), 视图解析器（ViewResolver）。

（4） 支持各种请求资源的映射策略。

核心组件

Spring MVC的主要组件?

（1）前端控制器 DispatcherServlet（不需要程序员开发）

作用：接收请求、响应结果，相当于转发器，有了DispatcherServlet 就减少了其它组件之间的耦合度。

（2）处理器映射器HandlerMapping（不需要程序员开发）

作用：根据请求的URL来查找Handler

（3）处理器适配器HandlerAdapter

注意：在编写Handler的时候要按照HandlerAdapter要求的规则去编写，这样适配器HandlerAdapter才可以正确的去执行Handler。

（4）处理器Handler（需要程序员开发）

（5）视图解析器 ViewResolver（不需要程序员开发）

作用：进行视图的解析，根据视图逻辑名解析成真正的视图（view）

（6）视图View（需要程序员开发jsp）

View是一个接口， 它的实现类支持不同的视图类型（jsp，freemarker，pdf等等）

什么是DispatcherServlet

Spring的MVC框架是围绕DispatcherServlet来设计的，它用来处理所有的HTTP请求和响应。

什么是Spring MVC框架的控制器?

控制器提供一个访问应用程序的行为，此行为通常通过服务接口实现。控制器解析用户输入并将其转换为一个由视图呈现给用户的模型。Spring用一个非常抽象的方式实现了一个控制层，允许用户创建多种用途的控制器。

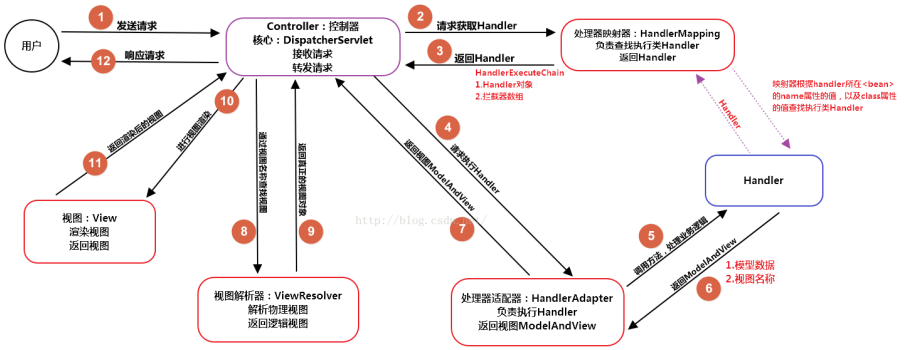
Spring MVC的控制器是不是单例模式,如果是,有什么问题,怎么解决?

答：是单例模式,所以在多线程访问的时候有线程安全问题,不要用同步,会影响性能的,解决方案是在控制器里面不能写字段。

工作原理

请描述Spring MVC的工作流程？描述一下 DispatcherServlet 的工作流程?

（1）用户发送请求至前端控制器DispatcherServlet；
（2） DispatcherServlet收到请求后，调用HandlerMapping处理器映射器，请求获取Handle；
（3）处理器映射器根据请求url找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet；
（4）DispatcherServlet 调用 HandlerAdapter处理器适配器；
（5）HandlerAdapter 经过适配调用 具体处理器(Handler，也叫后端控制器)；
（6）Handler执行完成返回ModelAndView；
（7）HandlerAdapter将Handler执行结果ModelAndView返回给DispatcherServlet；
（8）DispatcherServlet将ModelAndView传给ViewResolver视图解析器进行解析；
（9）ViewResolver解析后返回具体View；
（10）DispatcherServlet对View进行渲染视图（即将模型数据填充至视图中）
（11）DispatcherServlet响应用户。



MVC框架

MVC是什么？MVC设计模式的好处有哪些

mvc是一种设计模式（设计模式就是日常开发中编写代码的一种好的方法和经验的总结）。模型（model）-视图（view）-控制器（controller），三层架构的设计模式。用于实现前端页面的展现与后端业务数据处理的分离。

mvc设计模式的好处

1.分层设计，实现了业务系统各个组件之间的解耦，有利于业务系统的可扩展性，可维护性。

2.有利于系统的并行开发，提升开发效率。

常用注解

注解原理是什么

注解本质是一个继承了Annotation的特殊接口，其具体实现类是Java运行时生成的动态代理类。我们通过反射获取注解时，返回的是Java运行时生成的动态代理对象。通过代理对象调用自定义注解的方法，会最终调用AnnotationInvocationHandler的invoke方法。该方法会从memberValues这个Map中索引出对应的值。而memberValues的来源是Java常量池。

Spring MVC常用的注解有哪些?

@RequestMapping：用于处理请求 url 映射的注解，可用于类或方法上。用于类上，则表示类中的所有响应请求的方法都是以该地址作为父路径。

@RequestBody：注解实现接收http请求的json数据，将json转换为java对象。

@ResponseBody：注解实现将conreoller方法返回对象转化为json对象响应给客户。

SpingMvc中的控制器的注解一般用哪个,有没有别的注解可以替代?

答：一般用@Controller注解,也可以使用@RestController,@RestController注解相当于@ResponseBody + @Controller,表示是表现层,除此之外，一般不用别的注解代替。

@Controller注解的作用

在Spring MVC 中，控制器Controller 负责处理由DispatcherServlet 分发的请求，它把用户请求的数据经过业务处理层处理之后封装成一个Model ，然后再把该Model 返回给对应的View 进行展示。在Spring MVC 中提供了一个非常简便的定义Controller 的方法，你无需继承特定的类或实现特定的接口，只需使用@Controller 标记一个类是Controller ，然后使用@RequestMapping 和@RequestParam 等一些注解用以定义URL 请求和Controller 方法之间的映射，这样的Controller 就能被外界访问到。此外Controller 不会直接依赖于HttpServletRequest 和HttpServletResponse 等HttpServlet 对象，它们可以通过Controller 的方法参数灵活的获取到。

@Controller 用于标记在一个类上，使用它标记的类就是一个Spring MVC Controller 对象。分发处理器将会扫描使用了该注解的类的方法，并检测该方法是否使用了@RequestMapping 注解。@Controller 只是定义了一个控制器类，而使用@RequestMapping 注解的方法才是真正处理请求的处理器。单单使用@Controller 标记在一个类上还不能真正意义上的说它就是Spring MVC 的一个控制器类，因为这个时候Spring 还不认识它。那么要如何做Spring 才能认识它呢？这个时候就需要我们把这个控制器类交给Spring 来管理。有两种方式：

- 在Spring MVC 的配置文件中定义MyController 的bean 对象。
- 在Spring MVC 的配置文件中告诉Spring 该到哪里去找标记为@Controller 的Controller 控制器。

@RequestMapping注解的作用

RequestMapping是一个用来处理请求地址映射的注解，可用于类或方法上。用于类上，表示类中的所有响应请求的方法都是以该地址作为父路径。

RequestMapping注解有六个属性，下面我们把她分成三类进行说明（下面有相应示例）。

value， method

value： 指定请求的实际地址，指定的地址可以是URI Template 模式（后面将会说明）；

method： 指定请求的method类型， GET、POST、PUT、DELETE等；

consumes，produces

consumes： 指定处理请求的提交内容类型（Content-Type），例如application/json, text/html;

produces: 指定返回的内容类型，仅当request请求头中的(Accept)类型中包含该指定类型才返回；

params，headers

params： 指定request中必须包含某些参数值是，才让该方法处理。

headers： 指定request中必须包含某些指定的header值，才能让该方法处理请求。

@ResponseBody注解的作用

作用： 该注解用于将Controller的方法返回的对象，通过适当的HttpMessageConverter转换为指定格式后，写入到Response对象的body数据区。

使用时机：返回的数据不是html标签的页面，而是其他某种格式的数据时（如json、xml等）使用；

@PathVariable和@RequestParam的区别

请求路径上有个id的变量值，可以通过@PathVariable来获取 @RequestMapping(value = “/page/{id}”, method = RequestMethod.GET)

@RequestParam用来获得静态的URL请求入参 spring注解时action里用到。

其他

Spring MVC与Struts2区别

相同点

都是基于mvc的表现层框架，都用于web项目的开发。

不同点

1.前端控制器不一样。Spring MVC的前端控制器是servlet：DispatcherServlet。struts2的前端控制器是filter：StrutsPreparedAndExcutorFilter。

2.请求参数的接收方式不一样。Spring MVC是使用方法的形参接收请求的参数，基于方法的开发，线程安全，可以设计为单例或者多例的开发，推荐使用单例模式的开发（执行效率更高），默认就是单例开发模式。struts2是通过类的成员变量接收请求的参数，是基于类的开发，线程不安全，只能设计为多例的开发。

3.Struts采用值栈存储请求和响应的数据，通过OGNL存取数据，Spring MVC通过参数解析器是将request请求内容解析，并给方法形参赋值，将数据和视图封装成ModelAndView对象，最后又将ModelAndView中的模型数据通过reques域传输到页面。Jsp视图解析器默认使用jstl。

4.与spring整合不一样。Spring MVC是spring框架的一部分，不需要整合。在企业项目中，Spring MVC使用更多一些。

Spring MVC怎么样设定重定向和转发的?

（1）转发：在返回值前面加"forward:"，譬如"forward:user.do?name=method4"

（2）重定向：在返回值前面加"redirect:"，譬如"redirect:http://www.baidu.com"

Spring MVC怎么和AJAX相互调用的?

通过Jackson框架就可以把Java里面的对象直接转化成Js可以识别的Json对象。具体步骤如下：

（1）加入Jackson.jar

（2）在配置文件中配置json的映射

（3）在接受Ajax方法里面可以直接返回Object,List等,但方法前面要加上@ResponseBody注解。

如何解决POST请求中文乱码问题，GET的又如何处理呢?

（1）解决post请求乱码问题：

在web.xml中配置一个CharacterEncodingFilter过滤器，设置成utf-8；

```
<filter>
    <filter-name>CharacterEncodingFilter</filter-name>
    <filter-
class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

    <init-param>
        <param-name>encoding</param-name>
        <param-value>utf-8</param-value>
    </init-param>
</filter>

<filter-mapping>
    <filter-name>CharacterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
```

```
</filter-mapping>
```

（2）get请求中文参数出现乱码解决方法有两个：

①修改tomcat配置文件添加编码与工程编码一致，如下：

```
<ConnectorURIEncoding="utf-8" connectionTimeout="20000" port="8080"
protocol="HTTP/1.1" redirectPort="8443"/>
```

②另外一种方法对参数进行重新编码：

String userName = new String(request.getParamter("userName").getBytes("ISO8859-1"),"utf-8")

ISO8859-1是tomcat默认编码，需要将tomcat编码后的内容按utf-8编码。

Spring MVC的异常处理?

