

**Spring, springCloud, springBoot和springMVC的区别**

spring mvc < spring < springboot

1、Spring

Spring是一个开源容器框架，可以接管web层，业务层，dao层，持久层的组件，并且可以配置各种bean,和维护bean与bean之间的关系。其核心就是控制反转(IOC),和面向切面(AOP),简单的说就是一个分层的轻量级开源框架。

2、SpringMVC

Spring MVC属于SpringFrameWork的后续产品，已经融合在Spring Web Flow里面。SpringMVC是一种web层mvc框架，用于替代servlet（处理|响应请求，获取表单参数，表单校验等。SpringMVC是一个MVC的开源框架，SpringMVC=struts2+spring，springMVC就相当于是Struts2加上Spring的整合。

3、SpringBoot

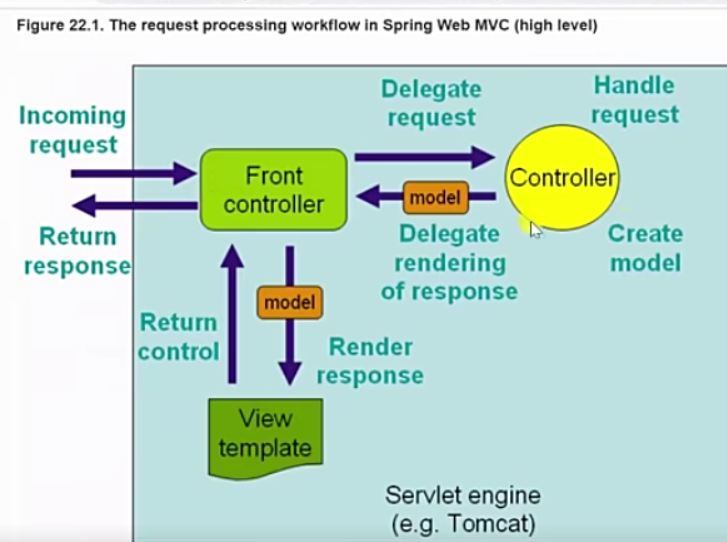
Springboot是一个微服务框架，延续了spring框架的核心思想IOC和AOP，简化了应用的开发和部署。Spring Boot是为了简化Spring应用的创建、运行、调试、部署等而出现的，使用它可以做到专注于Spring应用的开发，而无需过多关注XML的配置。提供了一堆依赖打包，并已经按照使用习惯解决了依赖问题--->习惯大于约定。

1. Spring cloud大部分的功能插件都是基于springBoot去实现的，springCloud关注于全局的微服务整合和管理，将多个springBoot单体微服务进行整合以及管理；  springCloud依赖于springBoot开发，而springBoot可以独立开发

**DispatcherServlet**

基于Servlet WebApplicationContext容器管理controller，viewResolver，handleMapping(映射路径的组件，把路径和controller相匹配)

Front controller是DispatcherServlet



**MVC框架**

MVC指model view controller=模型-视图-控制器，是一种软件设计典范，用一种业务逻辑、数据、界面显示分离的方法组织代码，把业务逻辑聚集到一个组件里，在改进和个性化定制界面和用户交互的时候，不用重写业务逻辑。主要是分三层的一种架构，主要采用封装（分层）的思想来分割页面和数据库访问的框架，降低了两者的耦合度，同时通过Controller的控制，也使系统更加的灵活，使我们的代码更健壮，复用性更强

1.model是用于处理数据逻辑的部分。通常模型对象负责在数据库中存取数据。

2.view是应用程序中处理数据显示的部分。通常视图是依据数据模型创建的。

3.controller是应用程序中处理数据交互的部分。通常控制器负责从视图读取数据，控制用户输入，并向模型发送。

**优点**

（1）多个视图共享一个模型，大大提高代码的可重用性。

（2）三个模块相互独立，改变其中一个不会影响其他两，所以依据这种设计模式能构建良好的松耦合性的组件。

（3）控制器提高了应用程序的灵活性和可控制性。控制器可以连接不同的模型和视图满足用户需求，(这样控制器可以为构造应用程序提高强有力的手段)

规避了JSP与Servlet各自的短板，Servlet只负责业务逻辑而不会通过out.append()动态生成HTML代码；JSP中也不会充斥着大量的业务代码。这大大提高了代码的可读性和可维护性。

