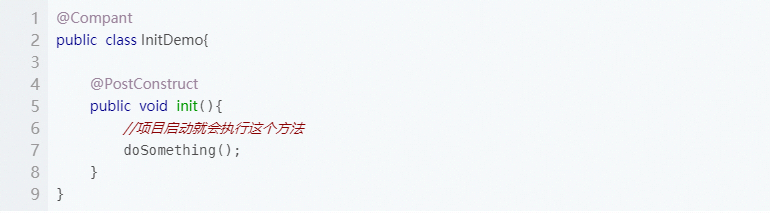
Spring实现项目启动时执行特定代码的三种方式？

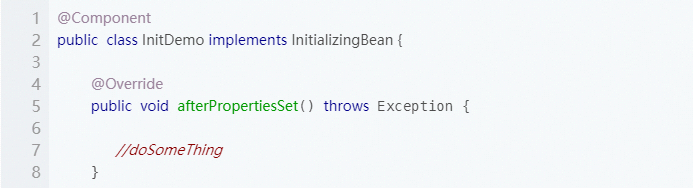
1、方法上加@PostConstruct注解



2、xml配置init-method方法



3、实现InitializingBean接口，重写afterPropertiesSet方法



Spring事物的五种隔离级别：

ISOLATION\_DEFAULT 这是一个PlatfromTransactionManager默认的隔离级别，使用数据库默认的事务隔离级别.另外四个与JDBC的隔离级别相对应   
ISOLATION\_READ\_UNCOMMITTED 这是事务最低的隔离级别，它充许别外一个事务可以看到这个事务未提交的数据。这种隔离级别会产生脏读，不可重复读和幻像读

ISOLATION\_READ\_COMMITTED 保证一个事务修改的数据提交后才能被另外一个事务读取。另外一个事务不能读取该事务未提交的数据。这种事务隔离级别可以避免脏读出现，但是可能会出现不可重复读和幻像读。

ISOLATION\_REPEATABLE\_READ 这种事务隔离级别可以防止脏读，不可重复读。但是可能出现幻像读。它除了保证一个事务不能读取另一个事务未提交的数据外，还保证了避免下面的情况产生(不可重复读)。

ISOLATION\_SERIALIZABLE 这是花费最高代价但是最可靠的事务隔离级别。事务被处理为顺序执行。除了防止脏读，不可重复读外，还避免了幻像读。

Spring事物的其中传播级别：

1、PROPAGATION\_REQUIRED ，默认的spring事务传播级别，使用该级别的特点是，如果上下文中已经存在事务，那么就加入到事务中执行，如果当前上下文中不存在事务，则新建事务执行。所以这个级别通常能满足处理大多数的业务场景。

调用端有事物在运行，被调用端将在该事物中运行，否则新建事物，如果被调用端发生异常，则调用端与被调用端事物都将回滚

2、PROPAGATION\_SUPPORTS ，从字面意思就知道，supports，支持，该传播级别的特点是，如果上下文存在事务，则支持事务加入事务，如果没有事务，则使用非事务的方式执行。所以说，并非所有的包在transactionTemplate.execute中的代码都会有事务支持。这个通常是用来处理那些并非原子性的非核心业务逻辑操作。应用场景较少。

3、PROPAGATION\_MANDATORY ， 该级别的事务要求上下文中必须要存在事务，否则就会抛出异常！配置该方式的传播级别是有效的控制上下文调用代码遗漏添加事务控制的保证手段。比如一段代码不能单独被调用执行，但是一旦被调用，就必须有事务包含的情况，就可以使用这个传播级别。

4、PROPAGATION\_REQUIRES\_NEW ，从字面即可知道，new，表示当前方法必须运行在自己的事物中（即该方法使用@Transcationl修饰，调用方也使用@Transcationl修饰），并且同时将上下文中的事务挂起，执行当前新建事务完成以后，上下文事务恢复再执行。

这是一个很有用的传播级别，举一个应用场景：现在有一个发送100个红包的操作，在发送之前，要做一些系统的初始化、验证、数据记录操作，然后发送100封红包，然后再记录发送日志，发送日志要求100%的准确，如果日志不准确，那么整个父事务逻辑需要回滚。  
怎么处理整个业务需求呢？就是通过这个PROPAGATION\_REQUIRES\_NEW 级别的事务传播控制就可以完成。发送红包的子事务不会直接影响到父事务的提交和回滚。

5、PROPAGATION\_NOT\_SUPPORTED ，这个也可以从字面得知，not supported ，不支持，当前级别的特点就是上下文中存在事务，则挂起事务，执行当前逻辑，结束后恢复上下文的事务。

这个级别有什么好处？可以帮助你将事务极可能的缩小。我们知道一个事务越大，它存在的风险也就越多。所以在处理事务的过程中，要保证尽可能的缩小范围。比如一段代码，是每次逻辑操作都必须调用的，比如循环1000次的某个非核心业务逻辑操作。这样的代码如果包在事务中，势必造成事务太大，导致出现一些难以考虑周全的异常情况。所以这个事务这个级别的传播级别就派上用场了。用当前级别的事务模板抱起来就可以了。

6、PROPAGATION\_NEVER ，该事务更严格，上面一个事务传播级别只是不支持而已，有事务就挂起，而PROPAGATION\_NEVER传播级别要求上下文中不能存在事务，一旦有事务，就抛出runtime异常，强制停止执行！这个级别上辈子跟事务有仇。

7、PROPAGATION\_NESTED ，字面也可知道，nested，嵌套级别事务。该传播级别特征是，如果上下文中存在事务，则嵌套事务执行，如果不存在事务，则同propagation\_required的一样。

那么什么是嵌套事务呢？很多人都不理解，我看过一些博客，都是有些理解偏差。

嵌套是子事务套在父事务中执行，子事务是父事务的一部分，在进入子事务之前，父事务建立一个回滚点，叫save point，然后执行子事务，这个子事务的执行也算是父事务的一部分，然后子事务执行结束，父事务继续执行。重点就在于那个save point。看几个问题就明了了：

如果子事务回滚，会发生什么？

父事务会回滚到进入子事务前建立的save point，然后尝试其他的事务或者其他的业务逻辑，父事务之前的操作不会受到影响，更不会自动回滚。

如果父事务回滚，会发生什么？

父事务回滚，子事务也会跟着回滚！为什么呢，因为父事务结束之前，子事务是不会提交的，我们说子事务是父事务的一部分，正是这个道理。那么：

事务的提交，是什么情况？

是父事务先提交，然后子事务提交，还是子事务先提交，父事务再提交？答案是第二种情况，还是那句话，子事务是父事务的一部分，由父事务统一提交。

现在你再体会一下这个”嵌套“，是不是有那么点意思？

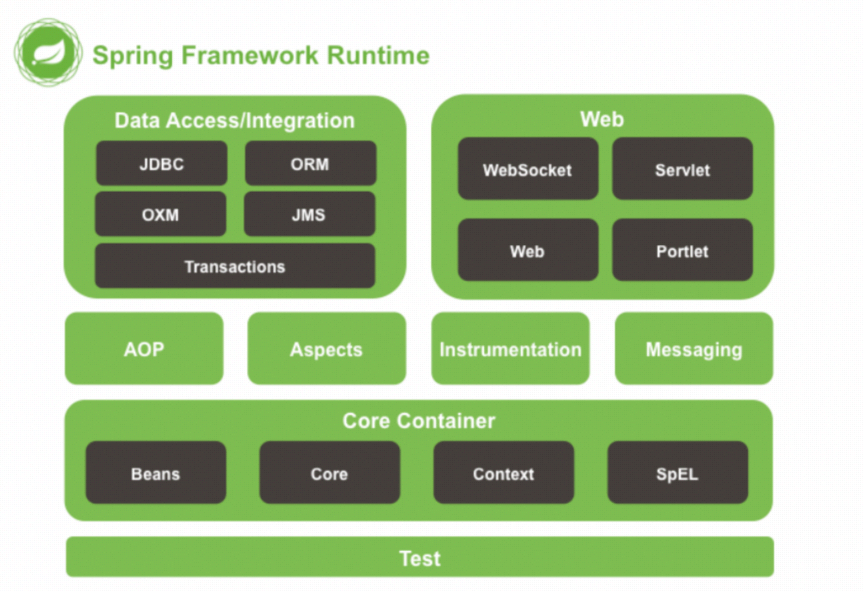
Spring中文文档

https://github.com/DocsHome/spring-docs/blob/master/SUMMARY.md

# 1、spring 使用

## 1.1 spring的核心模块的简介与使用

### 1.1 spring的核心模块简介



1 、Spring Core Container：  
模块作用: Core 和 Beans 模块是框架的基础部分， 提供 IoC （转控制） 和依赖注入特性。 这里的基础概念是 BeanFactory，它提供对 Factory 模式的经典实现来消除对程序’性单例模式的需要，并真正地允许你从程序逻辑中分离出依赖关系和配置

1 .1 Core主要包含 Spring 框架基本的核心工具类，Spring 的其他组件都要用到这个包里的类，Core模块是其他组件的基本核心。

1 .2 Beans (BeanFacotry的作用  
它包含访问配直文件、创建和管理 bean以及进行 Inversion of Control I Dependency Injection ( IoC/DI ） 操作相关的所有类

1 .3 Context(处理BeanFactory，一下还是ApplicationContext的作用)  
模构建于Core 和Beans 模块基础之上，提供了一种类似JNDI注册器的框架式的对象访问方法。 Context 模块继承了Beans 的特性，为 Spring 核心提供了大量扩展，添加了对国际化（例如资源绑定）、事件传播、资源加载和对 Context 的透明创建的支持。Context 模块同时也支持 J2EE 的一些特 性，ApplicationContext 接口是 Context 模块的关键  
本质区别:(使用BeanFacotry been是延时加载的,ApplicationContext是非延时加载的)

1 .4)Expression Language  
模块提供了强大的表达式语言， 用于在运行时查询和操纵对象。 它是 JSP 2.1 规范中定义的 unifed expression language 的扩展。该语言支持设直／获取属性的值，属性的分配，方法的调用，访问数组上下文（ accessiong the context of arrays ）、容器和索引器、逻辑和算术运算符、命名变量以及从Spring的 IoC 容器中根据名称检索对象。它也支持list投影、选择和一般的list聚合