答：可以将异常抛给Spring框架，由Spring框架来处理；我们只需要配置简单的异常处理器，在异常处理器中添视图页面即可。

如果在拦截请求中，我想拦截get方式提交的方法,怎么配置

答：可以在@RequestMapping注解里面加上method=RequestMethod.GET。

怎样在方法里面得到Request,或者Session?

答：直接在方法的形参中声明request,Spring MVC就自动把request对象传入。

如果想在拦截的方法里面得到从前台传入的参数,怎么得到?

答：直接在形参里面声明这个参数就可以,但必须名字和传过来的参数一样。

如果前台有很多个参数传入,并且这些参数都是一个对象的,那么怎么样快速得到这个对象?

答：直接在方法中声明这个对象,Spring MVC就自动会把属性赋值到这个对象里面。

Spring MVC中函数的返回值是什么?

答：返回值可以有很多类型,有String, ModelAndView。ModelAndView类把视图和数据都合并的一起的，但一般用String比较好。

Spring MVC用什么对象从后台向前台传递数据的?

答：通过ModelMap对象,可以在这个对象里面调用put方法,把对象加到里面,前台就可以通过el表达式拿到。

怎么样把ModelMap里面的数据放入Session里面?

答：可以在类上面加上@SessionAttributes注解,里面包含的字符串就是要放入session里面的key。

Spring MVC里面拦截器是怎么写的

有两种写法,一种是实现HandlerInterceptor接口，另外一种是继承适配器类，接着在接口方法当中，实现处理逻辑；然后在Spring MVC的配置文件中配置拦截器即可：

```
<!-- 配置Spring MVC的拦截器 -->
<mvc:interceptors>
    <!-- 配置一个拦截器的Bean就可以了 默认是对所有请求都拦截 -->
    <bean id="myInterceptor" class="com.zwp.action.MyHandlerInterceptor">
</bean>
    <!-- 只针对部分请求拦截 -->
    <mvc:interceptor>
       <mvc:mapping path="/modelMap.do" />
       <bean class="com.zwp.action.MyHandlerInterceptorAdapter" />
    </mvc:interceptor>
</mvc:interceptors>
```

介绍一下 WebApplicationContext

WebApplicationContext 继承了ApplicationContext 并增加了一些WEB应用必备的特有功能，它不同于一般的ApplicationContext ，因为它能处理主题，并找到被关联的servlet。

专题7：Tomcat 专题 部分

Tomcat是什么?

Tomcat 服务器Apache软件基金会项目中的一个核心项目，是一个免费的开放源代码的Web 应用服务器，属于轻量级应用服务器，在中小型系统和并发访问用户不是很多的场合下被普遍使用，是开发和调试JSP 程序的首选。

Tomcat的缺省端口是多少，怎么修改

1. 找到Tomcat目录下的conf文件夹
2. 进入conf文件夹里面找到server.xml文件
3. 打开server.xml文件
4. 在server.xml文件里面找到下列信息
5. 把Connector标签的8080端口改成你想要的端口

```
<Service name="Catalina">
<Connector port="8080" protocol="HTTP/1.1"
               connectionTimeout="20000"
               redirectPort="8443" />
```

tomcat 有哪几种Connector 运行模式(优化)?

下面，我们先大致了解Tomcat Connector的三种运行模式。

- **BIO：同步并阻塞** 一个线程处理一个请求。缺点：并发量高时，线程数较多，浪费资源。Tomcat7或以下，在Linux系统中默认使用这种方式。

配制项：protocol=”HTTP/1.1”

- NIO：同步非阻塞IO

  利用Java的异步IO处理，可以通过少量的线程处理大量的请求，可以复用同一个线程处理多个connection(多路复用)。

  Tomcat8在Linux系统中默认使用这种方式。

  Tomcat7必须修改Connector配置来启动。

  配制项：protocol=”org.apache.coyote.http11.Http11NioProtocol”

  备注：我们常用的Jetty，Mina，ZooKeeper等都是基于java nio实现.

- APR：即Apache Portable Runtime，从操作系统层面解决io阻塞问题。**AIO方式，**异步非阻塞IO**(Java NIO2又叫AIO) 主要与NIO的区别主要是操作系统的底层区别.可以做个比喻:比作快递，NIO就是网购后要自己到官网查下快递是否已经到了(可能是多次)，然后自己去取快递；AIO就是快递员送货上门了(不用关注快递进度)。

  配制项：protocol=”org.apache.coyote.http11.Http11AprProtocol”

  备注：需在本地服务器安装APR库。Tomcat7或Tomcat8在Win7或以上的系统中启动默认使用这种方式。Linux如果安装了apr和native，Tomcat直接启动就支持apr。

Tomcat有几种部署方式?

在Tomcat中部署Web应用的方式主要有如下几种：

1. 利用Tomcat的自动部署。

   把web应用拷贝到webapps目录。Tomcat在启动时会加载目录下的应用，并将编译后的结果放入work目录下。

2. 使用Manager App控制台部署。

   在tomcat主页点击“Manager App” 进入应用管理控制台，可以指定一个web应用的路径或war文件。

3. 修改conf/server.xml文件部署。

   修改conf/server.xml文件，增加Context节点可以部署应用。

4. 增加自定义的Web部署文件。

   在conf/Catalina/localhost/ 路径下增加 xyz.xml文件，内容是Context节点，可以部署应用。

tomcat容器是如何创建servlet类实例？用到了什么原理?

1. 当容器启动时，会读取在webapps目录下所有的web应用中的web.xml文件，然后对 **xml文件进行解析，并读取servlet注册信息**。然后，将每个应用中注册的servlet类都进行加载，并通过 **反射的方式实例化**。（有时候也是在第一次请求时实例化）
2. 在servlet注册时加上1如果为正数，则在一开始就实例化，如果不写或为负数，则第一次请求实例化。

Tomcat工作模式

Tomcat作为servlet容器，有三种工作模式：

- 1、独立的servlet容器，servlet容器是web服务器的一部分；
- 2、进程内的servlet容器，servlet容器是作为web服务器的插件和java容器的实现，web服务器插件在内部地址空间打开一个jvm使得java容器在内部得以运行。反应速度快但伸缩性不足；

- 3、进程外的servlet容器，servlet容器运行于web服务器之外的地址空间，并作为web服务器的插件和java容器实现的结合。反应时间不如进程内但伸缩性和稳定性比进程内优；

进入Tomcat的请求可以根据Tomcat的工作模式分为如下两类：

- Tomcat作为应用程序服务器：请求来自于前端的web服务器，这可能是Apache, IIS, Nginx等；
- Tomcat作为独立服务器：请求来自于web浏览器；

面试时问到Tomcat相关问题的几率并不高，正式因为如此，很多人忽略了对Tomcat相关技能的掌握，下面这一篇文章整理了Tomcat相关的系统架构，介绍了Server、Service、Connector、Container之间的关系，各个模块的功能，可以说把这几个掌握住了，Tomcat相关的面试题你就不会有任何问题了！另外，在面试的时候你还要有意识无意识的往Tomcat这个地方引，就比如说常见的Spring MVC的执行流程，一个URL的完整调用链路，这些相关的题目你是可以往Tomcat处理请求的这个过程去说的！掌握了Tomcat这些技能，面试官一定会佩服你的！

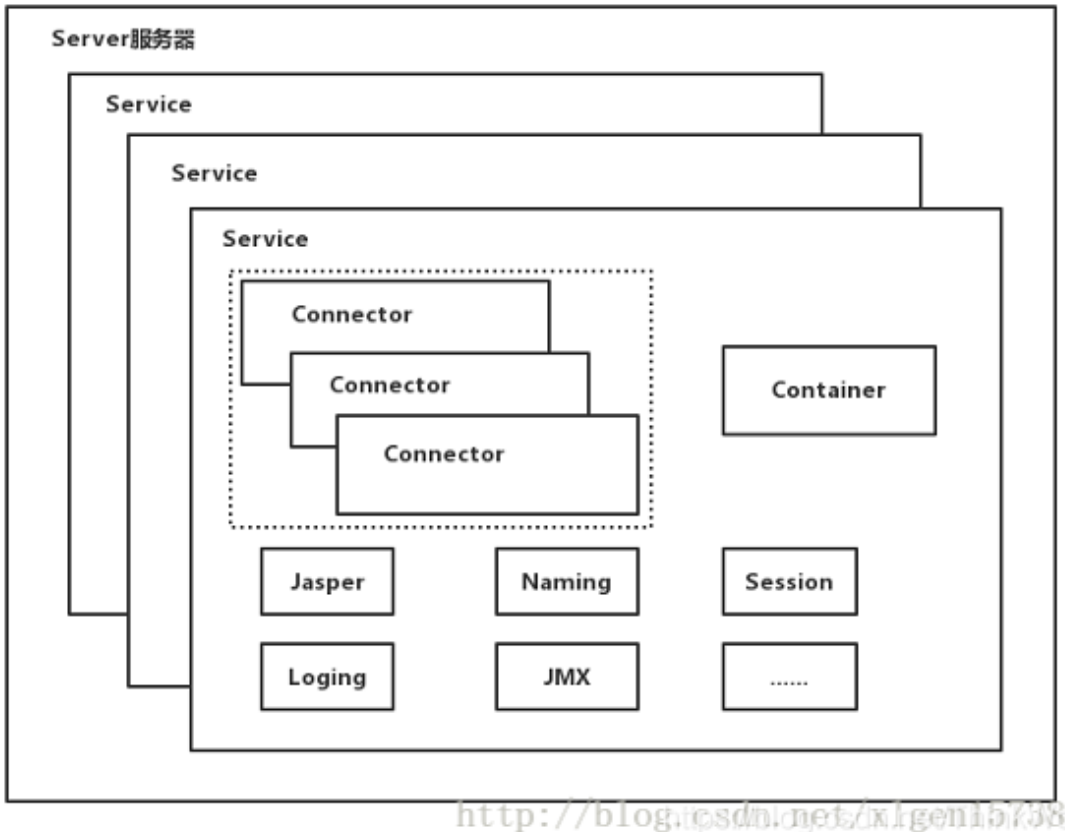
学了本章之后你应该明白的是：

- Server、Service、Connector、Container四大组件之间的关系和联系，以及他们的主要功能点；
- Tomcat执行的整体架构，请求是如何被一步步处理的；
- Engine、Host、Context、Wrapper相关的概念关系；
- Container是如何处理请求的；
- Tomcat用到的相关设计模式；

Tomcat顶层架构

俗话说，站在巨人的肩膀上看世界，一般学习的时候也是先总览一下整体，然后逐个部分个个击破，最后形成思路，了解具体细节，Tomcat的结构很复杂，但是 Tomcat 非常的模块化，找到了 Tomcat 最核心的模块，问题才可以游刃而解，了解了 Tomcat 的整体架构对以后深入了解 Tomcat 来说至关重要！