**缺点**

（1）增加了系统结构和实现的复杂性。

对于简单页面，严格遵循mvc，使模型、视图与控制器分离，会增加结构的复杂性，并可能产生过多的更新操作，降低运行效率。

（2）视图与控制器过于紧密的连接。

视图与控制器是相互分离，但确实联系紧密的部件，视图没有控制器的存在，其应用是很有限的，反之亦然，这样就妨碍了他们的独立重用。

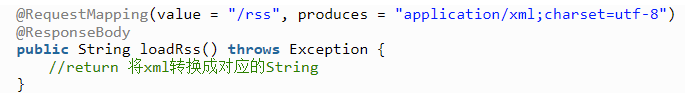
（3）视图对模型数据的低效率访问。

依据模型操作接口的不同，视图可能需要多次调用才能获得足够的显示数据。对未变化数据的不必要的频繁访问，也将损害操作性能。

**控制返回Json或XML：**

**@RequestMapping 的produces 属性指定**

produces = "application/xml" 或者 produces = "application/json"



一般来说访问一个处理器，然后会返回一个视图，或者跳转到另外的处理器，SpringMVC一个机制，允许控制器返回的数据不经过正常的视图处理流程，而是直接将返回的数据写入响应体中（response body）

**传统的Servlet**：用Java写的服务端程序，有独立于平台和协议的特性，可以响应任何类型的请求，但大多情况下只用来扩展基于HTTP协议的Web服务器，主要功能是交互式地浏览和生成数据，生成动态的Web内容

Servlet是一个接口或者是实现Servlet接口的类叫Servlet

**传统开发模式Servlet(不用IOC)的弊端**：分成3层架构：把视图控制，业务逻辑和数据库分离出来单独形成一个类—易于复用和维护。代码分成controller, service, DAO：一个DAO被多个Service对象使用，一个Service对象被多个Controller用---有逻辑复用，没有做到资源复用。多个controller创建多个重复的service，多个service创建多个重复的DAO，可以用单例模式，但每个类都是单例就会重复很多代码，而且变化的代价太大，一个service变了要改很多个controller。本质原因：组件的调用方参与了组件的创建和配置工作。所以用容器来创建和配置，比如Tomcat是Servlet的容器，它创建和配置好了Servlet. 我们用IOC容器来管理组件

**SpringBoot启动流程**

启动流程主要分为三个部分：

第一部分进行SpringApplication的初始化模块，配置一些基本的环境变量、资源、构造器、监听器，

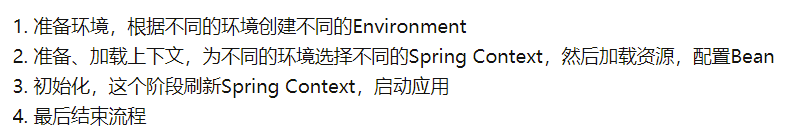
第二部分实现了应用具体的启动方案，包括启动流程的监听模块、加载配置环境模块、及核心的创建上下文环境模块，