2、Spring Data Access/Integration

2.1 )JDBC  
模块提供了一个 JDBC 抽象层， 它可以消除冗长的 JDBC 编码和解析数据库厂商特有的错误代码。这个模块包含了 spring 对JDBC数据访问进行封装的所有类

2.2 ORM模块为流行的对象－关系映射API，如 JPA、JDO、Hibernate、iBatis等，提供了一个交互层。利用ORM封装包，可以混合使用所有 Spring 提供的特性进行O/R 映射，如前边提到的简单声明性事务管理。

2.3 OXM 模块提供了一个对 ObjecνXML 映射实现的抽象层，Object/XML映射实现包括JAXB、 Castor、XMLBeans、JiBX和XStrearn

2.4 JMS ( Java Messaging Service）  
模块主要包含了一些制造和消费消息的特性。

2.5 Transaction  
支持编程和声明性的事务管理，这些事务类必须实现特定的接口，并且对所有的POJO都适用

3、Spring Web  
Web 模块：提供了基础的面向Web的集成特性c例如，多文件上传、使用servlet listeners 初始化IoC 容器以及一个面向Web的应用上下文。它还包含Spring远程支持中Web的相关部分。

4、Spring Aop  
4.1 Aspects 模块提供了对 AspectJ 的集成支持。  
4.2 Instrumentation 模块提供了 class instrumentation 支持和 classloader 实现，使得可以在特定的应用服务器上使用

5、Test

Test 模块支持使用 JUnit 和 TestNG 对 Spring 组件进行测试

Spring bean的生命周期



1、实例化一个bean，也就是new；  
2、按照Spring上下文对Bean进行配置；  
3、如果Bean实现了BeanNameAware接口，会调用他的实现setBeaName(String)方法， 此处传递的就是Spring配置文件中的Bean id的值。

4、如果Bean实现了BeanFactoryAware接口，则会调用setBeanFactory(BeanFactory) 接口。

5、如果Bean实现了ApplicationContextAware接口

setApplicationContext(ApplicationContext)方法，传入 Spring 上下文。  
6、如果Bean实现了BeanPostProcess接口，Spring将调用它们的

postProcessBeforeInitialization（预初始化）方法（作用是在Bean实例创建成功后对进行增强处理，如对Bean进行修改，增加某个功能）  
7、如果Bean实现了InitializingBean接口，Spring将调用它们的afterPropertiesSet方法，作用与在配置文件中对Bean使用init-method声明初始化的作用一样，都是在Bean的全部属性设置成功后执行的初始化方法。  
8、如果Bean实现了BeanPostProcess接口，Spring将调用它们

postProcessAfterInitialization（后初始化）方法（作用与6的一样，只不过6是在Bean初始化前执行的，而这个是在Bean初始化后执行的，时机不同 )  
9.经过以上的工作后，Bean将一直驻留在应用上下文中给应用使用，直到应用上下文被销毁

10.如果Bean实现了DispostbleBean接口，Spring将调用它的destory方法，作用与在配置文件中对Bean使用destory-method属性的作用一样，都是在Bean实例销毁前执行的方法。

InstantiationAwareBeanPostProcessor的postProcessBeforeInstantiation与postProcessAfterInstantiation方法CreateBean下resolveBeforeInstantiation会调用before方法，同时会缓存aop中的切面等对象，在doCreateBean下的populateBean调用after方法，

BeanPostProcessor接口 在doCreateBean下initializeBean(beanName,

exposedObject, mbd)下applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName)方法内调用，该方法内首先调用aware接口，然后调用BeanPostProcessor的before 方法，然后调用InitializingBean，然后调用BeanPostProcessor的after方法

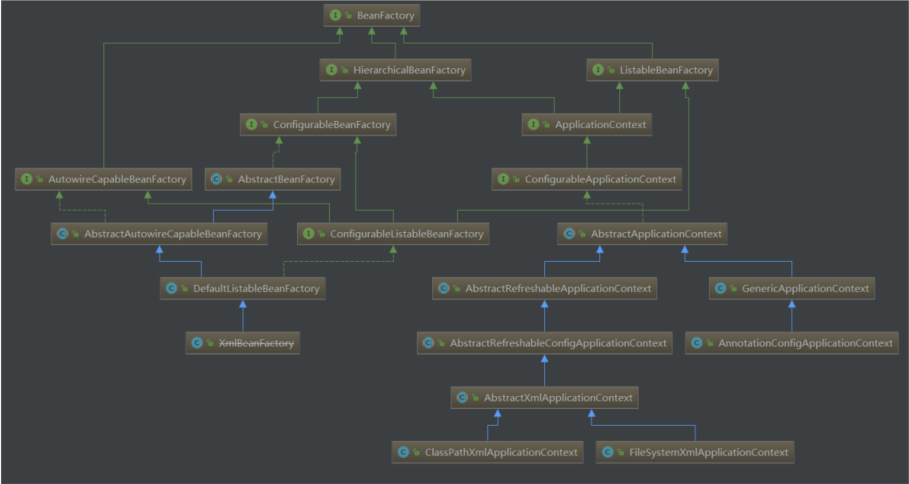
initializeBean(beanName, exposedObject, mbd)下同时调用

invokeAwareMethods(beanName, bean)该方法依次调用BeanNameAware、BeanClassLoaderAware、BeanFactoryAware

### 1.2 IOC的使用

ioc的思想最核心的地方在于， 资源不由使用资源的双方管理， 而由不使用资源的第三方管理， 这可以带来很多好处。

第一， 资源集中管理， 实现资源的可配置和易管理。 第二， 降低了使用资源双方的依赖程度， 也就是我们说的耦合



#### 依赖注入的方式与区别：

注入是将已存在于容器的实例赋值给应用

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（ConstructorInjection） 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：**

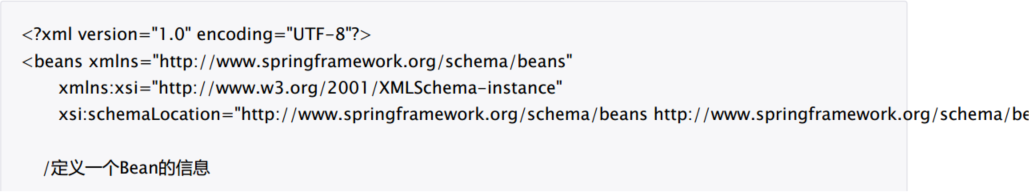
两种依赖方式都可以使用， 构造器注入和Setter方法注入。 最好的解决方案是用  
构造器参数实现强制依赖， setter方法实现可选依赖。





#### 1、xml配置文件与配置类配置bean

基于xml配置





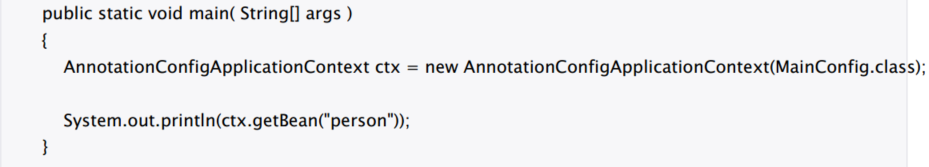
去容器中读取bean



配置类配置

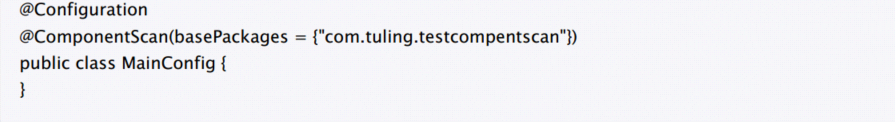


注意: 通过@Bean的形式是使用的话，bean的默认名称是方法名，若@Bean(value="bean的名称")  
那么bean的名称是指定的  
去容器中读取Bean的信息（传入配置类）

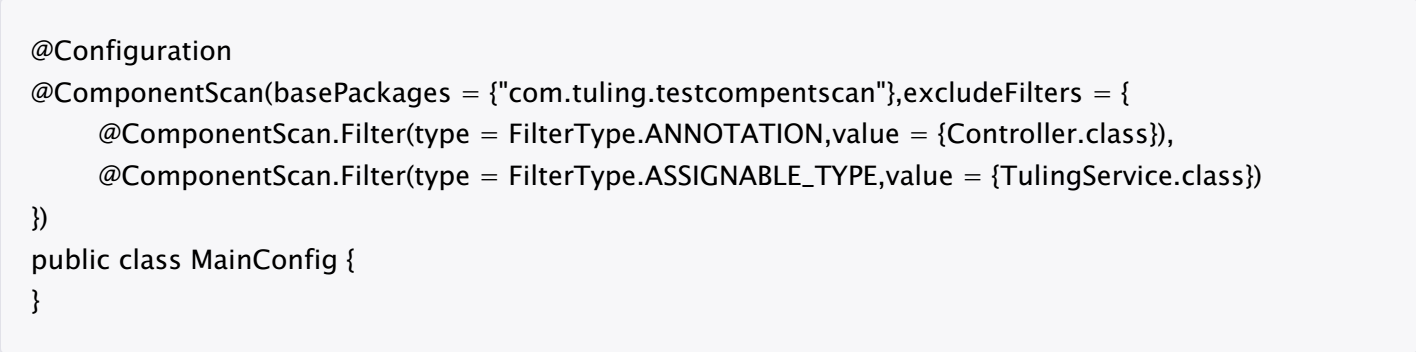


#### 2、在配置类上使用@CompentScan注解进行包扫描

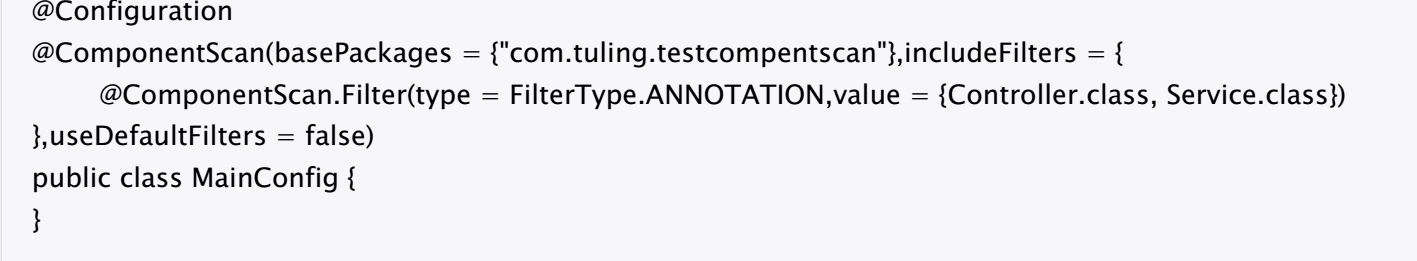
使用@Component或@Service或@Repository注解配置了bean，spring不能自动扫描，需要配置需要spring扫描的包，故需要@CompentScan注解配置，spring boot项目的启动类@SpringBootApplication包含了@CompentScan注解 ，所以其所在包及其子包会被自动扫描