先上一张Tomcat的顶层结构图（图A），如下：

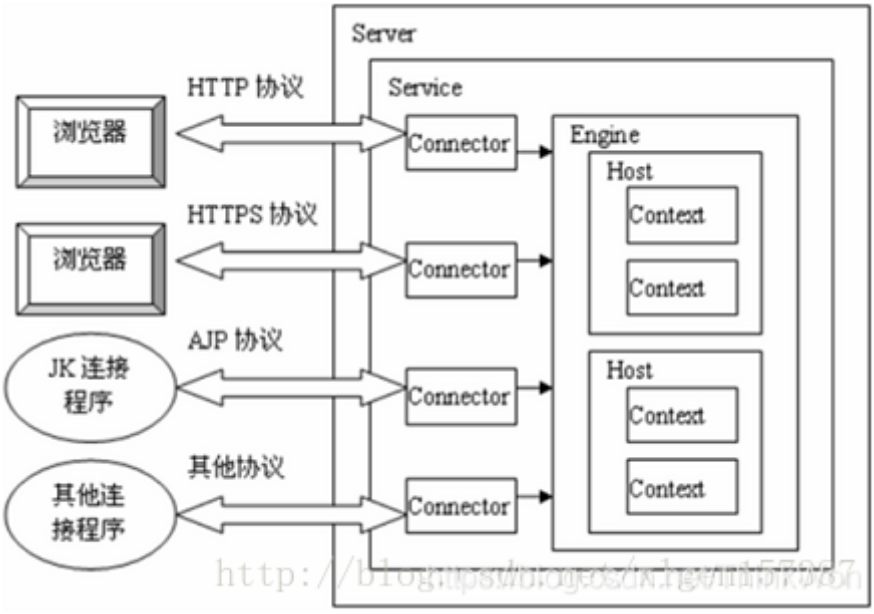


Tomcat中最顶层的容器是Server，代表着整个服务器，从上图中可以看出，一个Server可以包含至少一个Service，即可以包含多个Service，用于具体提供服务。

Service主要包含两个部分：Connector和Container。从上图中可以看出 Tomcat 的心脏就是这两个组件，他们的作用如下：

- Connector用于处理连接相关的事情，并提供Socket与Request请求和Response响应相关的转化;
- Container用于封装和管理Servlet，以及具体处理Request请求;

一个Tomcat中只有一个Server，一个Server可以包含多个Service，一个Service只有一个Container，但是可以有多个Connectors，这是因为一个服务可以有多个连接，如同时提供Http和Https链接，也可以提供向相同协议不同端口的连接，示意图如下（Engine、Host、Context下面会说到）：



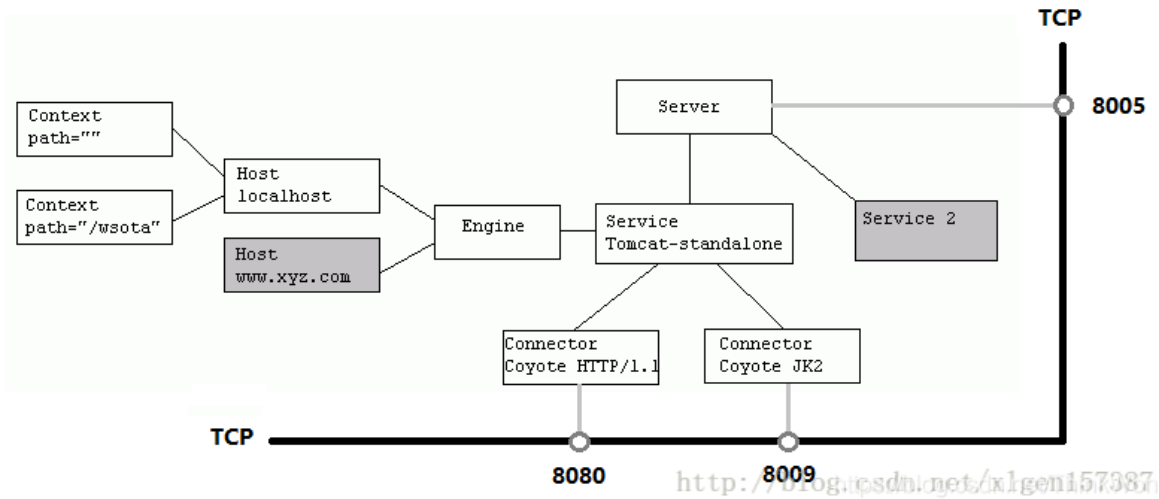
多个 Connector 和一个 Container 就形成了一个 Service，有了 Service 就可以对外提供服务了，但是 Service 还要一个生存的环境，必须要有人能够给她生命、掌握其生死大权，那就非 Server 莫属了！所以整个 Tomcat 的生命周期由 Server 控制。

另外，上述的包含关系或者说是父子关系，都可以在tomcat的conf目录下的server.xml配置文件中看出，下图是删除了注释内容之后的一个完整的server.xml配置文件（Tomcat版本为8.0）

```xml
<?xml version='1.0' encoding='utf-8'?>
<Server port="8005" shutdown="SHUTDOWN">
  <Listener className="org.apache.catalina.startup.VersionLoggerListener" />
  <Listener className="org.apache.catalina.core.AprLifecycleListener" SSLEngine="on" />
  <Listener className="org.apache.catalina.core.JasperListener" />
  <Listener className="org.apache.catalina.core.JreMemoryLeakPreventionListener" />
  <Listener className="org.apache.catalina.mbeans.GlobalResourcesLifecycleListener" />
  <Listener className="org.apache.catalina.core.ThreadLocalLeakPreventionListener" />

  <GlobalNamingResources>
    <Resource name="UserDatabase" auth="Container"
              type="org.apache.catalina.UserDatabase"
              description="User database that can be updated and saved"
              factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
              pathname="conf/tomcat-users.xml" />
  </GlobalNamingResources>

  <Service name="Catalina">
    <Connector port="8080" protocol="HTTP/1.1"
               connectionTimeout="20000"
               redirectPort="8443" />
    <Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />
    <Engine name="Catalina" defaultHost="localhost">
      <Realm className="org.apache.catalina.realm.LockOutRealm">
        <Realm className="org.apache.catalina.realm.UserDatabaseRealm" resourceName="UserDatabase"/>
      </Realm>
      <Host name="localhost"  appBase="webapps" unpackWARs="true" autoDeploy="true">
        <Valve className="org.apache.catalina.valves.AccessLogValve" directory="logs"
               prefix="localhost_access_log." suffix=".txt"
               pattern="%h %l %u %t &amp;quot;%r&amp;quot; %s %b" />
      </Host>
    </Engine>
  </Service>
</Server>
```

详细的配置文件内容可以到Tomcat官网查看：Tomcat配置文件

上边的配置文件，还可以通过下边的一张结构图更清楚的理解：



Server标签设置的端口号为8005，shutdown=”SHUTDOWN” ，表示在8005端口监听“SHUTDOWN”命令，如果接收到了就会关闭Tomcat。一个Server有一个Service，当然还可以进行配置，一个Service有多个Connector，Service左边的内容都属于Container的，Service下边是Connector。

Tomcat顶层架构小结

1. Tomcat中只有一个Server，一个Server可以有多个Service，一个Service可以有多个Connector和一个Container；
2. Server掌管着整个Tomcat的生死大权；
3. Service 是对外提供服务的；
4. Connector用于接受请求并将请求封装成Request和Response来具体处理；
5. Container用于封装和管理Servlet，以及具体处理request请求；

知道了整个Tomcat顶层的分层架构和各个组件之间的关系以及作用，对于绝大多数的开发人员来说Server和Service对我们来说确实很远，而我们开发中绝大部分进行配置的内容是属于Connector和Container的，所以接下来介绍一下Connector和Container。

Connector和Container的微妙关系

由上述内容我们大致可以知道一个请求发送到Tomcat之后，首先经过Service然后会交给我们的Connector，Connector用于接收请求并将接收的请求封装为Request和Response来具体处理，Request和Response封装完之后再交由Container进行处理，Container处理完请求之后再返回给Connector，最后在由Connector通过Socket将处理的结果返回给客户端，这样整个请求的就处理完了！

Connector最底层使用的是Socket来进行连接的，Request和Response是按照HTTP协议来封装的，所以Connector同时需要实现TCP/IP协议和HTTP协议！

Tomcat既然需要处理请求，那么肯定需要先接收到这个请求，接收请求这个东西我们首先就需要看一下Connector！

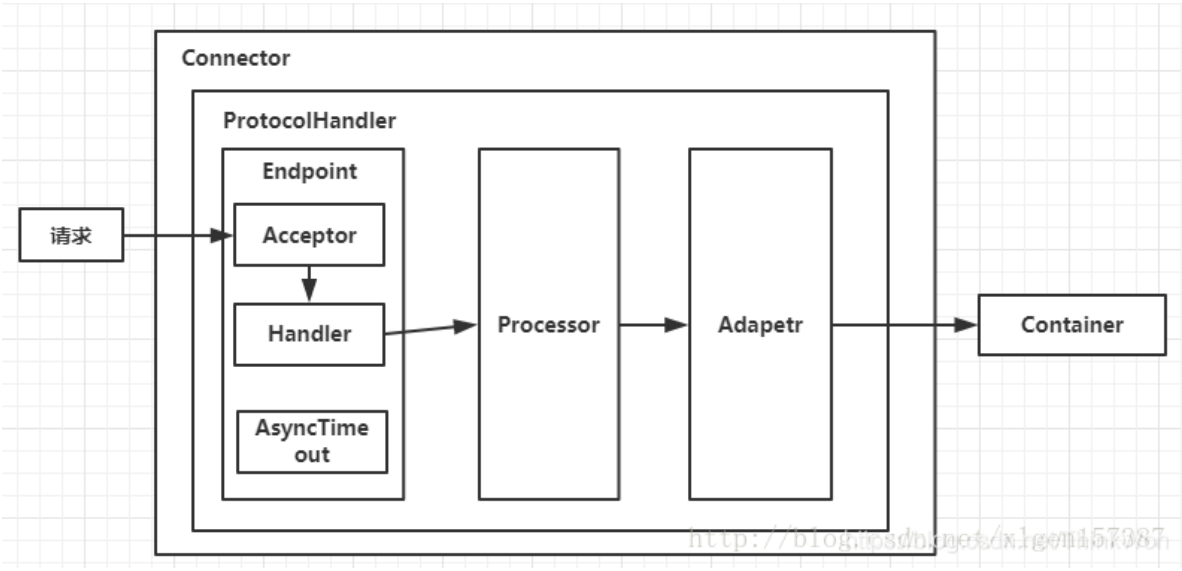
Connector架构分析

Connector用于接受请求并将请求封装成Request和Response，然后交给Container进行处理，Container处理完之后在交给Connector返回给客户端。

因此，我们可以把Connector分为四个方面进行理解：

1. Connector如何接受请求的?
2. 如何将请求封装成Request和Response的?
3. 封装完之后的Request和Response如何交给Container进行处理的?
4. Container处理完之后如何交给Connector并返回给客户端的?

