# Spring/SpringMVC/SpringBoot

## Spirng思想以及体系结构：

### 定义：

Java应用开发框架，提供一系列优良特性。

### 优良特性：

非侵入式：给于Spring开发的应用中的bean可以不依赖于Spring的API

控制反转：将对象的生命周期管理交给 Spring，而不是我们在我们的类中new

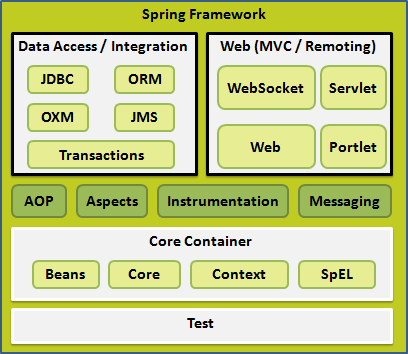
容器：Spring容器负责 管理Bean的生命周期

依赖注入：获取管理的对象不需要使用 getBean然后setBean，而是通过 xml 配置文件注入，或者使用xml或注解自动注入。

AOP：Spring提供面向切面编程，可以方便地实现对程序的 权限拦截，运行监控等功能。**可以在不改变现有代码的情况下增强现有功能（开闭原则）**

声明式事务：基于AOP实现，只需要通过配置就可以完成对事务的管理，无需手动编程。

### 体系结构



spring-core：核心功能，包括IOC和DI

spring-beans：提供BeanFactory等bean容器的相关类

spring-context：提供ApplicationContext相关容器类

spring-expression：提供用于对象操作的表达式语言

Aop+Aspects：提供面向切面编程能力

JDBC：封装JDBC

Transactions：事务相关

# SpringIOC：

## IOC思想：

对于bean对象的控制进行了反转，现在有Spring 的容器对bean的生命周期进行管理，然后通过依赖注入，进行管理。那么问题来了：

容器怎么初始化，如何创建对象？

通过什么创建对象？

创建好之后如何存储对象？

创建好之后如何获取对象？

## 容器：

基于配置文件或者 注解进行bean生命周期管理工作。

### 分类：

BeanFactory(**容器根接口**)：bean容器，常用的是XmlBeanFactory

ApplicationContext：是BeanFactory子接口，比BeanFactory更强大，如果游泳事件管理相关功能。更常用

常见的FileSystemXmlApplicationContext/ClassPathXmlApplicationContext

WebXmlApplicationContext