①排除用法 excludeFilters(排除@Controller注解的,和TulingService的)



②包含用法 includeFilters ,注意，若使用包含的用法，需要把useDefaultFilters属性设置为false（true表  
示扫描全部的）

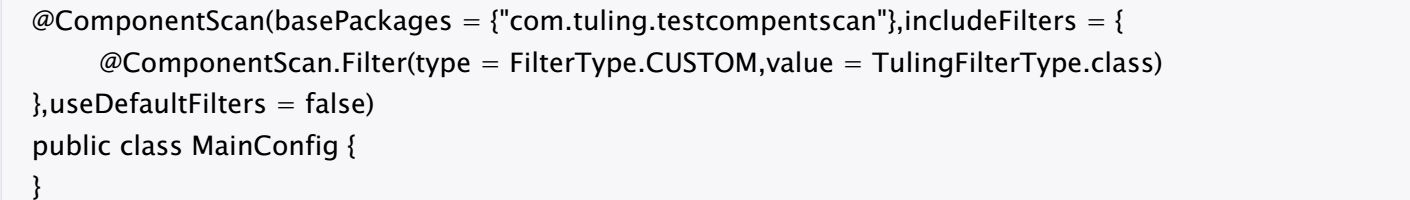


③ @ComponentScan.Filter type的类型  
a)注解形式的FilterType.ANNOTATION @Controller @Service @Repository @Compent  
b)指定类型的 FilterType.ASSIGNABLE\_TYPE @ComponentScan.Filter(type =

FilterType.ASSIGNABLE\_TYPE,value = {TulingService.class})  
c)aspectj类型的 FilterType.ASPECTJ(不常用)  
d)正则表达式的 FilterType.REGEX(不常用)  
e)自定义的 FilterType.CUSTOM

③.①FilterType.CUSTOM 自定义类型如何使用

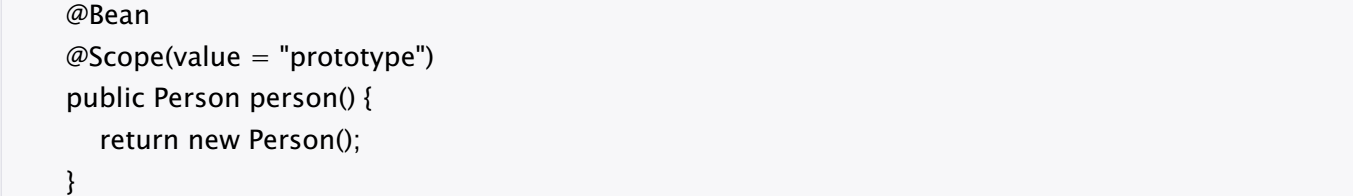




#### 3、配置bean的作用域

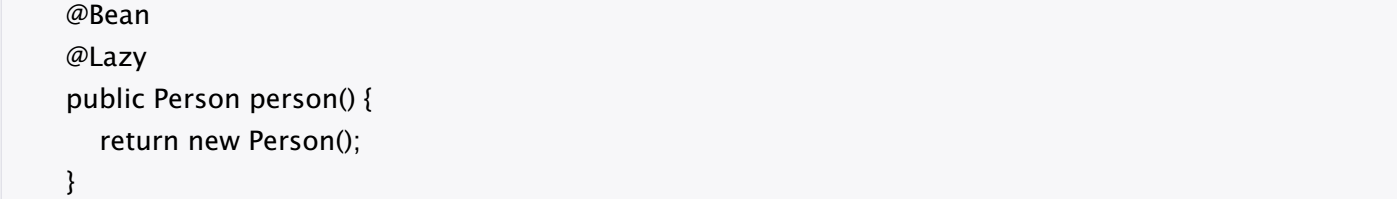
①在不指定@Scope的情况下，所有的bean都是单实例的bean,而且是饿汉加载(容器启动实例就创建好了)

②指定@Scope为 prototype 表示为多实例的， 而且还是懒汉模式加载（IOC容器启动的时候，并不会创建对象，而是在第一次使用的时候才会创建）



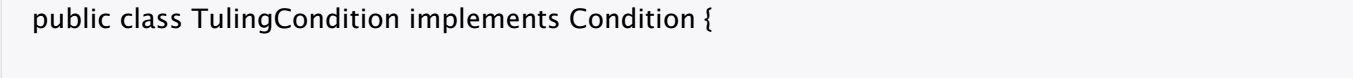
③@Scope指定的作用域方法取值  
a) singleton 单实例的(默认)  
b) prototype 多实例的  
c) request 同一次请求  
d) session 同一个会话级别

#### 4、Bean的懒加载@Lazy(主要针对单实例的bean 容器启动的时候，不创建对象，在第一次使用的时候才会创建该对象)

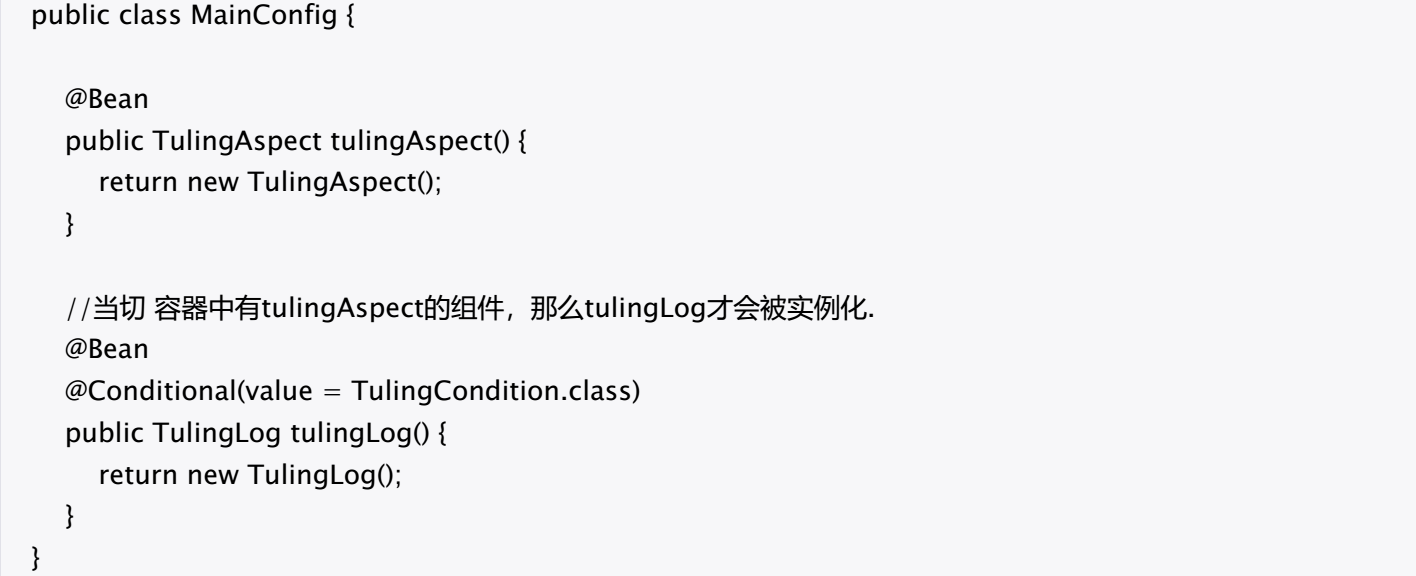


#### 5、@Conditional进行条件判断等

场景,有二个组件TulingAspect 和TulingLog，我的TulingLog组件是依赖于TulingAspect的组件  
应用:自己创建一个TulingCondition的类 实现Condition接口

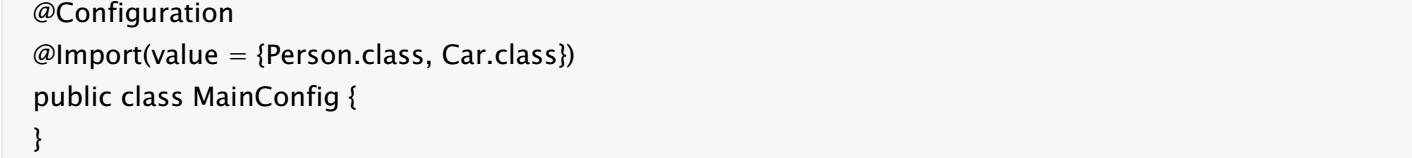






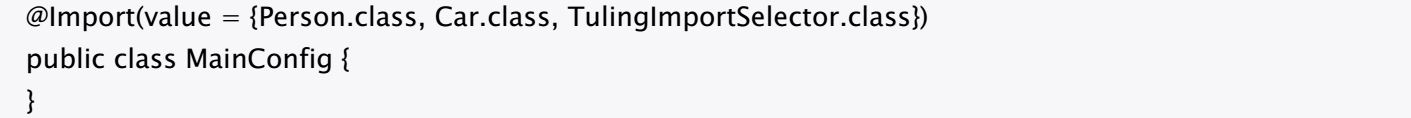
#### 6、向IOC容器中添加组件

①通过@CompentScan +@Controller @Service @Respository @compent  
适用场景: 针对我们自己写的组件可以通过该方式来进行加载到容器中。  
②通过@Bean的方式来导入组件(适用于导入第三方组件的类)  
③通过@Import来导入组件 （导入组件的id为全类名路径）