首先看一下Connector的结构图（图B），如下所示：



Connector就是使用ProtocolHandler来处理请求的，不同的ProtocolHandler代表不同的连接类型，比如：Http11Protocol使用的是普通Socket来连接的，Http11NioProtocol使用的是NioSocket来连接的。

其中ProtocolHandler由包含了三个部件：Endpoint、Processor、Adapter。

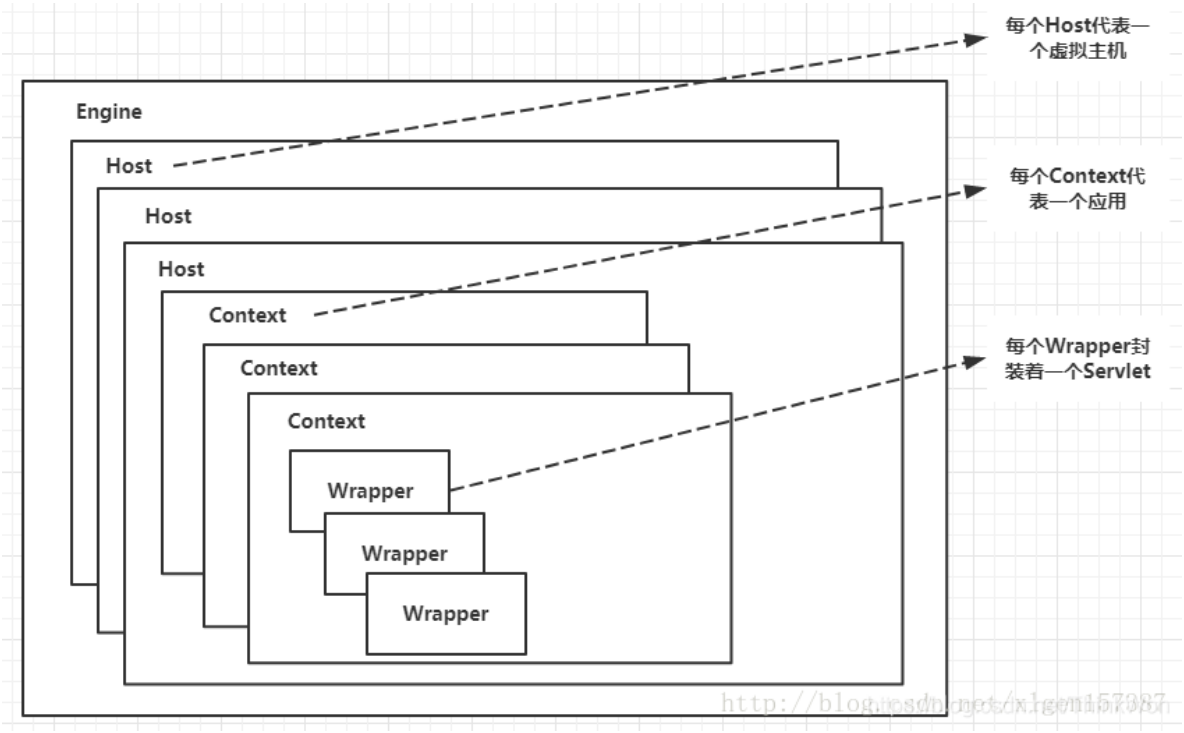
1. Endpoint用来处理底层Socket的网络连接，Processor用于将Endpoint接收到的Socket封装成Request，Adapter用于将Request交给Container进行具体的处理。
2. Endpoint由于是处理底层的Socket网络连接，因此Endpoint是用来实现TCP/IP协议的，而Processor用来实现HTTP协议的，Adapter将请求适配到Servlet容器进行具体的处理。

3. Endpoint的抽象实现AbstractEndpoint里面定义的Acceptor和AsyncTimeout两个内部类和一个Handler接口。Acceptor用于监听请求，AsyncTimeout用于检查异步Request的超时，Handler用于处理接收到的Socket，在内部调用Processor进行处理。

至此，我们应该很轻松的回答1，2，3的问题了，但是4还是不知道，那么我们就来看一下Container是如何进行处理的以及处理完之后是如何将处理完的结果返回给Connector的?

Container架构分析

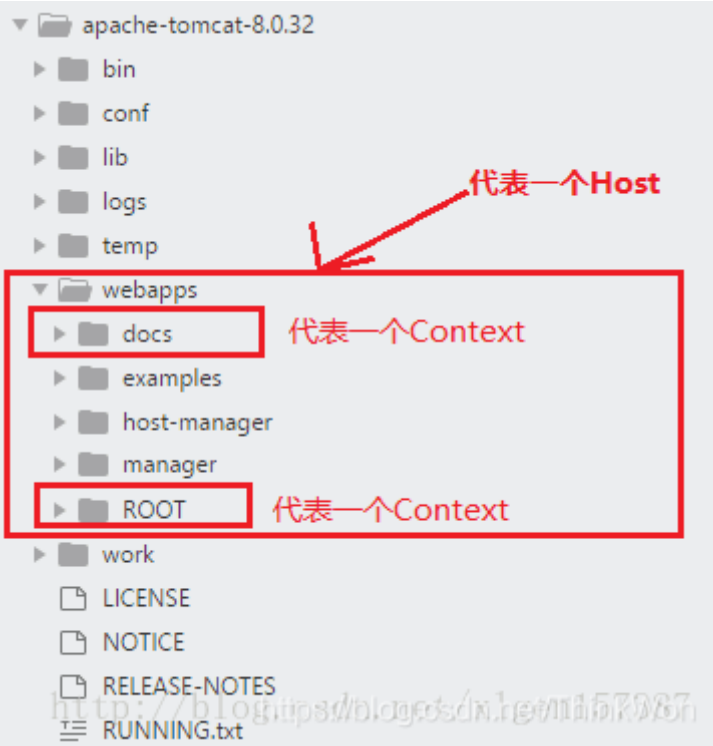
Container用于封装和管理Servlet，以及具体处理Request请求，在Container内部包含了4个子容器，结构图如下（图C）：



4个子容器的作用分别是：

1. Engine：引擎，用来管理多个站点，一个Service最多只能有一个Engine；
2. Host：代表一个站点，也可以叫虚拟主机，通过配置Host就可以添加站点；
3. Context：代表一个应用程序，对应着平时开发的一套程序，或者一个WEB-INF目录以及下面的web.xml文件；
4. Wrapper：每一Wrapper封装着一个Servlet；

下面找一个Tomcat的文件目录对照一下，如下图所示：



Context和Host的区别是Context表示一个应用，我们的Tomcat中默认的配置下webapps下的每一个文件夹目录都是一个Context，其中ROOT目录中存放着主应用，其他目录存放着子应用，而整个webapps就是一个Host站点。

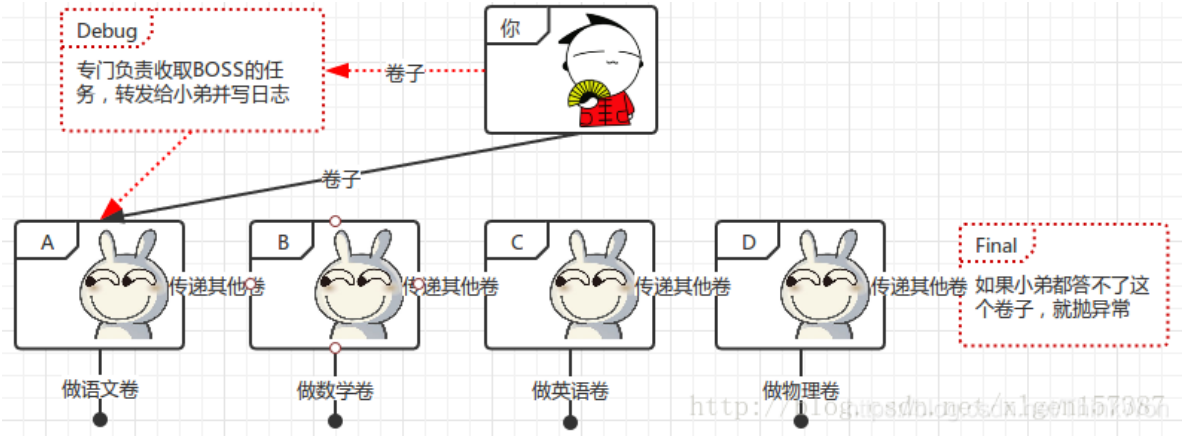
我们访问应用Context的时候，如果是ROOT下的则直接使用域名就可以访问，例如：www.baidu.com，如果是Host（webapps）下的其他应用，则可以使用www.baidu.com/docs进行访问，当然默认指定的根应用（ROOT）是可以进行设定的，只不过Host站点下默认的主应用是ROOT目录下的。

看到这里我们知道Container是什么，但是还是不知道Container是如何进行请求处理的以及处理完之后是如何将处理完的结果返回给Connector的？别急！下边就开始探讨一下Container是如何进行处理的！

Container如何处理请求的

Container处理请求是使用Pipeline-Valve管道来处理的！（Valve是阀门之意）

Pipeline-Valve是**责任链模式**，责任链模式是指在一个请求处理的过程中有很多处理者依次对请求进行处理，每个处理者负责做自己相应的处理，处理完之后将处理后的结果返回，再让下一个处理者继续处理。