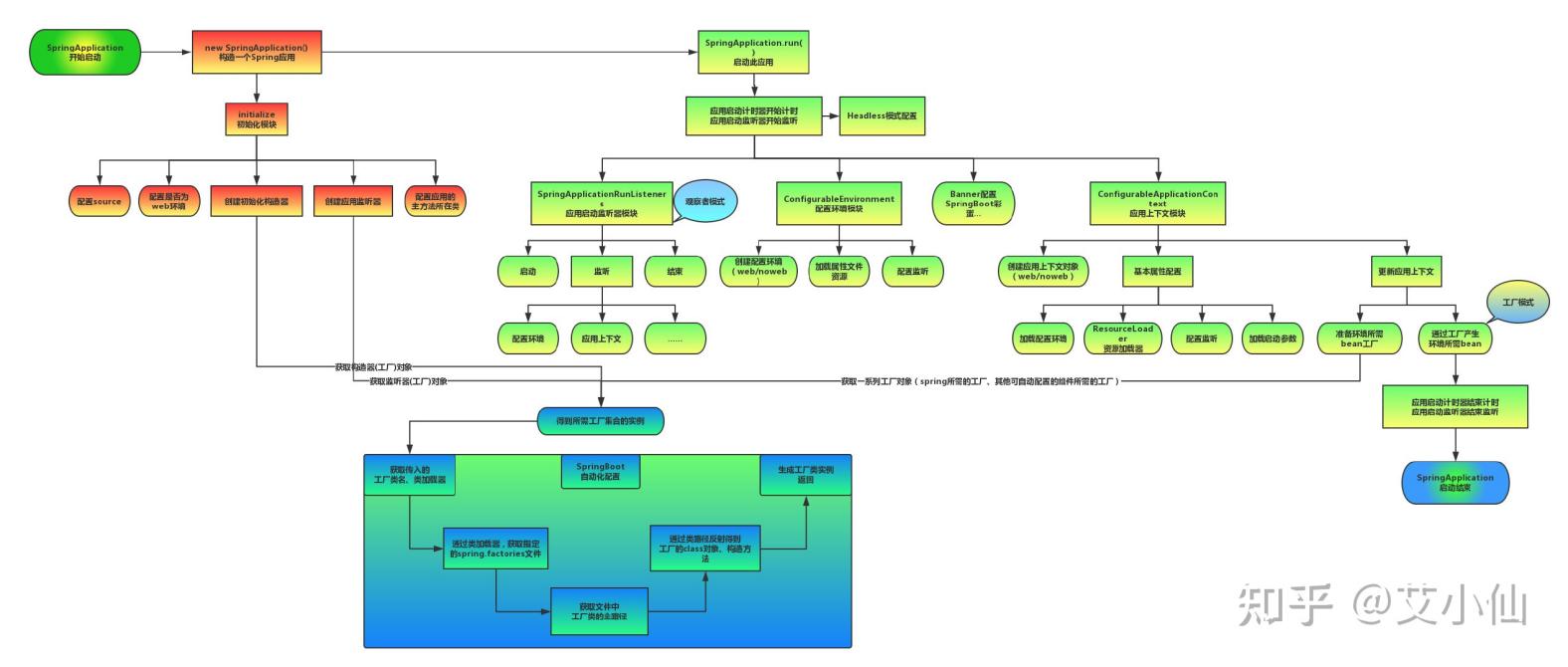
第三部分是自动化配置模块，该模块作为springboot自动配置核心，在后面的分析中会详细讨论。在下面的启动程序中我们会串联起结构中的主要功能。

当@EnableConfigurationProperties注解应用到你的@Configuration时， 任何被@ConfigurationProperties注解的beans将自动被Environment属性配置。 这种风格的配置特别适合与SpringApplication的外部YAML配置进行配合使用

@EnableConfigurationProperties 相当于把使用 @ConfigurationProperties 的类进行了一次注入到spring

**SpringBoot启动流程**





**IOC(控制反转)**：是面向对象编程的一个思想，指的是对象的创建和配置和生命周期的控制权从调用方移给Spring容器。把对象给IOC管理，调用方不用再负责对象之间复杂的依赖关系，达到解耦的目的，Spring里由BeanFactory和ApplicationContext两者IOC容器，通过他们来实现对bean的管理

**实现：**靠**依赖注入**。传统情况下，应用程序需要自己维护所创建的依赖对象。而依赖注入则是将对象的创建和维护交给IoC容器。调用方只要声明依赖项，所需要的Bean就自动配置完了，不用知道创建对象的过程

**依赖注入的四种方式**

 1）构造方法注入：将被调用对象作为调用对象构造函数的参数传入

 2）setter方法注入：将被调用对象作为调用对象的属性，使用set方法注入

 3）接口注入：依赖类必须要实现指定的接口，然后实现该接口中的一个函数，该函数就是用于依赖注入。该函数的参数就是要注入的对象。

4）注解方式注入：Spring和注入相关的常见注解有Autowired、Resource、Service、Controller、Repository、Component等

**AOP(面向切面编程)**

是一个编程范式，是OOP的延续，目的是提高代码模块性，基于动态代理实现，横切剖解开封装的对象内部，然后把那些与业务无关但是被多个类业务模块共用的逻辑封装到一个可重用模块，也就是切面来减少系统的重复代码，降低模块间的耦合度，增加之后的可操作性和可维护性。也就是把核心关注点和横切关注点(缓存，懒加载，调式，同步，权限认证，事务，日志，异常处理)分开，让核心模块只关心核心功能代码。

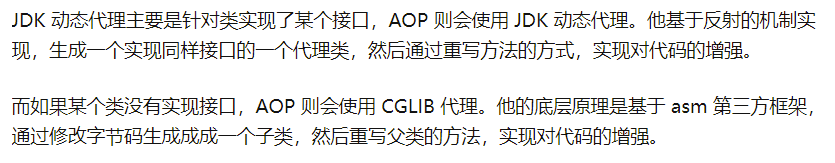
**实现**：1.XML配置文件 2.注解(before,after,around)3.自定义注解