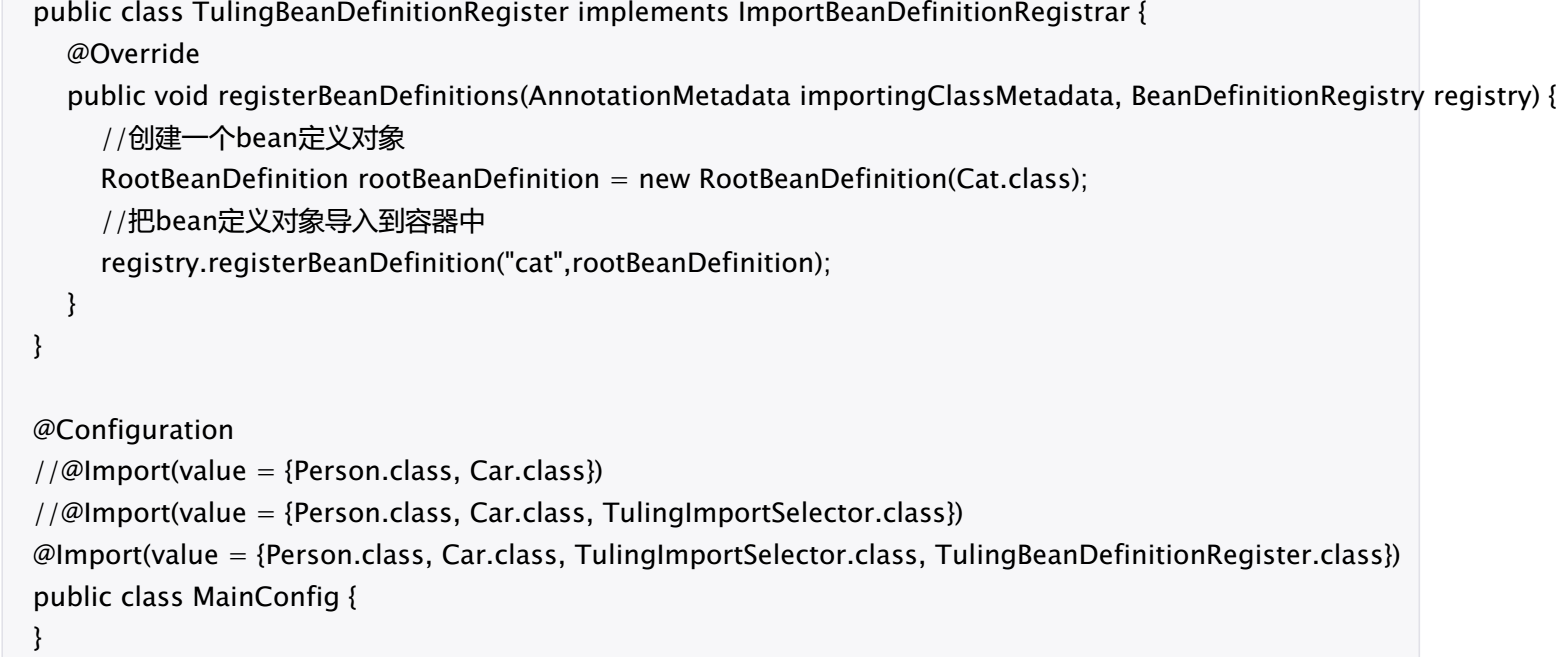


通过@Import 的ImportSeletor类实现组件的导入 (导入组件的id为全类名路径)





通过@Import的 ImportBeanDefinitionRegister导入组件 (可以指定bean的名称)

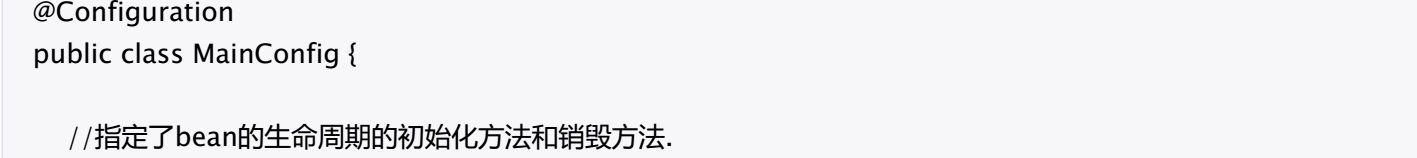


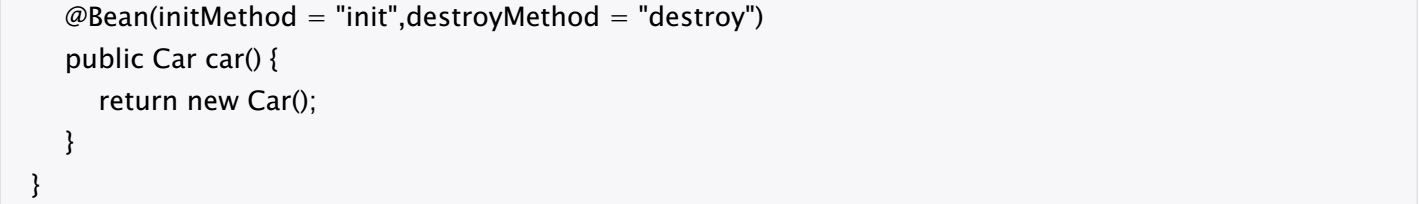
④通过实现FacotryBean接口来实现注册 组件



#### 7、Bean的初始化方法和销毁方法

①什么是bean的生命周期?  
bean的创建----->初始化----->销毁方法  
由容器管理Bean的生命周期，我们可以通过自己指定bean的初始化方法和bean的销毁方法



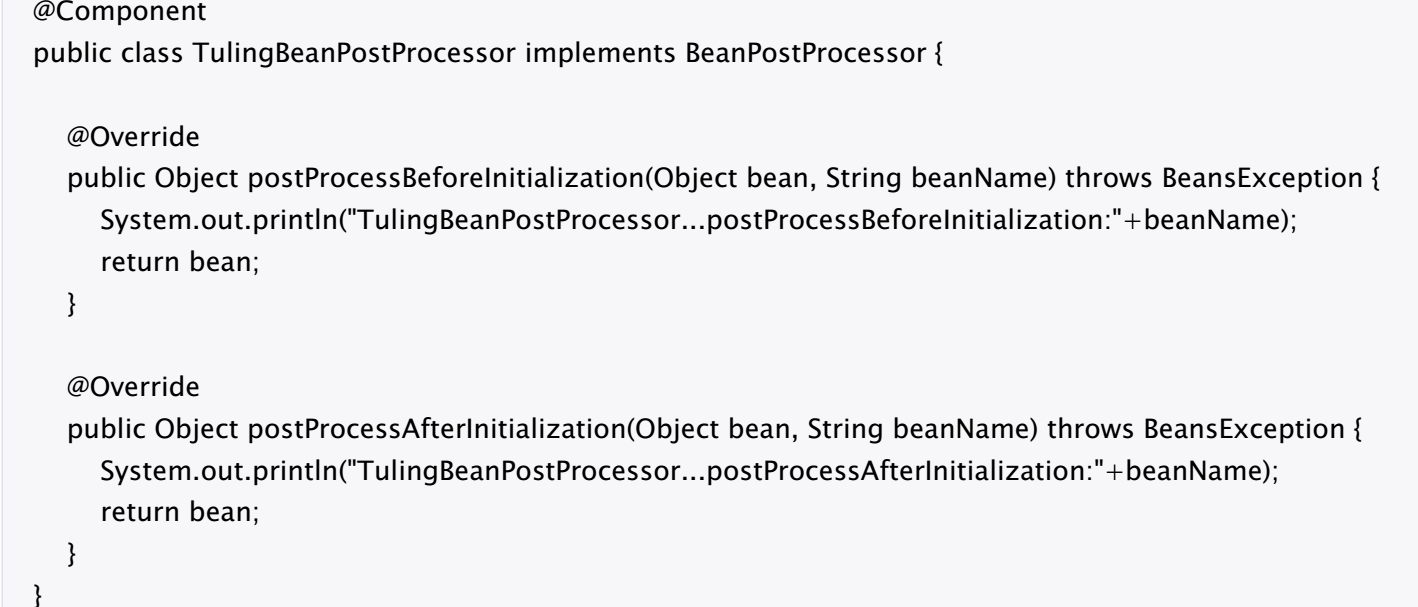


针对单实例bean的话，容器启动的时候，bean的对象就创建了，而且容器销毁的时候，也会调用Bean的销毁方法  
针对多实例bean的话,容器启动的时候，bean是不会被创建的而是在获取bean的时候被创建， 而且bean的销毁不受IOC容器的管理.  
②通过 InitializingBean和DisposableBean 的二个接口实现bean的初始化以及销毁方法