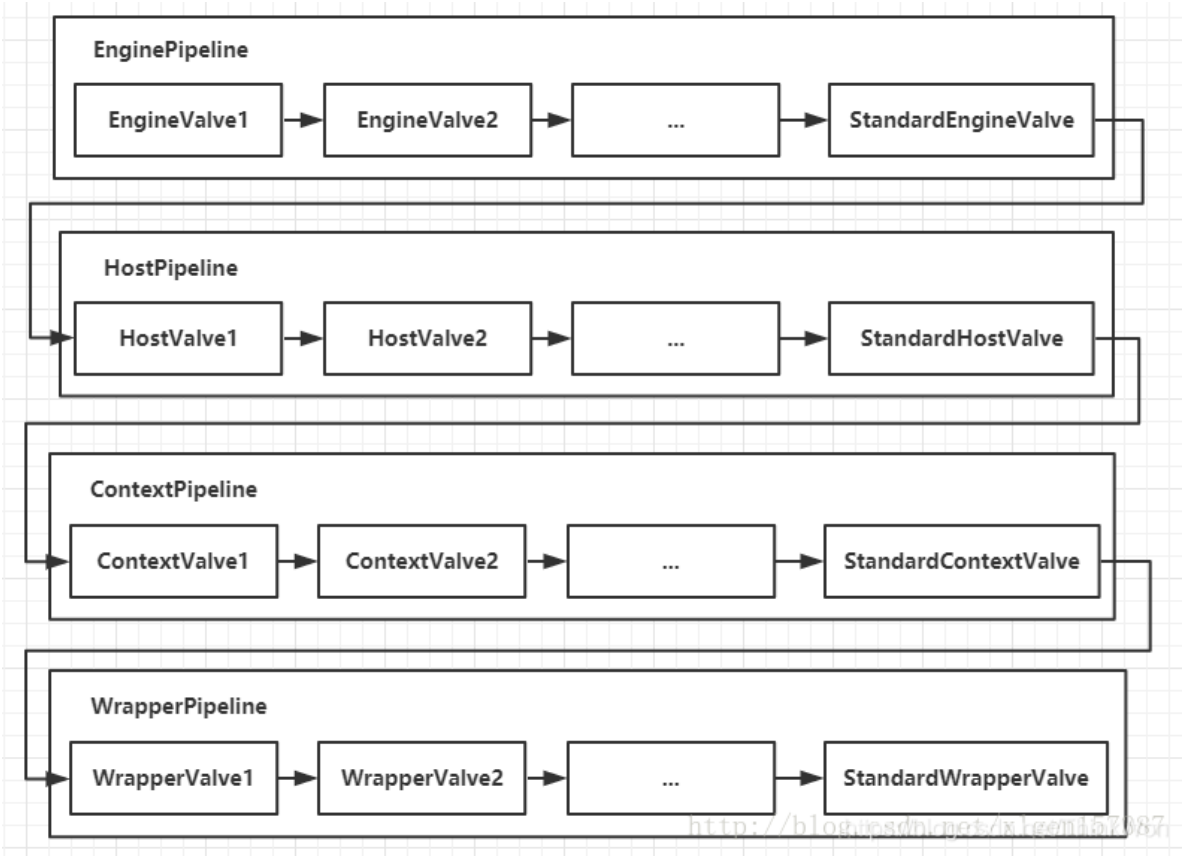


但是！Pipeline-Valve使用的责任链模式和普通的责任链模式有些不同！区别主要有以下两点：

- 每个Pipeline都有特定的Valve，而且是在管道的最后一个执行，这个Valve叫做BaseValve，BaseValve是不可删除的；
- 在上层容器的管道的BaseValve中会调用下层容器的管道。

我们知道Container包含四个子容器，而这四个子容器对应的BaseValve分别在：StandardEngineValve、StandardHostValve、StandardContextValve、StandardWrapperValve。

Pipeline的处理流程图如下（图D）：



- Connector在接收到请求后会首先调用最顶层容器的Pipeline来处理，这里的最顶层容器的Pipeline就是EnginePipeline（Engine的管道）；
- 在Engine的管道中依次会执行EngineValve1、EngineValve2等等，最后会执行StandardEngineValve，在StandardEngineValve中会调用Host管道，然后再依次执行Host的HostValve1、HostValve2等，最后在执行StandardHostValve，然后再依次调用Context的管道和Wrapper的管道，最后执行到StandardWrapperValve。
- 当执行到StandardWrapperValve的时候，会在StandardWrapperValve中创建FilterChain，并调用其doFilter方法来处理请求，这个FilterChain包含着我们配置的与请求相匹配的Filter和Servlet，其doFilter方法会依次调用所有的Filter的doFilter方法和Servlet的service方法，这样请求就得到了处理！
- 当所有的Pipeline-Valve都执行完之后，并且处理完了具体的请求，这个时候就可以将返回的结果交给Connector了，Connector在通过Socket的方式将结果返回给客户端。

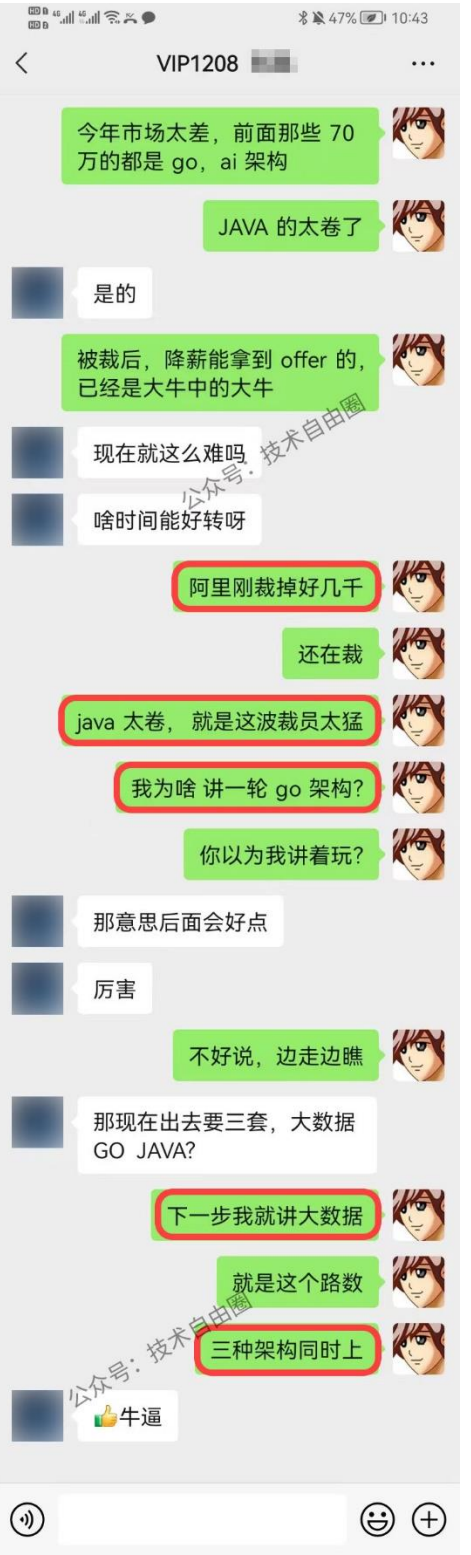
参考文献：

https://www.cnblogs.com/leeego-123/p/12159574.html

https://www.jianshu.com/p/1dec08d290c1

https://blog.csdn.net/weixin_40006977/article/details/112711947

未来职业，如何突围：三栖架构师



成功案例：2年翻3倍，35岁卷王成功转型为架构师

详情：http://topcoder.cloud/forum.php?mod=forumdisplay&fid=43&page=1

最新　最后发表　热门　精华

- 成功案例：[1057号卷王] 3年小伙拿到外企offer，薪酬涨了200%
  卷王1号　超级版主　前天 17:41
- 成功案例：[645号卷王] 4年经验卷王逆袭，被毕业后，反涨24W
  卷王1号　超级版主　2022-9-21
- 成功案例：[878号卷王] 小伙8年经验，年薪60W
  卷王1号　超级版主　2022-8-13
- 年薪70W案例：通过尼恩的指导，小伙伴年薪从40W涨到70W
  卷王1号　超级版主　2022-2-11
- 成功案例：[493号卷王] 5年小伙拿满意offer，就业寒冬季逆涨30%
  卷王1号　超级版主　前天 17:43
- 成功案例：[250号卷王] 就业极寒时代，收offer 涨25%
  卷王1号　超级版主　前天 17:38
- 成功案例：[612号卷王] 就业极寒时代，从外包到自研
  卷王1号　超级版主　前天 17:15
- 成功案例：[913号卷王] 热烈祝贺6年经验卷王，年薪40W
  卷王1号　超级版主　2022-9-21
- 成功案例：[959号卷王] 4年经验卷王，喜获百度、Boss直聘等N个优质offer，最高涨100%
  卷王1号　超级版主　2022-9-21
- 成功案例：[529号卷王] 5年经验卷王喜收2大offer，最高涨5K
  卷王1号　超级版主　2022-9-21
- 成功案例：[811号卷王] 热烈祝贺7年经验卷王，薪酬涨30%
  卷王1号　超级版主　2022-9-21
- 成功案例：[287号卷王] 不惧大寒潮，卷王逆市收4 offer，涨30%，可喜可贺
  卷王1号　超级版主　2022-5-30
- 成功案例：[1002号卷王] 5月份"被毕业"，改简历后，斩获顶级央企Offer，涨薪7000+
  卷王1号　超级版主　2022-7-5