底层是由目标类的代理类实现的。AOP代理其实是由AOP框架动态生成的一个对象(可以作为目标对象使用)。里面包含了目标对象的全部方法，但代理的方法个目标对象方法不一样：它的切入点添加了增强处理并回调了目标对象的方法。

**动态代理：**在java程序运⾏过程由JVM生成代理类的class信息，这个信息生成后是直接在内存的，没有写到磁盘保存，灾后通过反射实例化代理类对象

**静态代理：**由程序员创建代理类或特定⼯具自动生成源代码再对其编译。在程序运行前代理类的.class⽂件就已经存在了

**JDK动态代理和CGLIB代理**

****

**注解：**

**·@PointCut：**公共切入点表达式**·@JoinPoint**： 作为函数的参数传入切面方法，可以得到目标方法的相关信息**·@Aspect：**指定切面类

**·@EnableAspectJAutoProxy**： 开启基于注解的AOP模式

**连接点(Joinpoint)**：在程序执行过程中某个特定的点，比如某方法调用或者处理异常的时候。AOP中连接点表示一个方法的执行。就是加入切点的那个点

**切面(Aspect)**：拦截器类，其中会定义切点以及通知，包含横切代码和连接点信息，横切关注点的抽象。3种类型：一般切面(advisor)切点切面(PointcutAdvisor)引介切面(IntroductionAdvisor)

**切入点（Pointcut）：**指定一个通知将被引发的一系列连接点的集合,匹配连接点的断言。通知和一个切入点表达式关联，并在满足这个切入点的连接点上运行（例如，当执行某个特定名称的方法时）。Spring默认使用AspectJ切入点语法匹配切入点表达式和连接点

**通知（Advice）：**在切面的某个特定连接点上执行的动作。包括“around”、“before”和“after”等不同类型通知。Spring用拦截器做通知模型，并维护一个以连接点为中心的拦截器链。

**引入(Introduction)**：给一个类型声明额外的方法或属性(也被称为连接类型声明)。Spring允许引入新的接口(以及一个对应的实现)到任何被代理的对象。比如可以使用引入来使一个bean实现IsModified接口来简化缓存机制

**织入(Weaving)**：将切面应用到目标对象来创建新的代理对象的过程。这些可以在编译时，类加载时和运行时(Spring)完成。

**增强(Advice)**：是织入到目标类连接点上的一段程序代码。Spring使用增强类定义横切逻辑，同时由于Spring只支持方法连接点，增强还包括在方法上的哪一点加入横切代码的方位信息，所以增强包括横切逻辑和部分连接点的信息。

**引介(Introduction)**：是一种特殊的增强，为类添加一些属性和方法。

**OOP的局限性：OOP**有重复代码的时候就封装出来复用但只能复用核心业务逻辑不能复用辅助逻辑(日志记录，性能统计，事务管理), OOP是自上而下的，但辅助逻辑是横向贯穿所有方法的。这些辅助逻辑是切面，可以用代理模式，但切面逻辑的难点不是不修改原有业务，而是对所有业务生效。要增强一个业务类就要新建一个代理类，切面多了就会导致代理类嵌套。AOP可以让我们不修改原代码，就能让切面逻辑在所有业务逻辑里生效。



**IOC和AOP好处：**

IOC解决：1.创建重复对象，造成资源浪费

2.更换实现类要改很多地方

3.创建和配置组件工作繁杂，而且给组件的调用方会带来很多不便

AOP解决：切面逻辑编写繁琐，有多少个业务方法就要编几次

**框架编写流程**：1.导包 2.写配置 3.测试

**Spring注解**

1.[@Component](http://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/stereotype/Component.html" \t "https://www.cnblogs.com/zoe-java/p/_blank)：标注最普通的组件

2.[@Controller](http://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/stereotype/Controller.html" \t "https://www.cnblogs.com/zoe-java/p/_blank)：标注控制层（spring-mvc的注解）（如：\*Controller）

3.[@Service](http://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/stereotype/Service.html" \t "https://www.cnblogs.com/zoe-java/p/_blank)：标注业务层(如：\*Service)