### 工作流程

1:初始化BeanFactory容器

2:基于xml/注解解析 用户定义的Bean

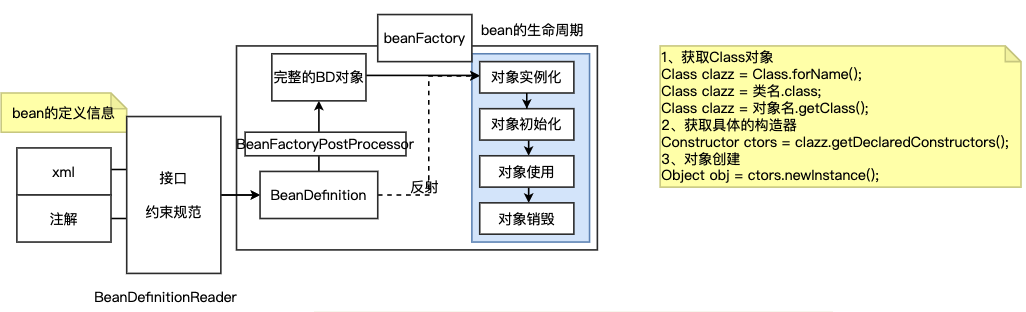
3:把定义的Bean解析为BeanDefination列表

4:执行BeanFactoryPostProccess，做一些BeanFactory初始化完的后置操作（比如BD[]中的把预占位符替换成真实数据）

注意：这个BeanFactoryPostProcess可以自己扩展，后面有例子

5:执行类的实例化/初始化操作（类的生命周期管理，见下面）

6:执行后置处理器的相关操作BeanPostProcess列表（AOP在此实现）



## Bean：

### Bean的定义：

#### 1:使用xml进行

关键元素：

id（或者叫name）：唯一标识一个bean

class：类地址

properties：类实例化参数**（基于setter的依赖注入/手动配置的）**

constructor-arg：构造函数的参数**（基于构造器的依赖注入/手动配置）**

autowiring mode：**依赖注入/自动装配**（byName or byType）

scope：作用域

init-method：在bean所有的必须属性被设置后，回调该方法（也可以使用@PostConstruct）

destroy-method：在包含该bean 的容器被销毁时，回调该方法（也可以使用@preDestory）

lazy-initialization mode：懒加载模式，如果是，那么非启动时创建类，而是使用时创建

**注意**：自动装配也可以使用@Autowrid/@Required/@Qulitailfer/@Primary注解，代替autowiring mode

#### 2:使用Java方式定义

@Configuration

@Bean（name/scopoe/lazy/initMethod/destoryMethod）

### Bean的作用域:

Singleton:在spring IOC容器中只存在一个实例，**默认方式**

Prototype：在每次调用getBean时候，都返回一个新的实例（**单例的原型**）

Request：同一个Http请求创建一个Bean

Session：同一个session请求创建一个Bean

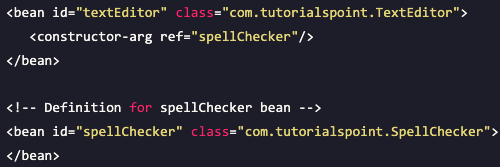