成功案例：[7号卷王] 热烈祝贺小伙伴涨薪120%

卷王1号 超级版主 2022-8-13

成功案例：[134号卷王] 大三小伙卷1年，斩获顶级央企Offer，成功逆袭

卷王1号 超级版主 2022-7-6

成功案例：[1008号卷王] 5年经验卷王收42W offer，月涨8000，可喜可贺

卷王1号 超级版主 2022-5-30

成功案例：[453号卷王] 非全日制 6年卷王喜提3 offer，年薪30W，可喜可贺

卷王1号 超级版主 2022-5-21

成功案例：[924号卷王] 6年卷王喜提4 offer，最高涨薪9000，可喜可贺

卷王1号 超级版主 2022-5-21

成功案例：[15号卷王] 4年卷王入职 微软，涨薪50%，可喜可贺

卷王1号 超级版主 2022-5-12

成功案例：[527号卷王] 4年卷王喜提2 offer，涨薪50%，可喜可贺

卷王1号 超级版主 2022-5-13

成功案例：[788号卷王] 3年卷王喜提优质Offer，涨薪60%

卷王1号 超级版主 2022-5-11

成功案例：热烈祝贺：非全日制卷王，喜提2个心仪offer，面3家过2家

卷王1号 超级版主 2022-4-21

成功案例：[693号卷王] 二线城市6年卷王喜提4大优质Offer，含央企offer，最高薪酬35W

卷王1号 超级版主 2022-4-16

成功案例：[85号卷王] 双非2本小伙，春招大捷，喜提9个offer，最高薪酬近30万

卷王1号 超级版主 2022-4-14

成功案例：[741号卷王] 卷王逆袭！6年小伙从很少面试机会到搞定35K*14薪Offer

卷王1号 超级版主 2022-4-12

成功案例：[642号卷王] 热烈祝贺，6年卷王喜提优质国企offer

卷王1号 超级版主 2022-4-7

成功案例：[796号卷王] 热烈祝贺，36岁卷王喜提52万优质offer

卷王1号 超级版主 2022-3-25

成功案例：[15号卷王] 小伙卷1年，涨薪9K+，喜收ebay等多个优质offer

卷王1号 超级版主 2022-3-24

成功案例：[821号卷王] 小伙狠卷3个月，喜提10多个offer

卷王1号 超级版主 2022-3-21

成功案例：[736号卷王] 3年半经验收22k offer，但是小伙志存高远，冲击25k+

卷王1号 超级版主 2022-3-20

成功案例：热烈祝贺1群小卷王offer拿到手软，甚至拒了阿里offer

卷王1号 超级版主 2022-3-16

简历案例：简历一改，腾讯的邀请就来了！热烈祝贺，小伙收到一大堆面试邀请

卷王1号 超级版主 2022-3-10

成功案例：祝贺我圈两大超级卷王，一个过了阿里HR面，一个过了阿里2面

卷王1号 超级版主 2022-3-10

成功案例：小伙伴php转Java，卷1.5年Java，涨薪50%，喜收多个优质offer

卷王1号 超级版主 2022-3-10

成功案例：4年小伙狠卷半年，拿到 移动、京东 两大顶级offer

尼恩 超级版主 2022-3-5

成功案例：[267号卷王] 助力3年经验卷王，拿到蜂巢的17k x 14薪的offer

卷王1号 超级版主 2022-2-27

成功案例：[143号卷王] 二本院校00后卷神，毕业没到一年跳到字节，年薪45W

卷王1号 超级版主 2022-2-27

成功案例：[494号卷王] 尼恩分布式事务助力卷王拿到 中信银行offer

卷王1号 超级版主 2022-2-27

成功案例：[76号卷王] 2线城市卷王，狠卷1.5年，喜收22K offer

卷王1号 超级版主 2022-2-27

成功案例：[429号卷王] 小伙伴在社群卷5个月，涨8k+

卷王1号 超级版主 2022-2-27

成功案例：[154号卷王] 双非学校毕业卷王，连拿 京东到家&滴滴 两个大厂Offer

卷王1号 超级版主 2022-2-27

成功案例：[232号卷王] 涨薪10K，继续卷向食物链顶端

卷王1号 超级版主 2022-2-27

成功案例：狠卷1年技术，喜收 腾讯、阿里、微软三大Offer，最高年薪56W

卷王1号 超级版主 2022-2-27

成功案例：[449号卷王] 应届毕业卷王喜收 滴滴offer, 年薪33W

卷王1号 超级版主 2022-2-27

成功案例：[551号卷王] 小伙伴学完后，成功进入大厂，并且推荐自己的朋友加VIP学习

卷王1号 超级版主 2022-2-10

成功案例：[214号卷王] 助力2年经验卷王，成功拿到17K月薪

卷王1号 超级版主 2022-2-10

成功案例：[92号卷王] 课程实操助力社群小伙伴喜收 喜马拉雅Offer

卷王1号 超级版主 2022-2-10

成功案例：社群卷王小伙伴成功过了滴滴三面 获滴滴Offer

卷王1号 超级版主 2022-2-10

[612号卷王]滴滴小伙伴，蹲点考察半年，觉得靠谱后加入 疯狂创客圈

卷王1号 超级版主 2022-2-10

成功案例：[732号卷王] 尼恩助力3年经验卷王收获 京东offer, 年薪35W

卷王1号 超级版主 2022-2-27

成功案例：[558号卷王] 2年经验卷王，喜收 网易和阿里子公司两个优质offer

卷王1号 超级版主 2022-2-27

成功案例：[569号卷王] 双非应届生卷王, 喜收字节跳动实习offer

卷王1号 超级版主 2022-2-25

成功案例：[420号卷王] 狠卷1年，卷王涨薪80%，涨薪12000元!

卷王1号 超级版主 2022-2-25

成功案例：[76号卷王] 通过尼恩1年半的指导，专科学历小伙伴从0.8K涨到22K

卷王1号 超级版主 2022-2-10

硬核推荐：尼恩Java硬核架构班

详情：https://www.cnblogs.com/crazymakercircle/p/9904544.html

架构班（社群 VIP）的起源：

最初的视频，主要是给读者加餐。很多的读者，需要一些高质量的实操、理论视频，所以，我就围绕书，和底层，做了几个实操、理论视频，然后效果还不错，后面就做成迭代模式了。

架构班（社群 VIP）的功能：

提供高质量实操项目整刀真枪的架构指导、快速提升大家的:

- 开发水平
- 设计水平
- 架构水平

弥补业务中 CRUD 开发短板，帮助大家尽早脱离具备 3 高能力，掌握：

- 高性能
- 高并发
- 高可用

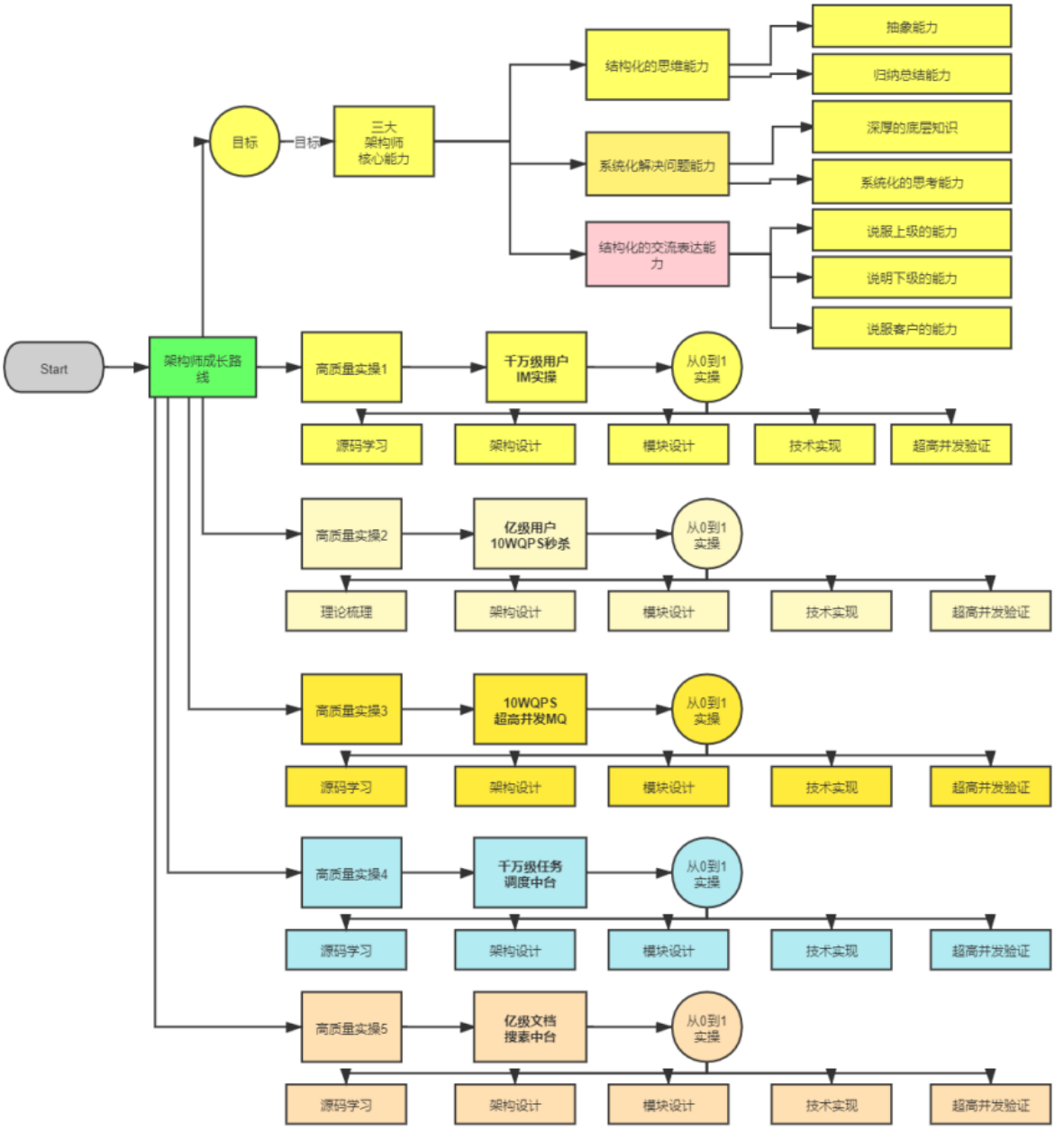
作为一个高质量的架构师成长、人脉社群，把所有的卷王聚焦起来，一起卷：

- 卷高并发实操
- 卷底层原理
- 卷架构理论、架构哲学
- 最终成为顶级架构师，实现人生理想，走向人生巅峰

架构班（社群 VIP）的目的：

- 高质量的实操，大大提升简历的含金量，吸引力，增强面试的召唤率
- 为大家提供九阳真经、葵花宝典，快速提升水平
- 进大厂、拿高薪
- 一路陪伴，提供助学视频和指导，辅导大家成为架构师
- 自学为主，和其他卷王一起，卷高并发实操，卷底层原理、卷大厂面试题，争取狠卷 3 月成高手，狠卷 3 年成为顶级架构师

N 个超高并发实操项目：简历压轴、个顶个精彩



【样章】第 17 章：横扫全网 Rocketmq 视频第 2 部曲：工业级 rocketmq 高可用（HA）底层原理和实操

工业级 rocketmq 高可用底层原理，包含：消息消费、同步消息、异步消息、单向消息等不同消息的底层原理和源码实现；消息队列非常底层的主从复制、高可用、同步刷盘、异步刷盘等底层原理。