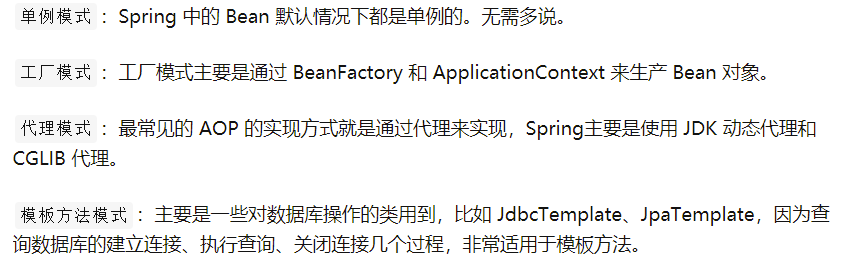
4.[@Repository](http://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/stereotype/Repository.html" \t "https://www.cnblogs.com/zoe-java/p/_blank)：标注持久层（如：\*Dao）

5.@bean 默认单例模式(只有一个实例，处理多次请求的时候在[Spring](http://mp.weixin.qq.com/s?__biz=MzI3ODcxMzQzMw==&mid=2247486678&idx=1&sn=2a5e38e67c3d267d6c58d963adb24ccc&chksm=eb5389e0dc2400f6f6d2e0eaded591fcfc32575ca227e2cebcf39eec13ff8071a649e494f7d8&scene=21" \l "wechat_redirect)容器里只实例化出一个bean，后续的请求都公用这个对象，这个对象会保存在一个[map](http://mp.weixin.qq.com/s?__biz=MzI3ODcxMzQzMw==&mid=2247487842&idx=1&sn=9974be8f5d9b0ba8ffe3724a887e9613&chksm=eb539454dc241d42258faccf3f0d98a97a508e9b5cde9b6af318b0365ed49173a8b2c997b6d4&scene=21" \l "wechat_redirect)里)

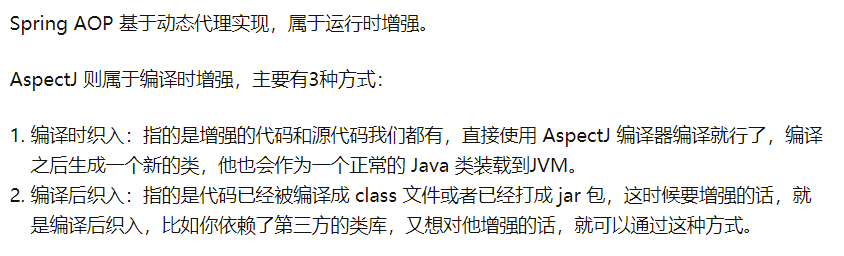
**好处**：单例的bean只有第一次创建新的bean 后面都会复用该bean，所以不会频繁创建对象，减少JVM的GC。而且可以快速获取bean，因为单例的获取bean操作除了第一次生成之外其余的都是从缓存里获取的所以很快

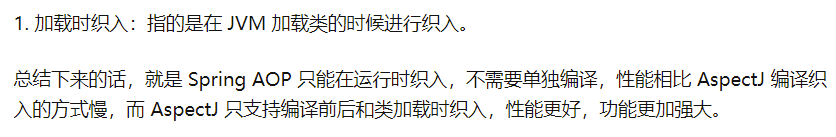
**坏处**：在并发的环境下**不能线程安全**

**设计模式**

****

**Spring AOP和AspectJ AOP**

****

****

**BeanFactory和FactoryBean**

**区别**：BeanFactory是个Factory，也就是IOC容器或对象工厂，是ApplicationContext的父类。

FactoryBean是个Bean。所有的Bean都是由BeanFactory(也就是IOC容器)来进行管理的。但FactoryBean不是简单的Bean，而是一个能生产或者修饰对象生成的工厂Bean, 它的实现和设计模式里工厂模式和修饰器模式类似

bean：1.bean是对象，一个或者多个不限定2.bean由IoC管理 3.应用程序由一个个bean构成



**BeanFactory和ApplicationContext**

BeanFactory是生产和管理bean的容器，ApplicationContext是他的子接口。

ApplicationContext读取配置文件来创建对象。BeanFactory在使用的时候就创建对象

**ApplicationContext接口的实现类**

ClassPathXmlApplicationContext：从类的根路径下加载配置文件，推荐使用这种

FileSystemXmlApplicationContext：从磁盘路径上加载配置文件，配置文件可以在磁盘的任意位置

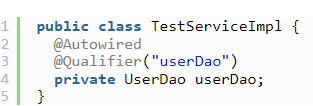
**Bean的生命周期**



**@Autowired和@Resource**

@**Resource**和@Autowired都是做bean的注入时使用，其实@Resource并不是Spring的注解，它的包是javax.annotation.Resource，需要导入，但是Spring支持该注解的注入。