Global-session：。。。。

注意：request/session/global-session都需要处于WebApplicationContext容器模式下

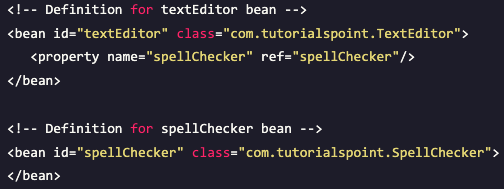
## 依赖注入：

### 手动配置：

1:给于构造器



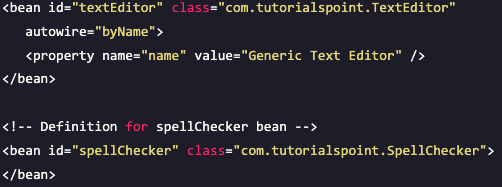
2:给于setX函数



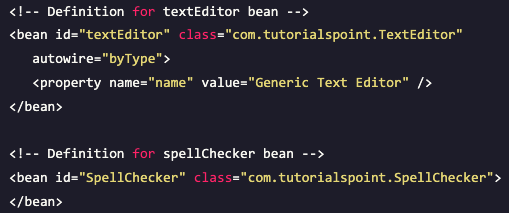
### 自动装配：

#### xml：

1:基于byName



2:基于byType



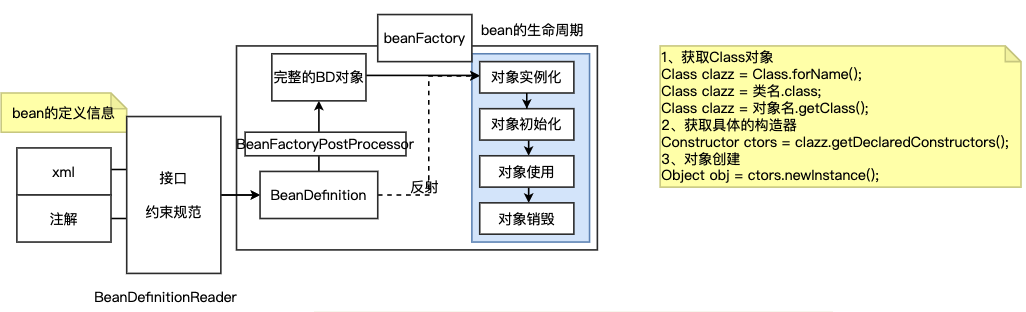
#### 注解：

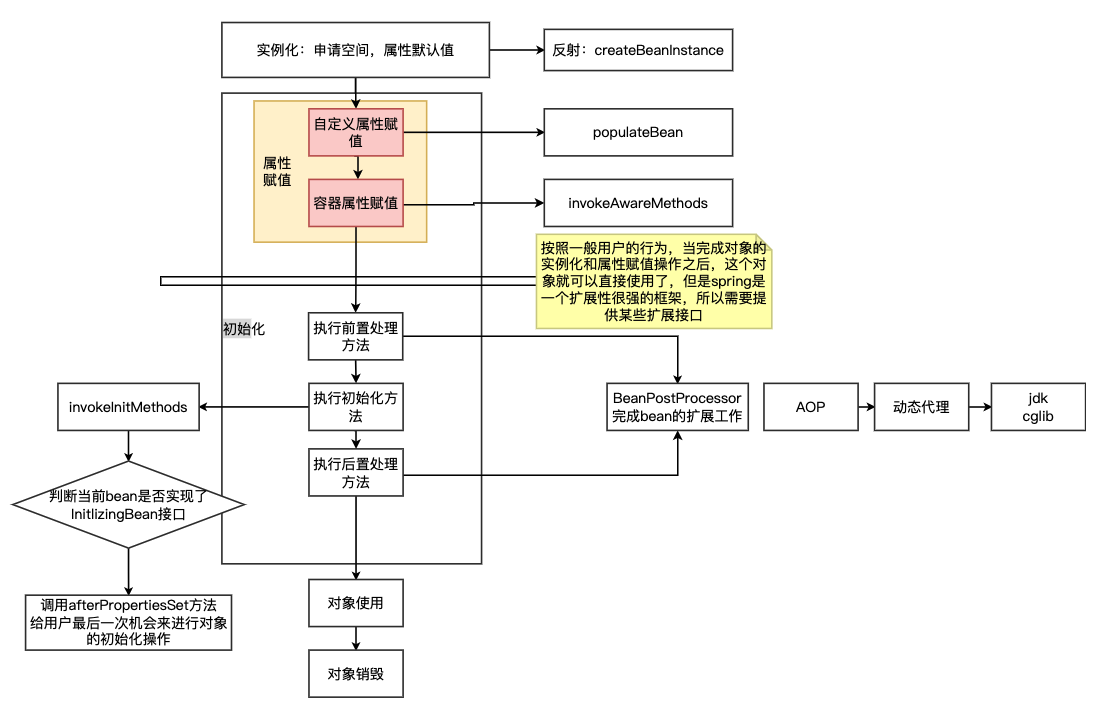
@Required：byname，用于setX，必须

@Autowired（required）/@Qualitifier/@Primary：byType用于方法，filed，构造器，可选是否必填。

<https://blog.csdn.net/qq_32885621/article/details/105793417>

## Bean的生命周期：



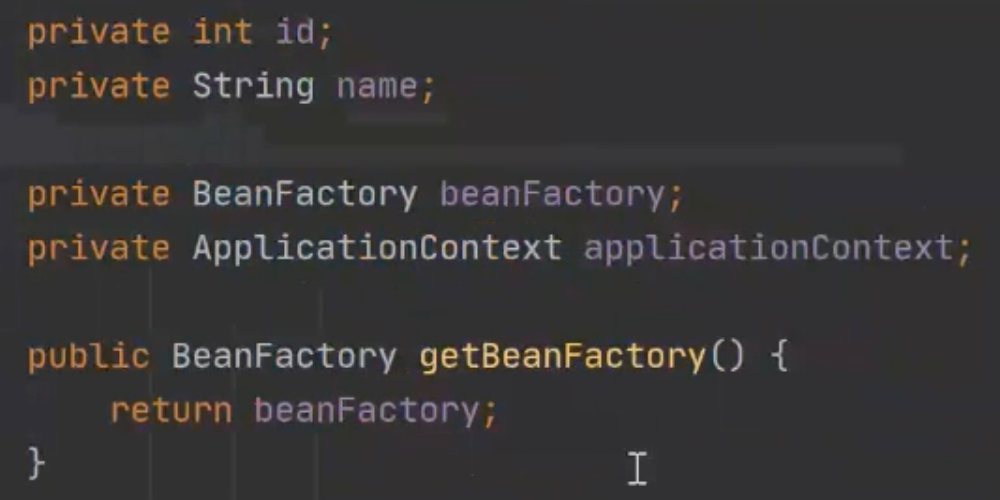


1:实例化：主要是用反射创建对象：createBeanInstance

2:初始化：

2-1自定义参数的参数的初始化（populateBean）这里可以引出循环引用的问题，后面会将。

2-2容器属性的初始化（invokeAwareMethods），这个主要是容器相关类向bean中注入的方式。类似下面BeanFactory/ApplicationContext的注入，这类的注入主要是靠 implements 对应的Aware接口，实现注入的。





3:执行init-method

就是在注册的时候注册的init-method，这个逻辑中还会判断 当前bean是否实现了InitizedBean接口，如果实现了，会回调afterPropertiesSet方法。

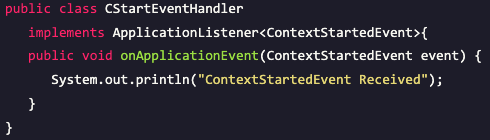
4:在init-method前后，会执行Bean的后置处理器。AOP就是在这里实现的。