工业级 rocketmq 高可用底层原理和搭建实操，包含：高可用集群的搭建。

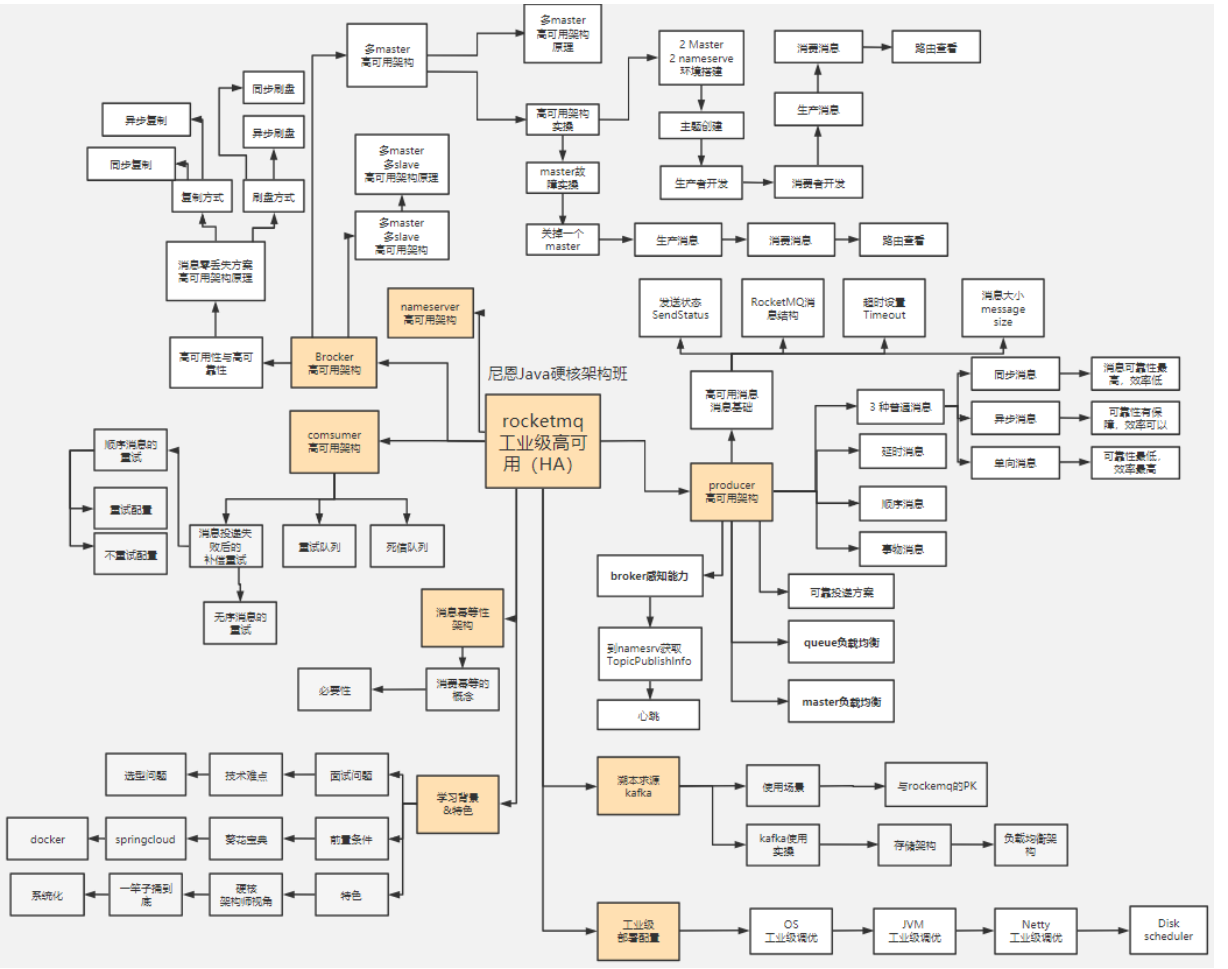
解决以下难题：

1、技术难题：RocketMQ 如何最大限度的保证消息不丢失的呢？RocketMQ 消息如何做到高可靠投递？

2、技术难题：基于消息的分布式事务，核心原理不理解

3、选型难题： kafka or rocketmq ，该娶谁？

下图链接：https://www.processon.com/view/6178e8ae0e3e7416bde9da19



简历优化后的成功涨薪案例（VIP含免费简历优化）


6年小伙收60W年薪
一月速提3大offer
4.24号改简历
5.27号报喜
VIP1239 6年
4月24日 晚上19:10
预期的岗位：60W
预期的岗位：go 和 java 的后端开
4月24日 晚上19:56
秘诀：
简历指导+ 狠狠卷
4月24日 晚上20:50
6年经验 1239
年薪 60W.
21.7 KB
微信电脑版
前几天改 ai，今天改 go
谢谢，我根据这个模式，再整理一下我简历，到时候老师再帮我把把关
ok
4月27日 下午14:37
老师，再帮我看看
准备开始投简历了
简历-6 年后端开发.pdf
210.1 KB
PDF
VIP1239 6年
嗯嗯，我先对着简历准备些东西，然后再开始投吧，反正现在刚刚好五一放假
上午8:01
来还愿啥，3周斩获3个offer，准备入职了
上午8:05
您太牛啦
简历优化后，面试机会太多了，拿完3个offer后，还有许多公司在流程中的都拒绝了
这几天大厂动作不断，联想，阿里都在裁员，您太牛啦
还是老师你强，offer中也达到了预期60w
非常不错
现在都上岸有offer就行，薪酬还能达到预期，已经超级牛掰啦
很多人，连一个面试电话都没有，崩溃的一塌糊涂
抖音上到处是这种
主要是简历优化后，感觉如有神助，每天基本3个面试，除了字节一面没过，其它都通过了
恭喜您
谢谢老师
后续有啥问题，可以找我支撑哈
好嘞，学习圈一直在，要持续提高自己
好的，撸起袖子加油卷，搞技术前途无限好
被裁后转架构，逆涨50%
8年小伙喜提年薪75W
4.16号改简历
5.6号报喜
VIP1236
最近面试了几个一轮游
捞了太多人上岸了
都是你这号
捞我
助力我 一个月时间
绞杀 下钻 打破瓶颈
咱们开始不？
好
4月16日 下午15:04
预期岗位：高级开发、架构
4月16日 下午15:12
预期薪酬：60W
-8 年高级服务端-0404 - 副本(2).
30.2 KB
微信电脑版
4月16日 下午16:19
秘诀：
简历指导+ 狠狠卷
4月16日 下午19:21
通话时长 03:02:25
辛苦老师
VIP1236
上午11:08
尼架，我决定要去上海了。
拿了几个 offer 了？
两个
上海这个是架构师对吧？还有一个呢？
还有个广州高级 Java，待遇 40w 左右，老板比较喜欢我，开了很多绿灯，薪资可以再加，但我还是想闯闯，昨天拒绝了。
年包多少呢？
就之前说的
那都快 80 个 W 了
我 argue 了下，他们控制内部薪酬平衡有点难办到 80，但已经是标出来的上限了。
都快是 高级 java 的 两倍
你撤回了一条消息
75 w 也非常多了，在现在的环境下
除开税，差不了多少，主要是这个方向的潜在价值
关键对你来说，这个是一个 成长机会
是的，寒意还是有的
您是我的贵人
是呢 有个外接大脑就是爽
好好卷，先祝贺您，拿到年薪 75W +
再预祝您，2 年之后，年薪 200W+
谢谢


9年 小伙伴拿到
年薪90W offer
9月11日改简历
11月29日晒offer
秘诀:
简历指导+ 狠狠卷
这个是给你微调的
省略号的地方，需要你再补充一点
上面你留着这个就行
Java 开发 9年-修改.docx
34.1 KB
薪资高 稳
主要的工作是啥?
提升了自己的实力，就不用怕
易所 关于数字货 的 易
po 也有 打盹的时候，该裁员，照样一个不少
年包比 po 多 19w
这么多
po 估计有 70 万
那你不有 90 个 W?
po 给我 68
就是吗


小伙8年经验
年薪60w
7月12日改简历
8月10日晒offer
秘诀:
改简历 + 狠狠卷
明天晚上哈
好哒
恩哥，今晚还改简历吗?
今晚还在外边应酬，估计回去比较晚
要不，咱们延迟到明天，如何
明天白天也行
好的，白天吧，答应别人明天给他们简历了。
OK
那就上午 11 点左右哈
好的
涨幅多少呀?
或者，额度多少呀
之前 36*15，现在这个 39*15
今年行情不太好，还有一些 offer 基本都是平薪，或降薪的。
OK
这个马上来
刚在指导简历
哈哈
稍等哈
辛苦啊恩哥
这次找工作，您的 起到效果了。
我这次复习基本看的都是咱们课程的 。


6年小伙伴
年薪40w
9月6日改简历
9月21日晒offer
秘诀:
简历指导+ 狠狠卷
Java - 6年.docx
24.7 KB
恩哥，简历我改好了，您再帮我看一下
Java - 6年(1).docx
24.5 KB
恩哥，看第二个吧
给你前面调整了一下
涨这多
今年这行情，也算可以了
总包多少呀，让我也陶醉一下
大概 40w 吧
谢谢恩哥的指导和鼓励
是在深圳
深圳的行情尤其难
能有面试电话就不错啦
是的
太牛啦
哈哈哈哈，恩哥的鼓励指导也很重要


5年小伙喜提3个offer
年薪 35个W
5月22日改简历
11月29日晒offer
我简历还没写出来 不然可以给老师您看看
算法方面，可以找点 刷刷，当作复习
项目的话，再好好改造一下
好的 谢谢老师
java 开发-本科统招-2022-5 年经验.docx
36.0 KB
OK
我排好号，回头找你哈
秘诀:
简历指导+ 狠狠卷
尼恩老师晚上好，汇报下最近的 offer 情况。最近面试收到了三个 offer，两个是平薪，一个是跨境电商公司的 offer，涨幅暂时不到 20%。通过这次面试也让我知道自己距离高级开发还有一点点距离，还要再多看才能突破。
最后结合自己的情况，先选择去跨境电商的公司再提升下
之前是 20k*13.5，跨境电商这个是说 23k*（14-15）
恭喜恭喜
就业寒冬，能提到 3 个 offer，已经很 ，已经是高手了
很多小伙伴，面试电话一个都接不到，简历海投 7000 份，只收到 3 次面试机会，没有一个机会拿到最终 offer
您这个年薪，算下来也有 35W 了吧
年终部分还要确认下，大部分人听说只有两个月
这个时间点，拿到这个水平，挺不错的啦
持续加油卷哈
嗯嗯！看看明年自己有没有能力冲击高开
把你我视频都吃透
辛苦老师再帮忙指导下哈


1.5年小伙搞定15K offer
就业寒冬涨100%
5月7日改简历
11月21日晒offer
秘诀：
简历指导+狠狠卷
多谢啦
你好，好多公司都觉得我的简历太简单了，我感觉是不是没法优化了
简历.pdf
397.4 KB
好的
晚上指导你改
推送中台需要加不
我还没做啊，准备八股文暂时没时间搞这些，入职了会开始搞
OK
还有其他的项目吗
一个项目，太少啦
那种练习项目也行
他那个不止两倍吧，我才是两倍，原先7k，现在15k
那也很牛
都翻倍了，都是牛人
正常就30%
慢慢来
下一步可以瞄准25k
其实我工作一年
那你就更牛逼拉
一年经验，搞定15k offer，舍你其谁


卷王逆袭成功案例
6年小伙从很少面试机会到
搞定35K*14薪
3月5日改简历
4月11日拿offer
面试邀请少
小伙很苦恼
理想很丰满
目标35K
一个月拿到了
理想的offer
面试法宝
rocketmq四部曲