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



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



@Resource装配顺序：

①如果同时指定了name和type，则从Spring上下文中找到唯一匹配的bean进行装配，找不到则抛出异常。

②如果指定了name，则从上下文中查找名称（id）匹配的bean进行装配，找不到则抛出异常。

③如果指定了type，则从上下文中找到类似匹配的唯一bean进行装配，找不到或是找到多个，都会抛出异常。

④如果既没有指定name，又没有指定type，则自动按照byName方式进行装配；如果没有匹配，则回退为一个原始类型进行匹配，如果匹配则自动装配。

**@Component  VS  @Bean**

**相同点**：都是注册bean到Spring容器中

**不同点**：

**@Component** 和 它的子类型（@Controller, @Service and @Repository）注释在类上。告诉Spring，我是一个bean，通过类路径扫描自动检测并注入到Spring容器中。

**@Bean**不能注释在类上，只能用于在配置类中显式声明单个bean。意思就是，我要获取这个bean的时候，spring要按照这种方式去获取这个bean。默认情况下@Bean注释的方法名作为对象的名字，也可以用name属性定义对象的名字。

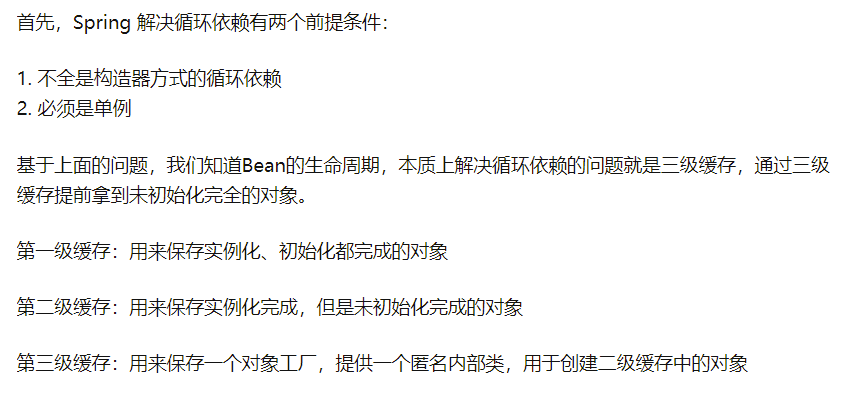
**循环依赖**

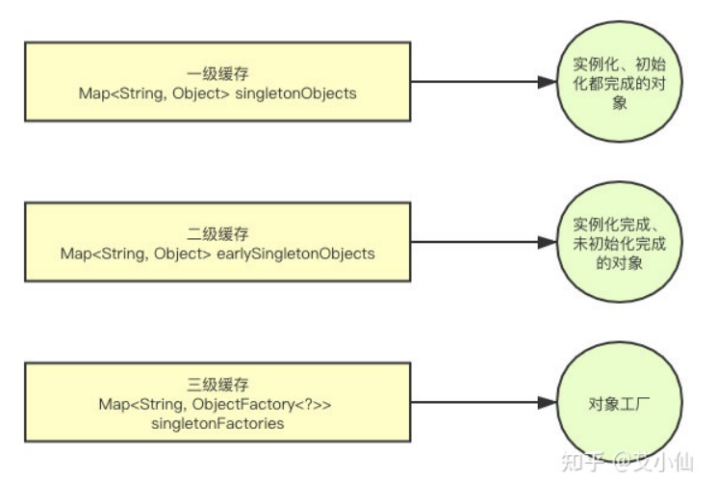
spring 对于bean 的实例化和初始化是分开的，初始化是为bean 对象属性赋值的是时候，如果目前初始化的对象A，其中有一个属性对象B，那么这里就会先去加载注册对象B，但如果B中又有属性对象A，但是A目前还在初始化过程，就会形成了循环依赖

**简单描述就是**：A 对象加载B对象的时候，B 对象反过来需要加载A 对象。甚至可以再加多个中间对象，比如加一个C 对象，那么这就是A 对象加载B 对象的时候，B 对象需要加载C 对象，而C 对象又需要加载A 对象。只要形成加载循环，就会存在循环依赖的问题。