## Spring事件：

### 注册处理事件（订阅事件）：

ApplicationContext容器支持事件，容器本身有很多的事件，我们可以 像注册bean一样，注意事件监听器：

1:实现ApplicationListener接口，并指定范型类型（表示自己处理哪种事件）



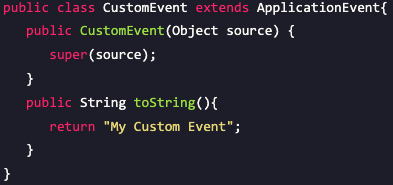
2:注册bean



### 自定义事件：

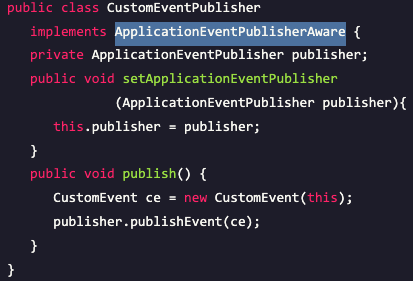
自己也可定义事件，像注册普通bean 那样注册 事件发布器，事件处理器

1:定义事件



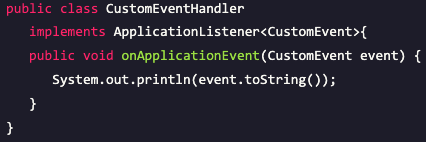
2:定义事件发布器

继承ApplicationEventPublisherAware接口





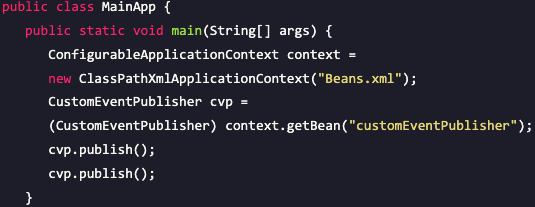
3:定义事件处理器（同订阅事件）





4:发布事件

通过容器获取publisher，然后发布事件



## Spring相关问题：

### 1:Spring如何解决循环依赖

<https://segmentfault.com/a/1190000023647227>

比如A依赖B，B依赖A

问题原因：

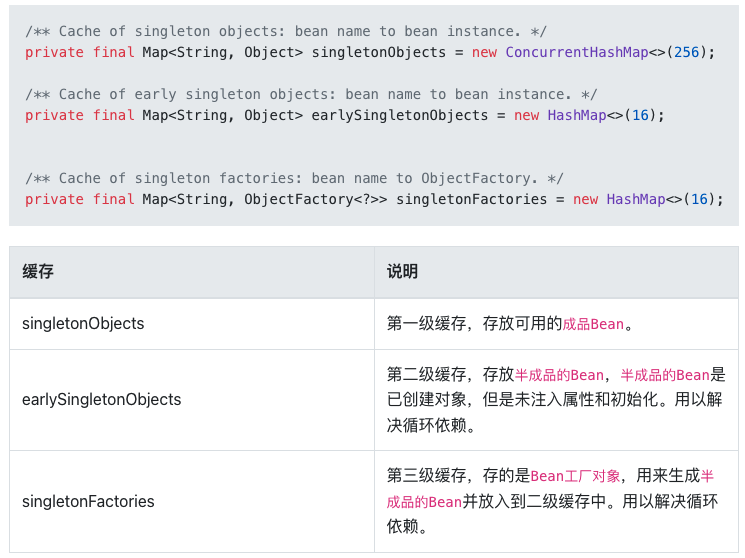
在填充数据polulateBean阶段（已经实例化），给A填充B，发现B不在，那么会去创建B，创建B的polulateBean阶段会去找A，此时A处于实例化了，但是没有初始化，此时如果再去创建A就会死循环。