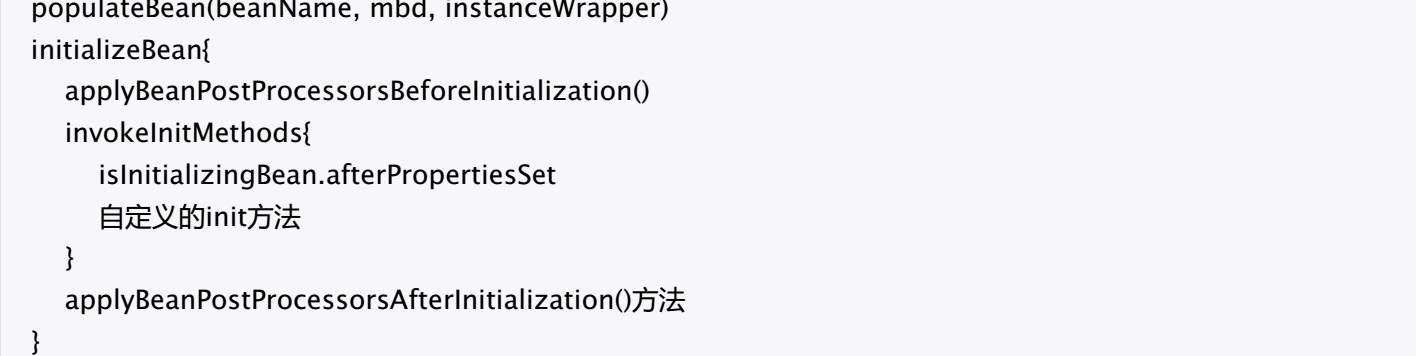


③通过JSR250规范 提供的注解@PostConstruct 和@ProDestory标注的方法

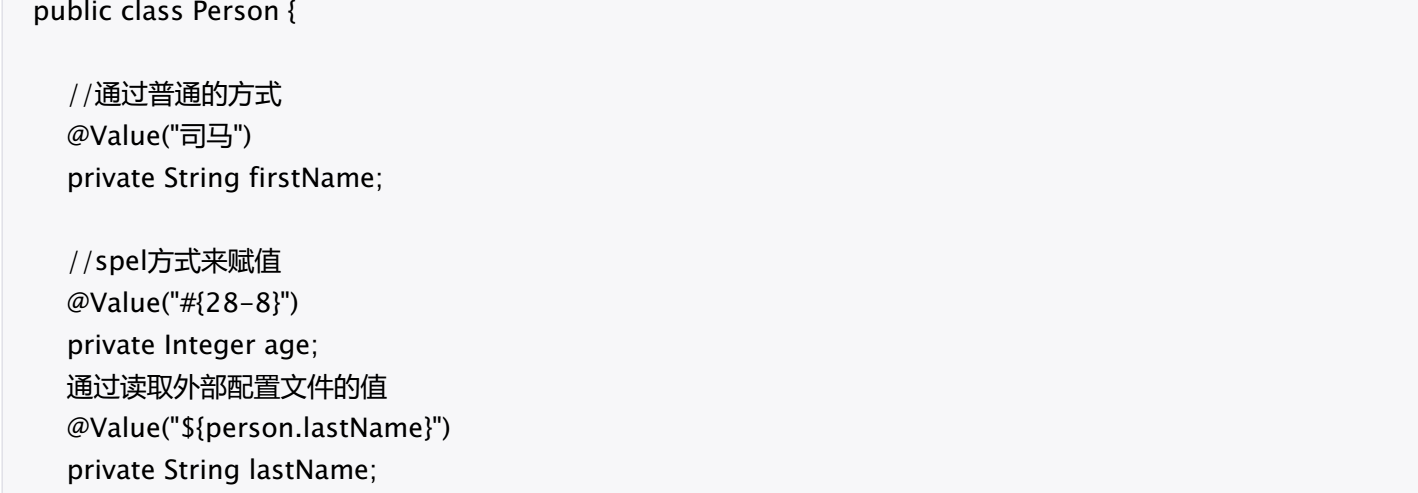


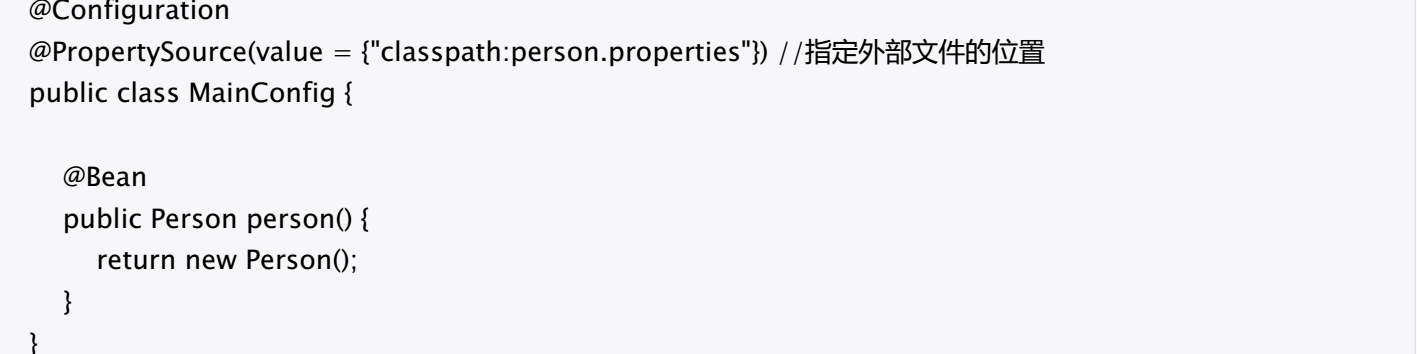
④通过Spring的BeanPostProcessor的 bean的后置处理器会拦截所有bean创建过程  
postProcessBeforeInitialization 在init方法之前调用  
postProcessAfterInitialization 在init方法之后调用  


BeanPostProcessor的执行时机



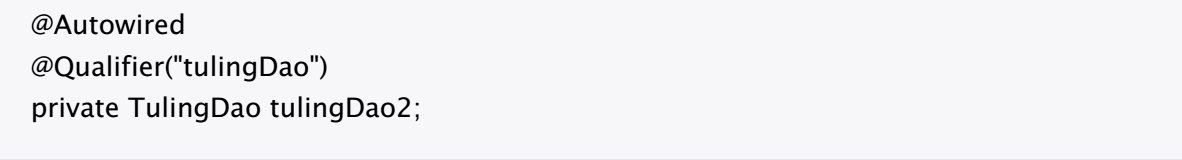
#### 8、通过@Value +@PropertySource来给组件赋值



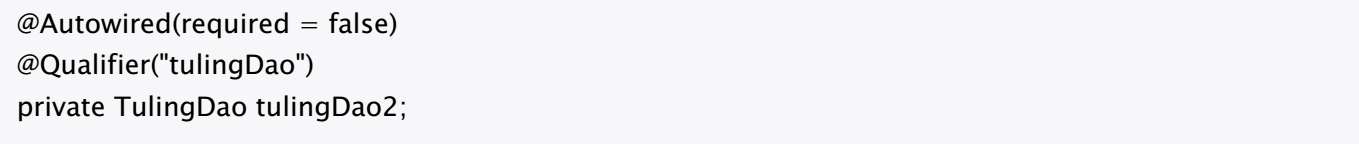


#### 9、自动装配

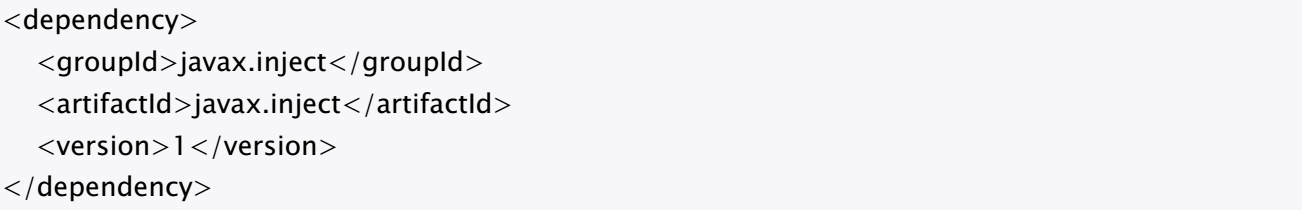
a: @Autowired自动装配首先时按照类型进行装配，若在IOC容器中发现了多个相同类型的组件，那么就按照属性名称来进行装配，@Resource属于J2EE默认使用名称装配，找不到相应的名称，则使用类型装配  
@Autowired  
private TulingDao tulingDao;  
比如，我容器中有二个TulingDao类型的组件 一个叫tulingDao 一个叫tulingDao2  
那么我们通过@AutoWired 来修饰的属性名称时tulingDao，那么拿就加载容器的tulingDao组件，若属性名称为tulignDao2 那么他就加载的时tulingDao2组件  
b：假设我们需要指定特定的组件来进行装配， 我们可以通过使用@Qualifier("tulingDao")来指定装配的组件或者在配置类上的@Bean加上@Primary注解



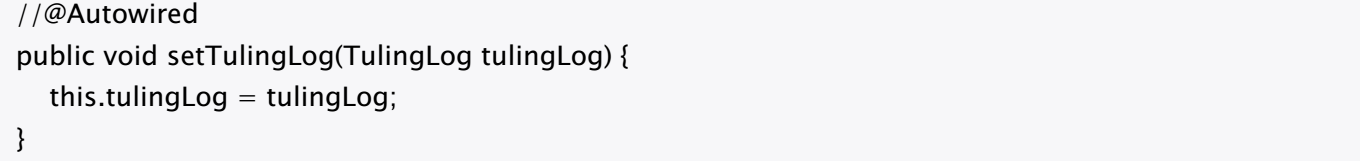
c假设我们容器中即没有tulingDao 和tulingDao2,那么在装配的时候就会抛出异常  
No qualifying bean of type 'com.tuling.testautowired.TulingDao' available  
若我们想不抛异常，我们需要指定 required为false的时候可以了



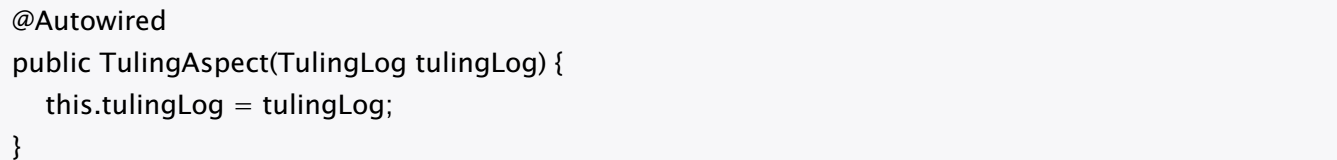
d:@Resource(JSR250规范)  
功能和@AutoWired的功能差不多一样，但是不支持@Primary 和@Qualifier的支持