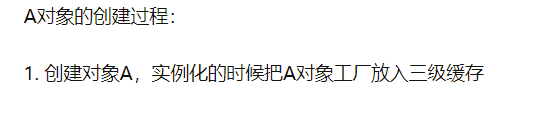
**解决**：1.要把实例化和初始化流程分开，所以用构造器的话就没法分离

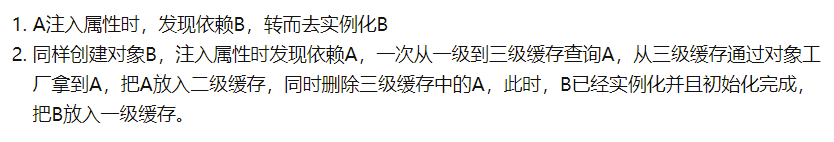
2.利用一、二、三级缓存去解决。为什么不用一个或者两个缓存解决呢，因为一个解决不了（一级缓存是单例池，也就说这个缓存是不可避免的，只用单例池缓存是无法解决循环依赖的），两个无法做到最优

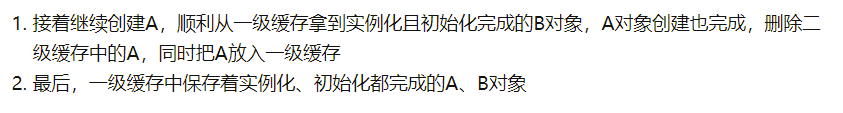


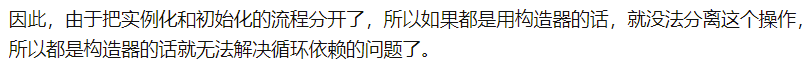


**具体过程例子：**



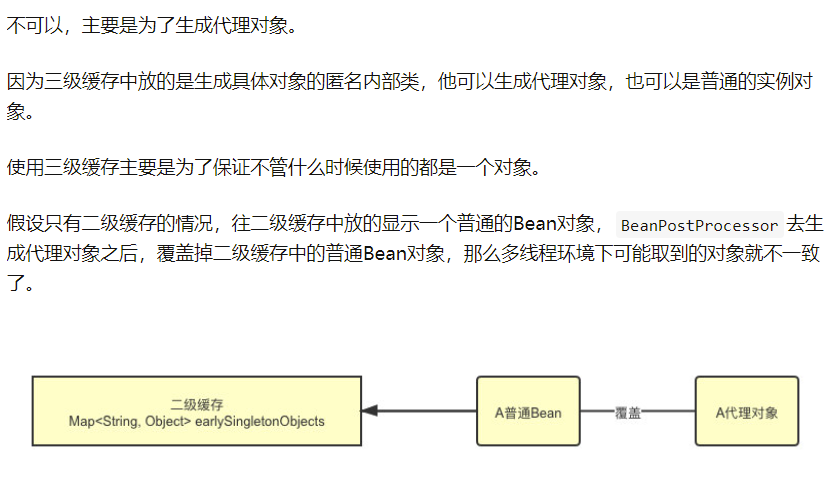




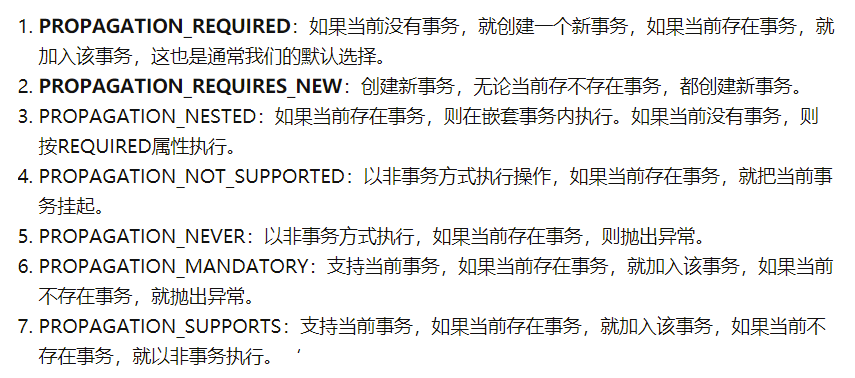


**为什么要三级缓存，二级不行吗**

如果只存在两个缓存也能解决问题，但是会非常的复杂，要判断对象是否是完整对象，还要判断代理对象重复加载之类的，三个缓存同时存在才是最优解



**事务传播机制**



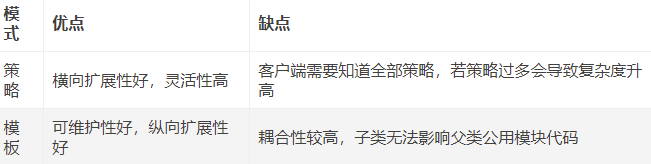
**模板模式 vs 策略模式**

**模板模式**：由抽象类来定义一个算法，在算法实现的不同步骤上抽象方法由子类继承并提供具体实现，常见的就是不同步骤提供doXXX抽象方法留给子类实现。模板模式一般有两部分组成，即抽象模板和具体模板。

**策略模式：**以接口形式提供抽象接口。由具体实现类提供不同算法。策略模式一般由3部分组成：1.一个Context持有所有策略实现类引用，提供给客户端运行

2.一个策略接口提供 3.具体的策略实现类

**区别**：模板模式一般只针对一套算法，注重对同一个算法的不同细节进行抽象提供不同的实现。而策略模式注重多套算法多套实现，在算法中间不应该有交集，因此算法和算法之间一般不会有冗余代码！因为不同算法只间的实现一般不同很相近。所以策略模式的关注点更广，模板模式的关注点更深。而且两种模式可以一起使用，即具体某个策略下可以通过模板减少不同步骤的冗余代码



**MyBatis--基于java的持久层框架**

核心组件：1.sqlSessionFactory：创建sqlSession的工厂类

2.sqlSession：核心组件，用于向数据库执行SQL

3.主配置文件：XML配置文件，对底层进行配置

4.Mapper接口：就是DAO接口