通过三级缓存来解决：

**核心思想：提前曝光**

第一层放完成构造好的对象，第二层放已经实例化的，但是没有初始化的，比如上面的A。此时B在第二层找到了A，那么B初始化完成，A也就完成了。

第三层放置半成品bean，和动态代理有关系

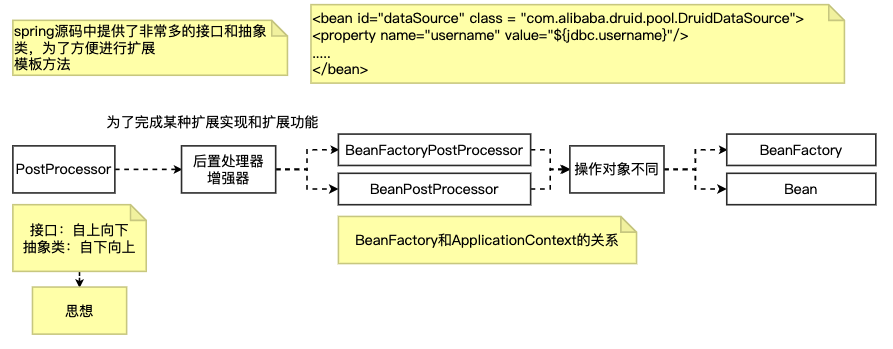


### 5:BeanFactory和FactoryBean的区别

BeanFactory：bean工厂的基础接口

FactoryBean：给程序员用户扩展实现自己的容器

## 后置处理器（可以自己扩展）：



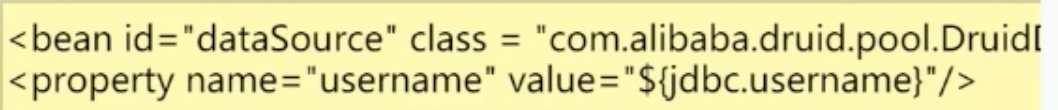
分为两种：

### BeanFacotry的后置处理器：

这个是在BeanFacotry处理化完之后，做的后置处理，可以自己扩展

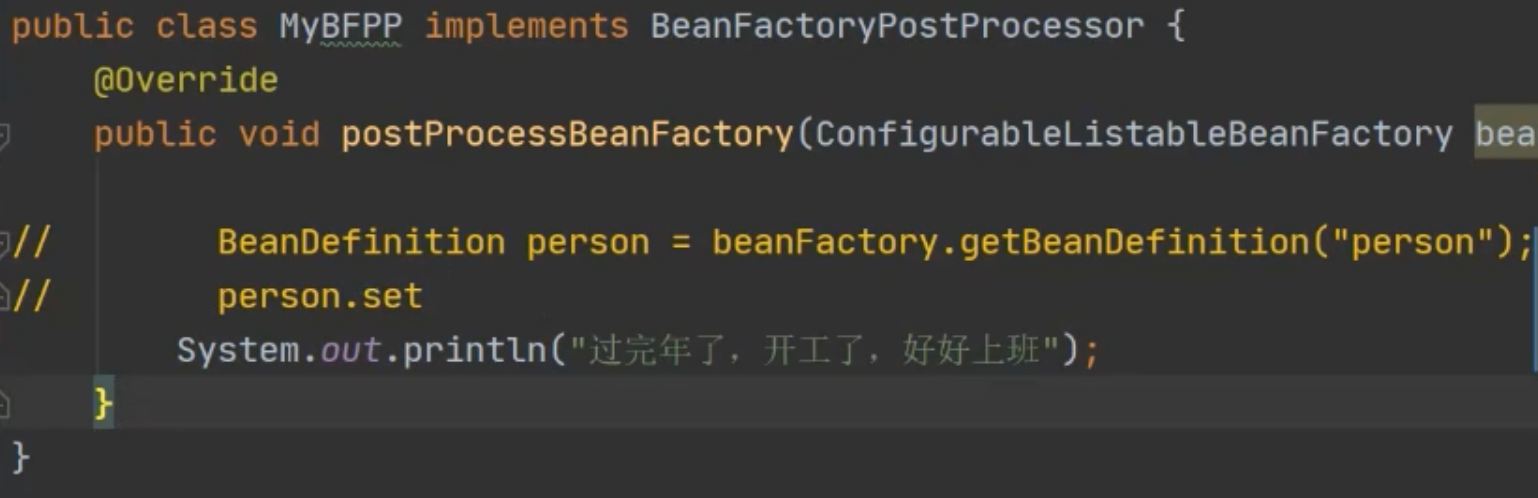
举例：

BeanFactory中有BeanDefination，对BeanFactoryPostProccess 也就是对BeanDeination做扩展操作，所以比如占位符这样的替换功能，就是使用的BeanfactoryPostProccess实现的。