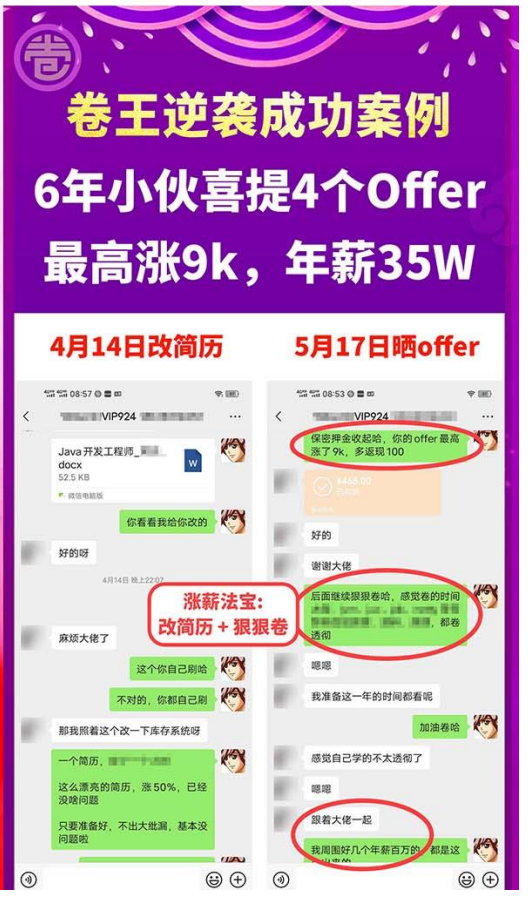

6年 经验小伙伴
喜收25K offer
3月12日改简历
12月1日晒offer
秘诀：
简历指导+狠狠卷
老师
咱们以这个项目为模板改哈
嗯好
项目有多少人，你带领多少人？
你工作几年了？
6年了
项目有多少人，你带领多少人？
语音交流一下
好
你方便吗
OK
不知道往哪个方向发展
改简历挪窝，20 涨到 25 k，高级开发，P7 带的后端团队
任务重，道路远，并不是没有方向


7年经验卷王
薪酬涨30%
7月11日改简历
9月1日晒offer
秘诀：
改简历 + 狠狠卷
咱们开始修改哈
要语音还是打字也行，还没到家
解决过最复杂的难题是啥？说 3 个哈
看看，有没有问题？
没问题
看看，有没有问题？
只能挪窝
下次你喊足一点哦
只要本事好，拿 offer 比较容易的
喊了，但是我流水太差了。。。我这家是补贴，银行不认这玩意，只能在上家的基础上加到 30% 封顶（特别优秀除外，我只是不错。。。）
入职了，最终并不能大幅涨薪，还是旧的太低，只涨了 30%
恭喜一下
现在不比往年
能涨 30% 是大牛
拿到 offer 都算大牛
现在很多人投几百发，连个电话都没有
你已经很


4年经验卷王逆袭
被毕业后，反涨24W
7月改简历
8月30日晒offer
秘诀:
改简历 + 狠狠卷
这就是你的简历
差得太多啦
ok
总共写了四个项目，最近的一年的还没补充上
你是在职，还是离职呢?
离职
原因大概是啥?
这个地方要这样写
项目被终止
方便语音沟通不
ok
是的 经你这么指导，非常重要
老哥 我八月十号开始找工作，今天已经入职了
现金基本持平，股票 +24W
总计涨了多少呢
能涨24W
股票这个吧只能到手了才算
也不错啦
很多小伙伴，面试机会都没有
感谢老哥的指导，继续跟你卷技术
继续狠狠卷哈，马上就技术自由拉

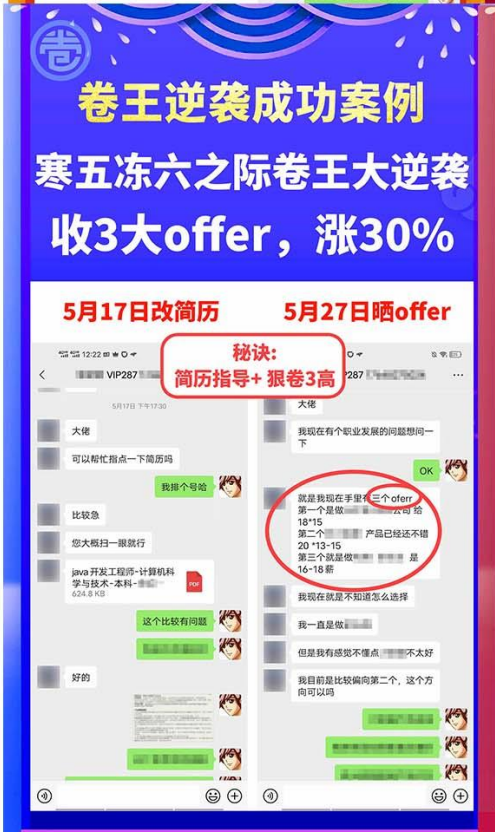

小伙5月份"被毕业"，改简历后
斩获顶级央企Offer
涨薪7000+
5月29日改简历
7月5日晒offer
秘诀:
简历指导+ 狠卷3高
快速看书，就要不求甚解，把目录和 场景大概一下，然后重点的地方，用到的地方，再去回炉
尼恩 我被“ 毕业 ”了
这周末或下周我你改一下简历
毕业了没有关系
ok，发我吧
it行业，跳来跳去，太频繁啦
嗯，其实有点心理准备
简历
不太会写简历
尼恩 我拿到的offer了
涨20%，不还价的
看起来不差钱呀
超过了8000没
平均算下来
7000多
好的
有啥面试的心得吗
可以分享给其他小伙伴的


卷王逆袭成功案例
武汉6年喜收4个优质offer
最高的年薪35W
2月9日改简历
4月15日晒offer
尼恩老师，新年好!
能帮忙修改下简历吗?
金三银四准备跳了
可以的
java 开发-6年-简历-.pdf
拜托了，恩师。希望能拿 25k 回来给你报喜
好，我加一下
还有吗?
面试法宝:
改简历 + 实操
恩哥，决赛圈 offer 了
截图是以我目前认知能写出来的评分了，麻烦帮我参考下
选择大于努力，恩师助我上岸
这么多 offer，我看看哈
都是恩哥指点有方，本来还有个新能源汽车的，35w给拒了，主要太远了
跟着恩哥卷的时间太短了，目前实力也只能到这儿了
这里边有个大数据的，感觉也不错

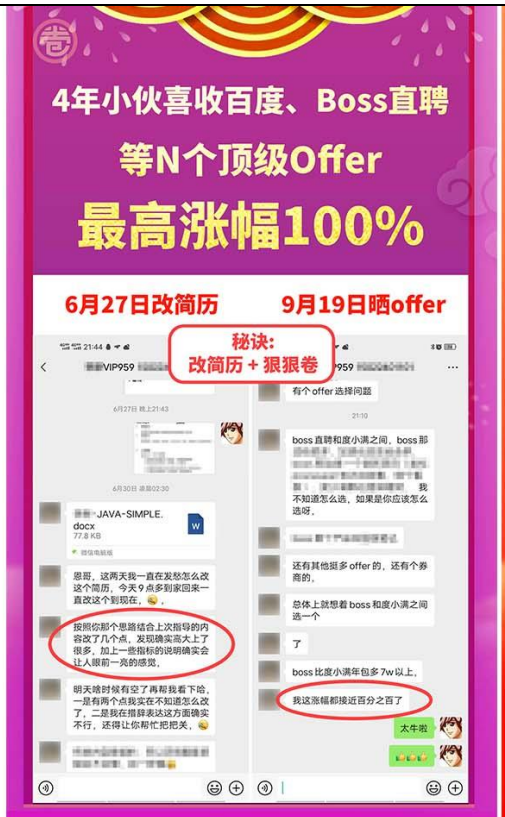

卷王逆袭成功案例
6年小伙喜提4个Offer
最高涨9k，年薪35W
4月14日改简历
5月17日晒offer
Java 开发工程师_.docx
你看看我给你改的
好的呀
麻烦大佬了
这个你自己刷哈
不对的，你都自己刷
那我照着这个改一下库存系统呀
一个简历，
这么漂亮的简历，涨 50%，已经没啥问题
只要准备好，不出大纰漏，基本没问题啦
涨薪法宝:
改简历 + 狠狠卷
保密押金收起哈，你的 offer 最高涨了 9k，多返现 100
好的
谢谢大佬
后面继续狠狠卷哈，感觉卷的时间
透彻
嗯嗯
我准备这一年的时间都看呢
加油卷哈
感觉自己学的不太透彻了
嗯嗯
跟着大佬一起
我周围好几个年薪百万的，都是这


卷王逆袭成功案例
5年经验小伙收2个offer
最高涨薪8k，年薪42W
5月9日改简历
5月30日晒offer
秘诀:
简历指导+ 狠卷3高
以此为样
大家狠狠卷
打造最卷IT社群


卷王逆袭成功案例
非全日制 6年经验卷王
喜提3个Offer，年包30W
5月9日改简历
5月18日晒offer
面试法宝:
改简历 + 狠狠卷


卷王逆袭成功案例
寒五冻六之际卷王大逆袭
收3大offer，涨30%
5月17日改简历
5月27日晒offer
秘诀:
简历指导+ 狠卷3高


卷王逆袭成功案例
4年卷王入职微软，涨50%
3月7日改简历
5月12日晒offer
涨薪法宝:
改简历 + 狠狠卷


4年小伙喜收百度、Boss直聘
等N个顶级Offer
最高涨幅100%
6月27日改简历
9月19日晒offer
秘诀:
改简历＋狠狠卷


卷王逆袭成功案例
4年卷王入收2个offer，涨50%
3月23日改简历
5月12日晒offer
涨薪法宝:
改简历＋狠狠卷


小伙大三暑期很焦虑
跟着尼恩卷一年
校招斩获顶级央企Offer
去年5月19日加入VIP群
今年7月5日晒offer
秘诀:
狠狠卷 书+视频


小伙高中学历
薪酬涨120%
5月6日改简历
7月22日晒offer
秘诀:
改简历＋狠狠卷


卷王逆袭成功案例
非全日制卷王 面试3家
收2个offer 涨薪30%
4月13日改简历
4月21日晒offer
面试法宝：
改简历 + 面试题


5年卷王喜收2大Offer
最高涨5K
5月19日改简历
9月13日晒offer
秘诀：
改简历 + 狠狠卷


卷王逆袭成功案例
3年经验卷王，涨60%
4月16日改简历
5月11日晒offer
涨薪法宝：
改简历 + 狠狠卷


卷王逆袭成功案例
双非二本小伙春招大翻身
喜提9大offer
2月22日改简历
4月13日晒offer
面试法宝：
改简历 + IM实操
9大offer
最高年薪30万

修改简历找尼恩（资深简历优化专家）

- 如果面试表达不好，尼恩会提供 简历优化指导
- 如果项目没有亮点，尼恩会提供 项目亮点指导
- 如果面试表达不好，尼恩会提供 面试表达指导

作为 40 岁老架构师，尼恩长期承担技术面试官的角色：

- **从业以来，“阅历”无数，对简历有着点石成金、改头换面、脱胎换骨**的指导能力。
- **尼恩指导过刚刚就业的小白，也指导过 P8 级的老专家，都指导他们上岸。**

如何联系尼恩。尼恩微信，请参考下面的地址：

语雀：https://www.yuque.com/crazymakercircle/gkkw8s/khigna
码云：https://gitee.com/crazymaker/SimpleCrayIM/blob/master/疯狂创客圈总目录.md