5.Mapper映射器：用于写SQL +将SQL和实体类映射的组件 用XML, 注解都可

前三个不用写

**Mybatis**: 支持定制化 SQL、存储过程和对象关系映射(ORM—就是把数据库表和实体类以及实体类的属性对应起来，让我们可以操作实体类就实现操作数据库表), 内部封装了jdbc(java数据库连接)细节，让开发者只要关注sql语句本身，不用花精力去处理加载驱动、创建连接、创建statement等繁杂的过程，主要设计目的就是让我们对执行SQL语句时对输入输出的数据管理更加方便，能方便地写出SQL和方便地获取SQL的执行结果。

**具体过程**：mybatis通过xml或注解的方式将要执行的各种statement配置起来，并通过java对象和statement中sql的动态参数进行映射生成最终执行的sql语句，最后由mybatis框架执行sql并将结果映射为java对象并返回

**分三层**：  
1、API接口层：提供给外部使用的接口API，开发人员通过这些本地API来操纵数据库。接口层一接收到调用请求就会调用数据处理层来完成具体的数据处理。  
2、数据处理层：负责具体的SQL查找、SQL解析、SQL执行和执行结果映射处理等。它主要的目的是根据调用的请求完成一次数据库操作。  
3、基础支撑层：负责最基础的功能支撑，包括连接管理、事务管理、配置加载和缓存处理，这些都是共用的东西，将他们抽取出来作为最基础的组件。为上层的数据处理层提供最基础的支撑

自定义mybastis：

Mybatis用代理dao的方式实现增删改查：1.创建代理对象，在代理对象中调用selectList

步骤：1.读配置文件2.创建sqlsessionFactory工厂(mybatis用了创建者模式，传in就行，创建工厂细节被隐藏)3.使用工厂生产sqlsession对象(用了工厂模式—解耦，降低类之间的依赖关系，因为让工厂去生产对象) 4.使用sqlsession创建DAO接口的代理对象(用了代理模式—可以不修改源码的基础上对已有方法增强) 5.使用代理对象执行方法

用1和5就行，但有234更灵活

SelectList方法执行—要提供连接信息+映射信息(1.SQL语句2.封装结果的实体类全限定类名—封装成一个对象):

1. 根据配置文件创建connection对象 2.获取预处理对象(要SQL语句) 3.执行查询 4.遍历结果集用于封装，把每个结果加到一个list里(因为实体类的属性和表的列名一样，所以可以把列名看成是实体类的属性名称，就可以用反射的方式来根据名称获取每个属性，并把值赋进去) 5.返回list