BeanFactoryPostProcess自己定义方式：

1:定义类



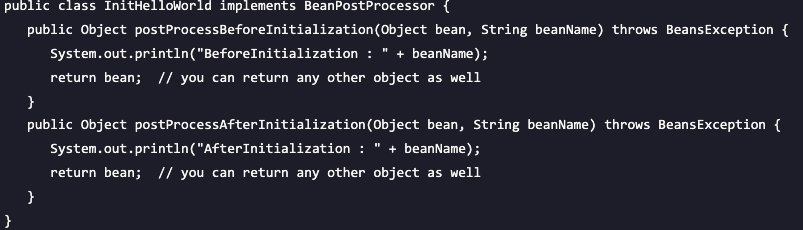
2:注册

### Bean的后置处理器：

Bean在初始化完成之后，执行init-method前后，可以扩展一些操作，就是使用的后置处理器BeanPostProccess（AOP在这里实现）

Bean 后置处理器允许在调用初始化方法（init-method）前后对 Bean 进行额外的处理。

1:定义



2:注册：对于全局的bean生效

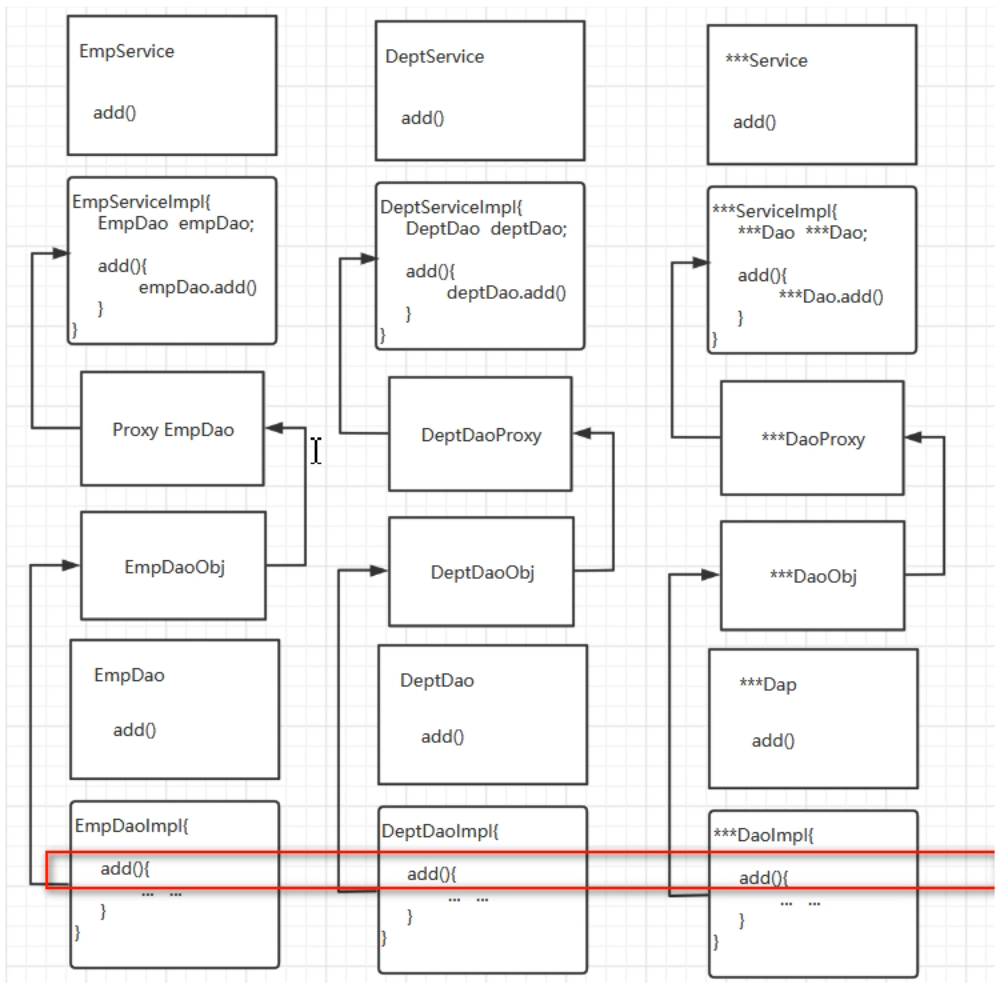


## SpringAOP：

面向切面编程，**可以在不修改现有代码的情况下，进行功能的拓展**（开闭原则）

层体现在这里，对于整个Dao层横切一刀，然后对Dao做代理，Service拿到的是Dao的代理对象。

### AOP基础知识：



切面：逻辑层面就是上图的横切面，物理上实际就是增强的功能的结合，也就是一系列Advice的组合

连接点：可以被增强的方法（可以增强，但是不一定会去增强）

切入点：实际被增强的方法（所以每个切点都是连接点）

目标target：被增强方法所属的对象/**或者说是被代理的对象**

通知：实际增强的功能

前置通知：切点执行前增强

后置通知：切点执行后增强

异常通知：切点抛出异常的时候，增强

最终通知：后置可能因为抛异常而不执行，后置相当于try/catch/finally中的finally，无论是否出现异常，都会执行