@Resource属于J2EE默认使用名称装配，找不到相应的名称，则使用类型装配  
  
e:@InJect（JSR330规范）  
需要导入jar包依赖  
功能和支持@Primary功能 ,但是没有Require=false的功能  


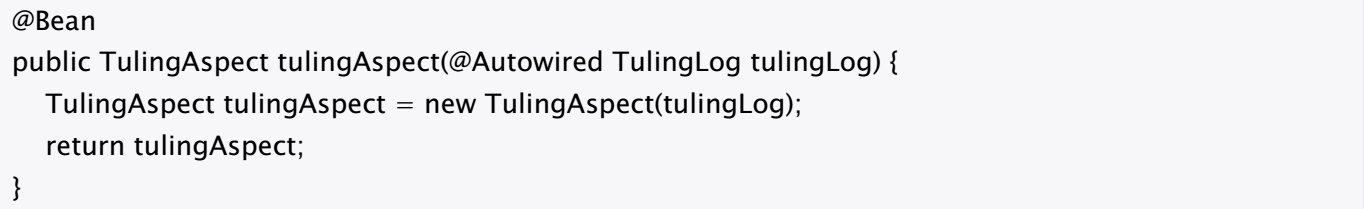
f使用autowired 可以标注在方法上  
标注在set方法



标注在构造方法上

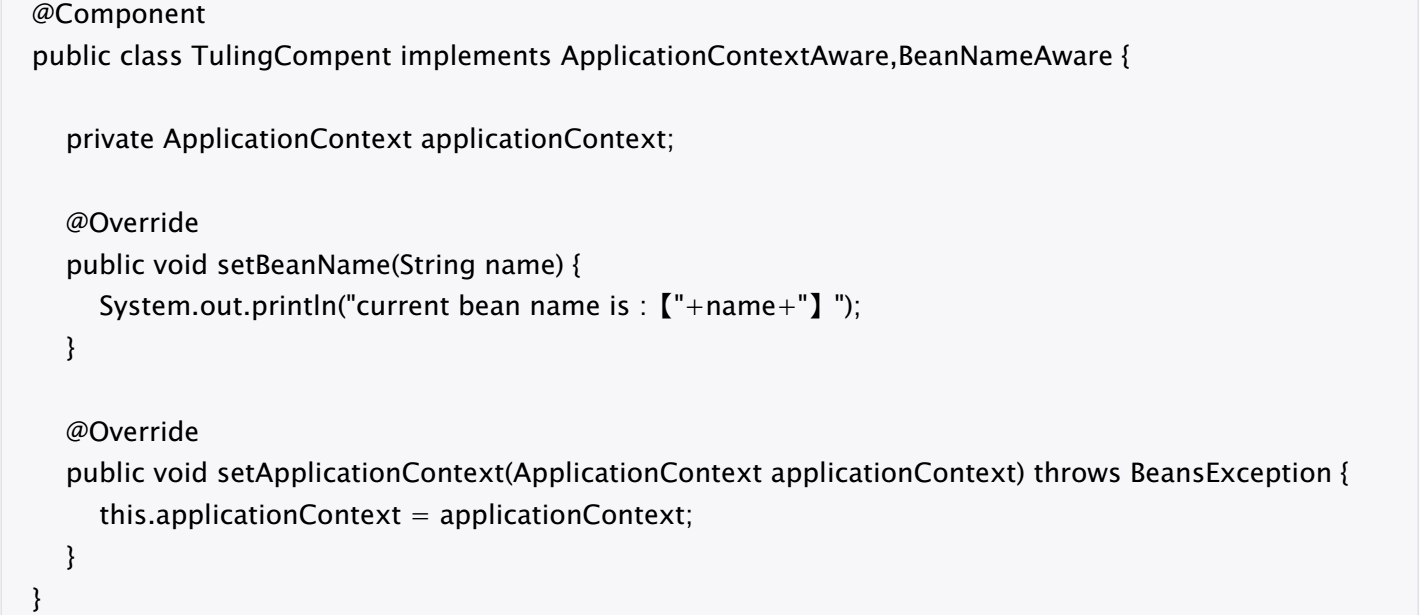


标注在配置类上的入参中（可以不写）



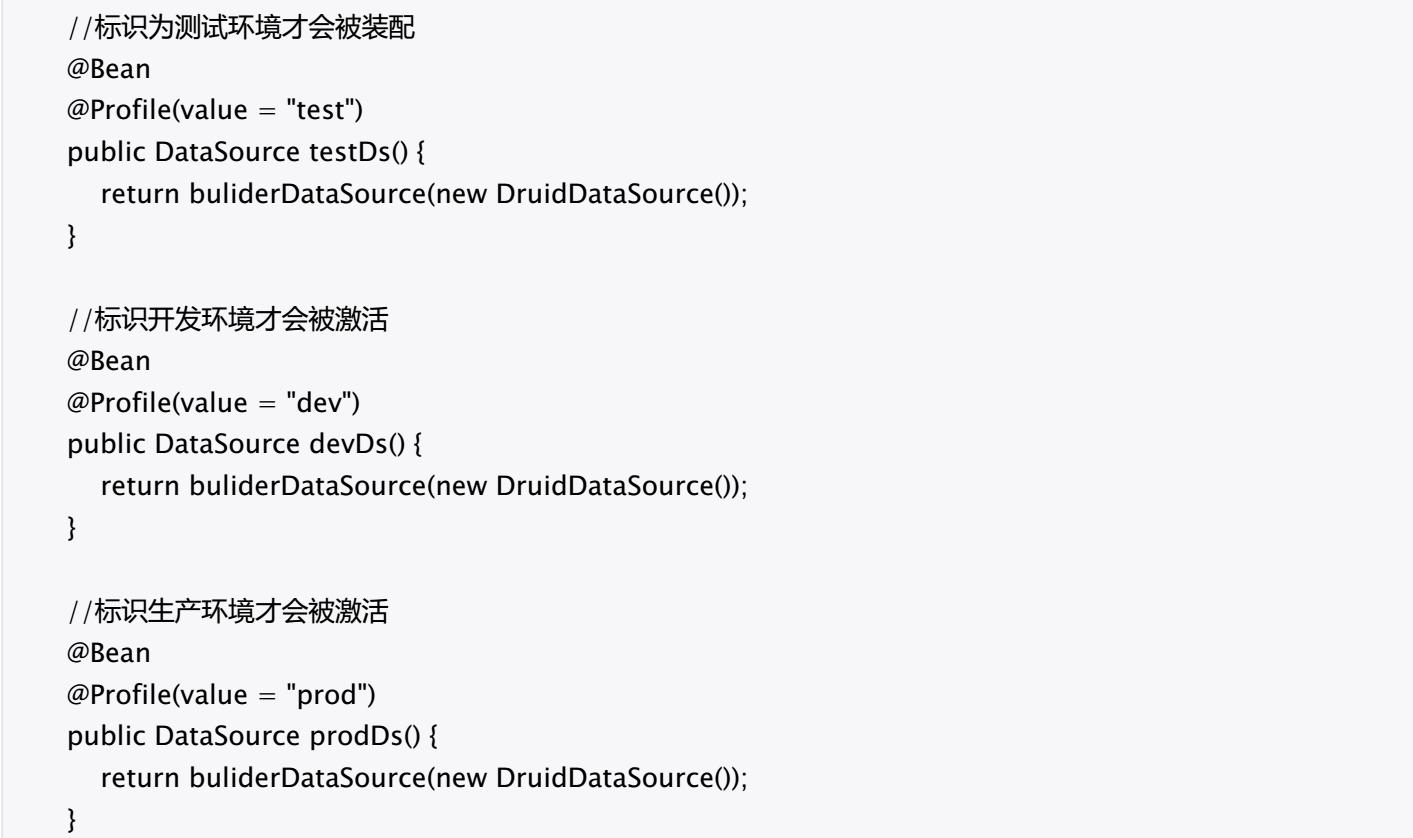
#### 10、我们自己的组件 需要使用spring ioc的底层组件的时候,比如 ApplicationContext

BeanNameAware接口给当前实例的bean的名称赋值，ApplicationContextAware获取applicationcontext后可以直接在容器中获取bean实例

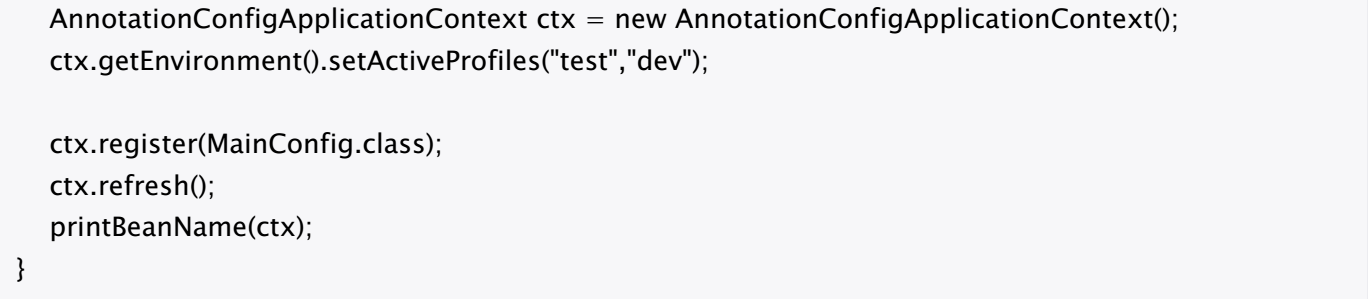


#### 11、通过@Profile注解 来根据环境来激活标识不同的Bean

@Profile标识在类上， 那么只有当前环境匹配， 整个配置类才会生效  
@Profile标识在Bean上 ， 那么只有当前环境的Bean才会被激活  
没有标志为@Profile的bean 不管在什么环境都可以被激活



激活切换环境的方法  
方法一:通过运行时jvm参数来切换 -Dspring.profiles.active=test| dev| prod  
方法二:通过代码的方式来激活



方式三：通过xml文件配置，并设置active profile实现目前该种方式最常用

## 1.2 AOP

### 2.1 aop的简介

AOP（Aspect Oriented Programming），即面向切面编程，可以说是OOP（Object Oriented Programming，面向对象编程）的补充和完善。OOP引入封装、继承、多态等概念来建立一种对象层次结构，用于模拟公共行为的一个集合。不过OOP允许开发者定义纵向的关系，但并不适合定义横向的关系，例如日志功能。日志代码往往横向地散布在所有对象层次中，而与它对应的对象的核心功能毫无关系对于其他类型的代码，如安全性、异常处理和透明的持续性也都是如此，这种散布在各处的无关的代码被称为横切（cross cutting），在OOP设计中，它导致了大量代码的重复，而不利于各个模块的重用。