环绕通知：前后的结合

织入：创建代理对象的工具类，把被代理的对象和 我们的切面中的切点的组合 声称代理对象。

**切入点定义：使用表达式进行定义，正则表达式方式的！**

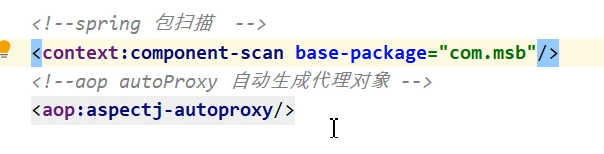


### 使用方式：

#### 1:配置配置文件

配置自动扫描路径

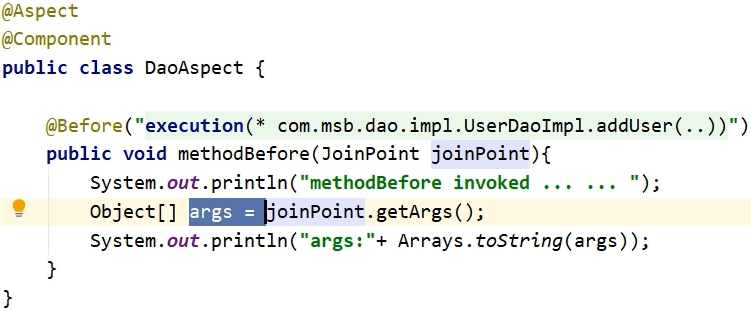
配置自动生成代理对象



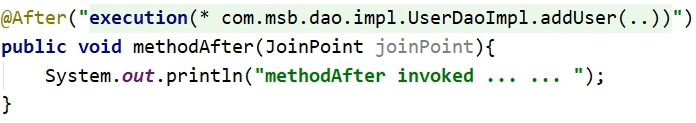
#### 2:创建切面类（需要注册到容器）

定义中也可以看到，切点也就是 被代理的方法（JoinPoint）。

前置通知：

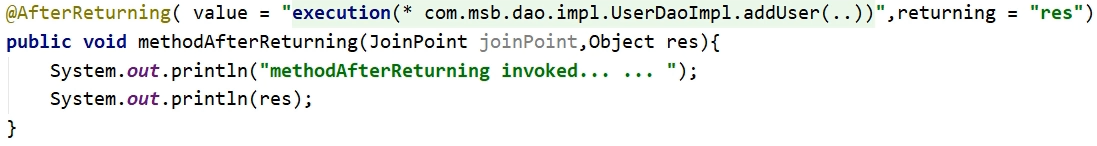


最终通知：



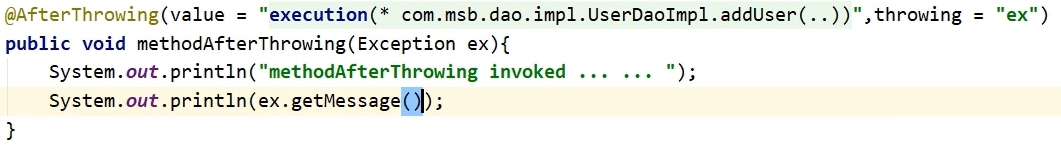
返回通知：

如果出现异常，不执行



异常通知：

出现异常会执行，不出现不执行；可以在通知中获取异常对象



环绕通知：

proceedingJoinPoint就是被代理的方法。

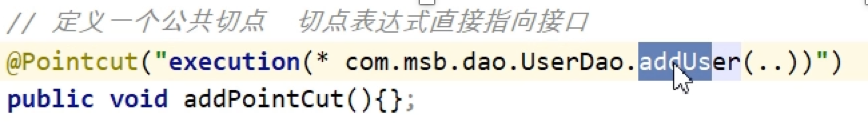
还需手动将参数反回给上层



3:可以创建公共切点：就是在定义通知的时候，使用公共切点（切点的变量）



**注意：**定义切点的时候，可以给接口进行定义，这样所有的实现类都会被代理。类似下面

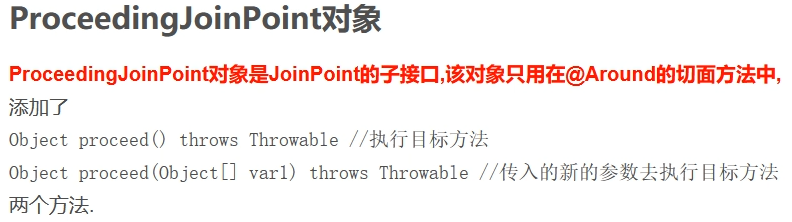


### 相关类：

JoinPoint：



procceedingJoinPoint



被多个切面进行切的时候，被代理的一个优先级：

使用@Order(int)进行定义优先级：



### 动态代理：

<https://www.cnblogs.com/zjfjava/p/13919437.html>

可以在运行时 对类进行动态的加强，在运行时声称代理类。

#### 1:JDK动态代理：

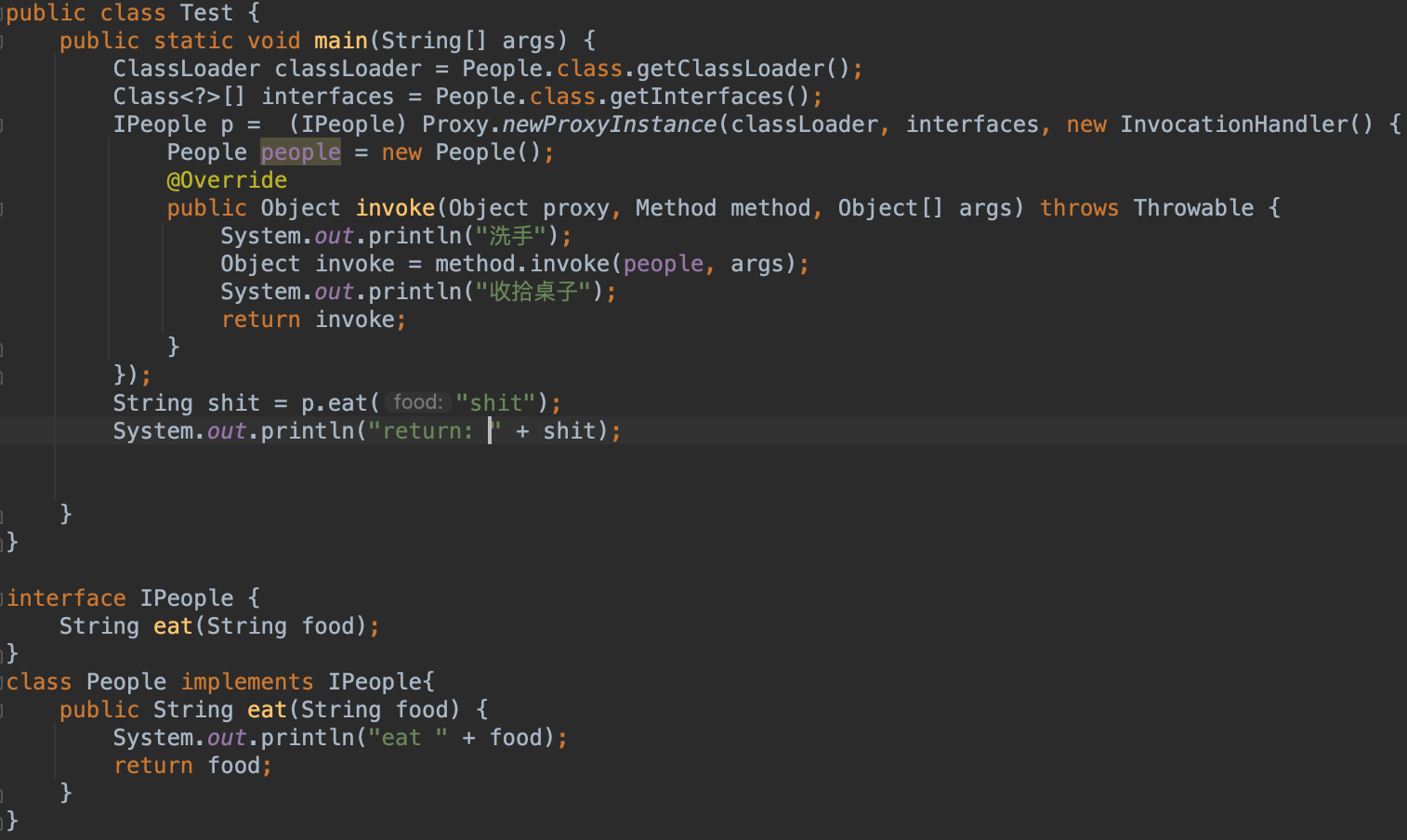
**面向接口**

原理是通过实现一个匿名内部类，然后通过组合的方式，实现动态代理。要求被代理的对象是需要实现代理的接口的

因为需要继承Proxy，所以匿名内部类只能通过接口实现

（<https://segmentfault.com/a/1190000021821314>）

实现方式：



#### 2:cglib动态代理：

**面向父类（继承）：**

原理是通过继承的方式，生成一个新的子类，然后重写子类的方法实现的，通过自动生成字节码技术。(重写，所以不能面向接口。)

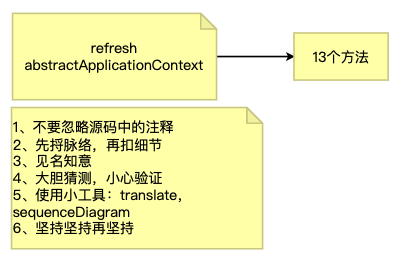
## Spring声明式事务管理：

# SpringMVC

# 如何读源码：

先Spring在SpringMVC再SpringBoot再SpringCloud

不要按模块看，分散且联系不到一起，应该从使用角度，按流程跟踪代码。



TODO 三级缓存看一下连桑的课

@Value

Spring事务管理

SpringMVC