AOP技术恰恰相反，它利用一种称为"横切"的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其命名为"Aspect"，即切面。所谓"切面"，简单说就是那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块之间的耦合度，并有利于未来的可操作性和可维护性。

使用"横切"技术，AOP把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处基本相似，比如权限认证、日志、事物。AOP的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。

### 2.2 aop核心概念

1、横切关注点(对哪些方法进行切入)

对哪些方法进行拦截，拦截后怎么处理，这些关注点称之为横切关注点

2、切面（aspect,把原来糅杂在业务逻辑代码中的非业务代码抽取出来，把功能相同的放在一个类中形成一个切面）

类是对物体特征的抽象，切面就是对横切关注点的抽象

3、连接点（joinpoint）（需要切入的点）

被拦截到的点，因为Spring只支持方法类型的连接点，所以在Spring中连接点指的就是被拦截到的方法，实际上连接点还可以是字段或者构造器

4、切入点（pointcut）

对连接点进行拦截的定义

5、通知（advice）

所谓通知指的就是指拦截到连接点之后要执行的代码，通知分为前置、后置、异常、最终、环绕通知五类

6、目标对象

代理的目标对象

7、织入（weave）

将切面应用到目标对象并导致代理对象创建的过程

8、引入（introduction）

在不修改代码的前提下，引入可以在运行期为类动态地添加一些方法或字段

### 2.3 几种通知advice的使用

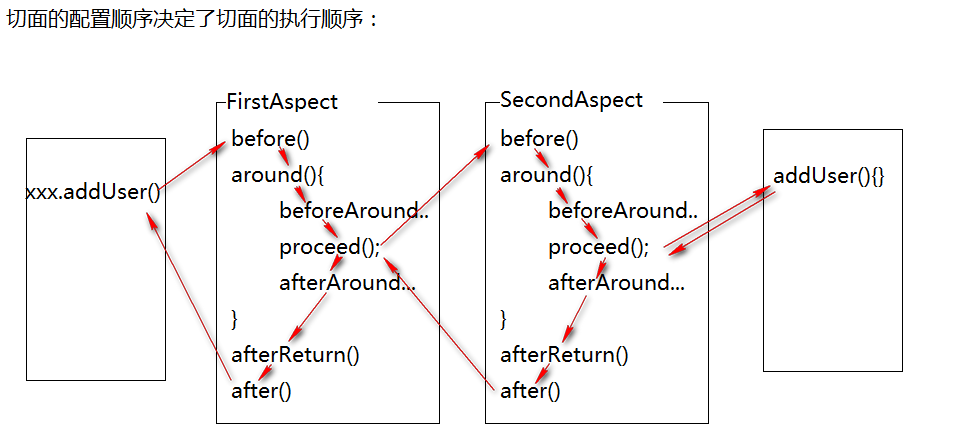
#### 通知的执行顺序：

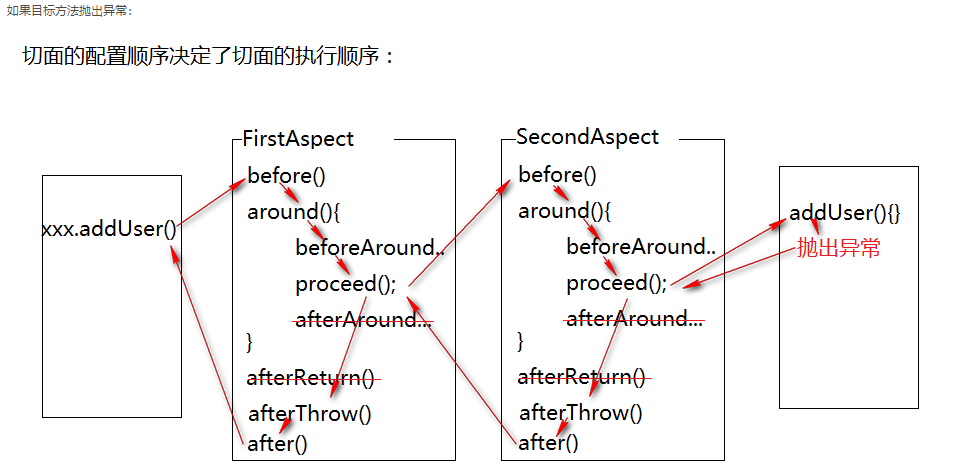
异常通知会将其他通知放入其try catch中，而后置通知在finally代码中，所以后置通知一定会执行

前置通知—》环绕通知的调用目标方法之前的代码—》目标方法—》环绕通知的调用目标方法之后的代码—》后置通知—》最终通知

前置通知—》环绕通知的调用目标方法之前的代码—》目标方法 抛出异常 –》异常通知—》后置通知

多切面下的执行顺序





#### 环绕通知：

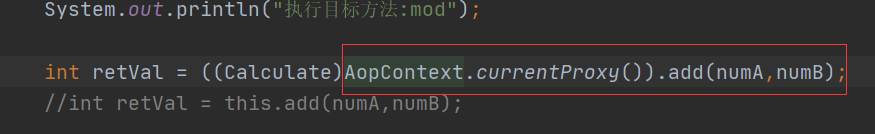
1）目标方法的调用由环绕通知决定，即你可以决定是否调用目标方法，而前置和后置通知是不能决定的，它们只是在方法的调用前后执行通知而已，即目标方法肯定是要执行的。joinPoint.proceed()就是执行目标方法的代码。

1. 环绕通知可以控制返回对象，即可以返回一个与目标对象完全不同的返回值。虽然这很危险，但是却可以做到。

### 2.4 jdk动态代理与cglib的区别

Jdk动态代理的目标对象只能是接口，是针对类实现代理，主要是对指定的类生成一个或多个子类，覆盖其中的方法（继承），在jdk1.8之前jdk动态代理的效率低于cglib，之后jdk动态代理的效率更高，jdk动态代理，目标类中的方法内部在调用一个目标方法，则内部调用不会走动态代理增强的逻辑，而cglib则会为每个目标方法走增强的逻辑，原因为cglib生成动态代理的原理是生成一个继承目标类的子类，子类调用增强的代码逻辑，最后再调用目标类的方法，此时目标类方法调用另一个方法，则会进入目标类的子类的代码逻辑，从而仍然可以增强，Jdk动态代理可以解决这一问题

方式@EnableAspectJAutoProxy(exposeProxy = true)配置类的属性exposeproxy设置为true，会将代理对象暴露在threadLocal本地变量中，通过这种方式调用，jdk调用使用的反射调用目标类方法



Jdk动态代理只是修改字节码，初始化效率高，而调用时使用的反射，效率低，cglib动态代理会为 目标类生成多个子类，初始化效率低，但是调用时是直接调用，调用的效率高

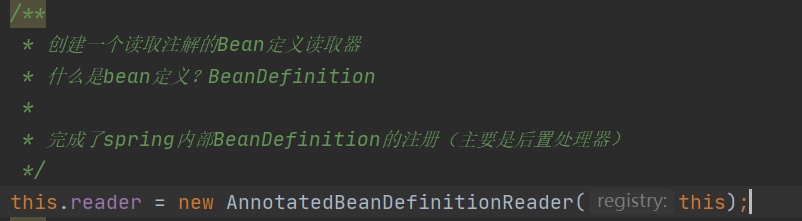
# 2、spring 源码

## 2.1 IOC

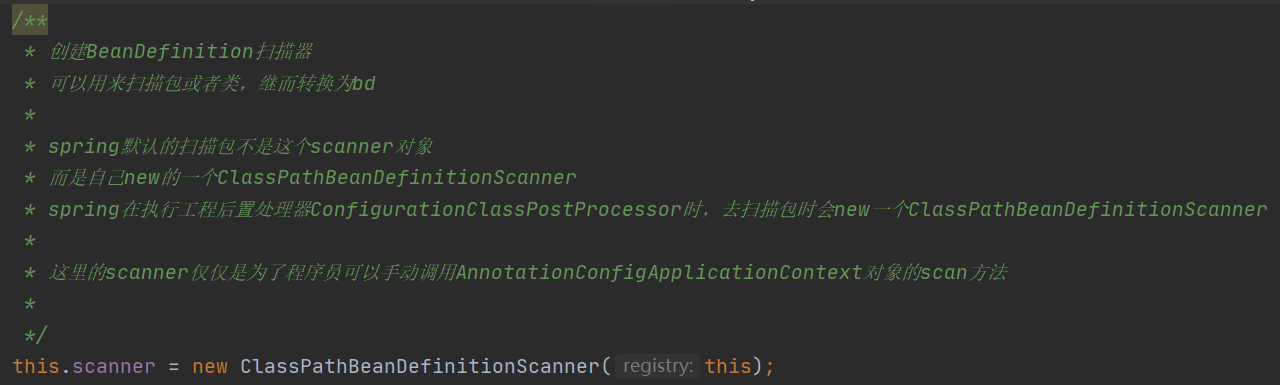
### 1.1 ioc容器初始化

以AnnotationConfigApplicationContext容器为例

1. 调用无参构造器this()

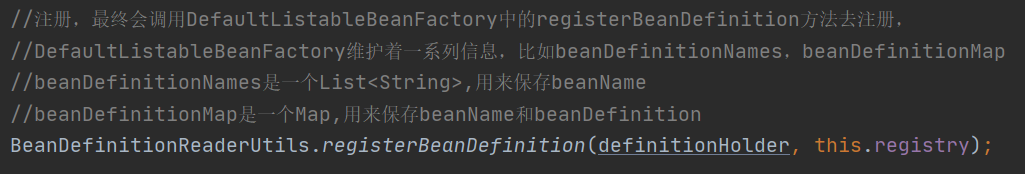


容器在构造方法中创建该读取器，注册后置处理器用于后面的bean定义注册



该扫描器主要用于客户自己扫描指定的配置类，容器会在后续扫描bean定义时重新创建扫描类

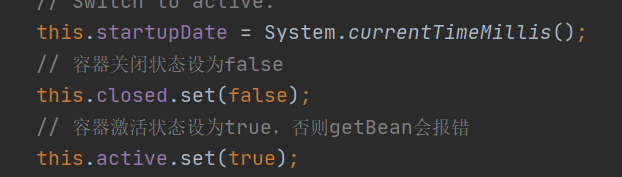
2、注册配置类register(annotatedClasses)



3、容器的refresh方法

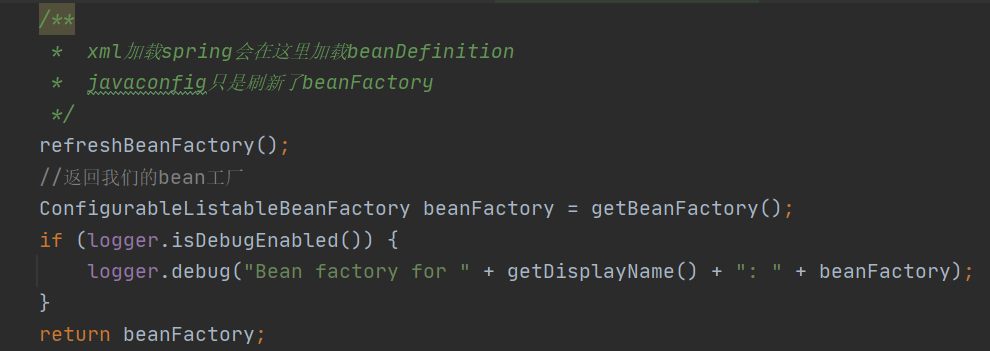
#### a、prepareRefresh()

刷新上下文环境，否则getBean会报错



#### b、obtainFreshBeanFactory()

给上下文创建bean工厂，不同的容器有不同的实现，此处创建DefaultListableBeanFactory工厂类，



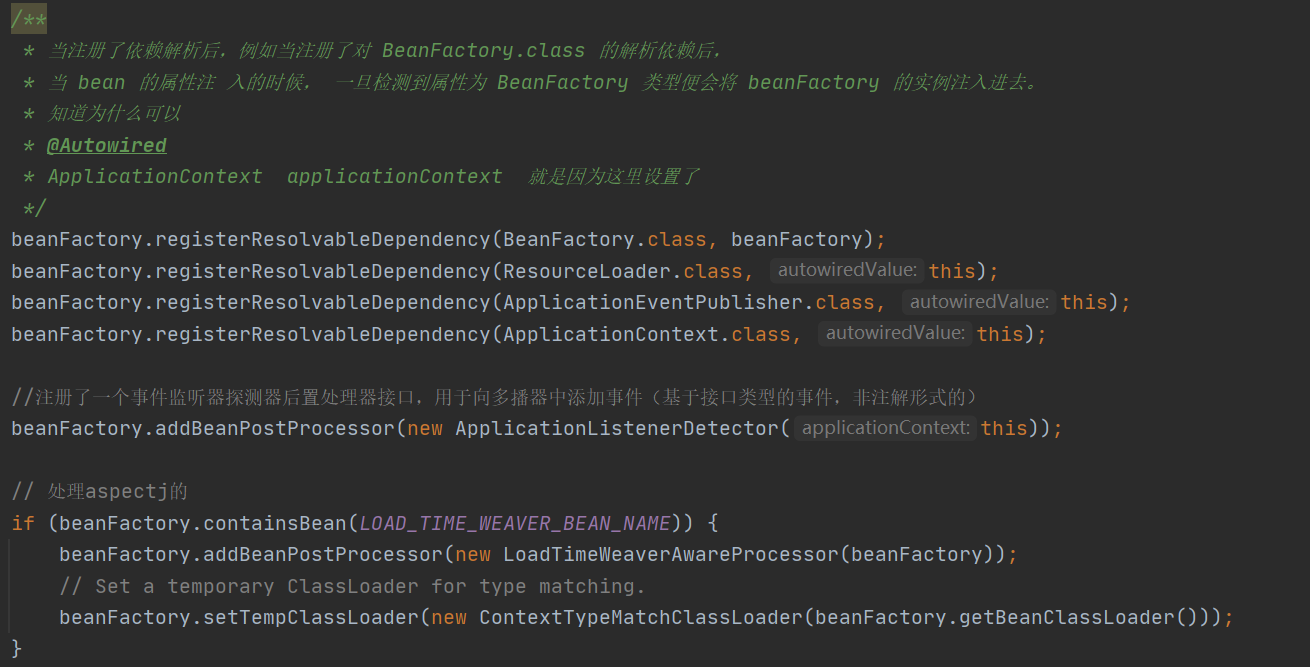
#### c、prepareBeanFactory(beanFactory)

对bean工厂填充属性





前四个bean注册到map中，key为类实例，可以直接通过autowired注解直接获取



#### d、postProcessBeanFactory(beanFactory)

开放式接口，留给子类实现

#### e、invokeBeanFactoryPostProcessors(beanFactory)

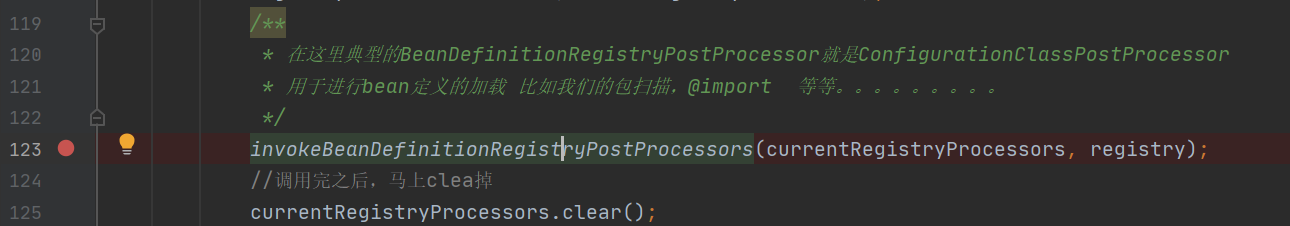
调用我们的bean工厂的后置处理器. 1. 会在此将class扫描成beanDefinition 2.bean工厂的后置处理器调用



**此处需要重点看ConfigurationClassPostProcessor解析配置类的后置处理器**

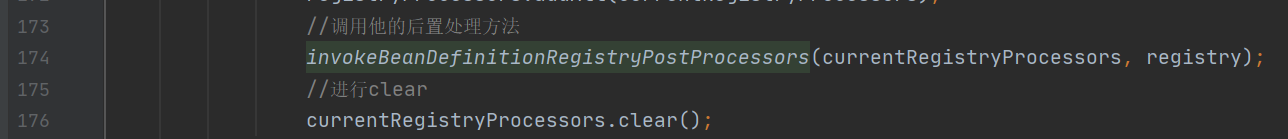
调用bean工厂BeanDefinitionRegistryPostProcessor的后置处理器一共有四次

1. 调用实现PriorityOrdered接口的bean工厂的后置处理器

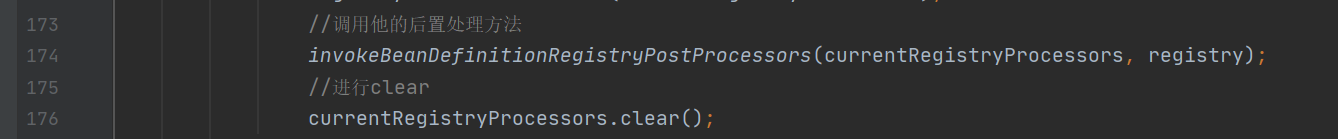


本次调用主要是扫描配置类，将配置类下的bean进行bean定义解析，此过程会将自定义的bean工厂的后置处理器加载进来，下面调用

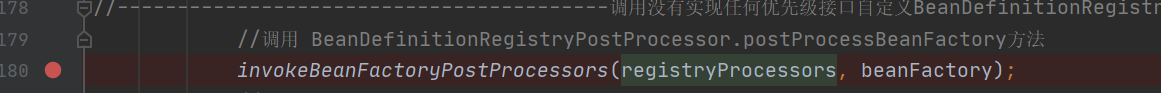
2、调用实现order接口的bean工厂的后置处理器



3、调用普通的后置处理器



4、调用实现自BeanFactoryPostProcessor的后置处理方法，用于修改bean定义



后面会获取所有实现BeanFactoryPostProcessor接口的实例且在上面没被调用的后置处理器

**下面重点看下ConfigurationClassPostProcessor解析配置类的后置处理器**

配置类上有configuration注解则解析时会给该类生成cjlab动态代理，这样配置类上某个bean调用生成另一个bean的方法时就会去容器中获取，因为会调用动态代理增强的代码，否则就会重新创建，如果用jdk自带的动态代理也会直接创建对象，因为也是直接调用原方法

#### f、registerBeanPostProcessors(beanFactory)

注册bean的后置处理器

在这个方法里会给bean工厂直接添加两个bean后置处理器，例如处理事件监听器的后置处理器 ApplicationListenerDetector，获取bean定义中的所有bean的后置处理器，后置处理器根据实现PriorityOrdered接口的部分拿出，在筛选出实现MergedBeanDefinitionPostProcessor的后置处理器放到集合中，其余的部分直接加入到bean工厂的bean后置处理器集合中，后面依次按照实现order的后置处理器与没有实现优先级接口的后置处理器加入到 bean工厂中，最后将实现MergedBeanDefinitionPostProcessor的后置处理器加入bean工厂

#### g、initMessageSource()

初始化国际资源处理器

如果配置文件中没有配置，则默认创建一个DelegatingMessageSource资源服务器，并将其放入单例缓存池中

#### h、initApplicationEventMulticaster()

创建多播器，首先通过getBean获取多播器，如果获取不到则创建一个

SimpleApplicationEventMulticaster

#### onRefresh()

留给子类实现，springboot也是从这个方法进行启动tomcat的

#### j、registerListeners()

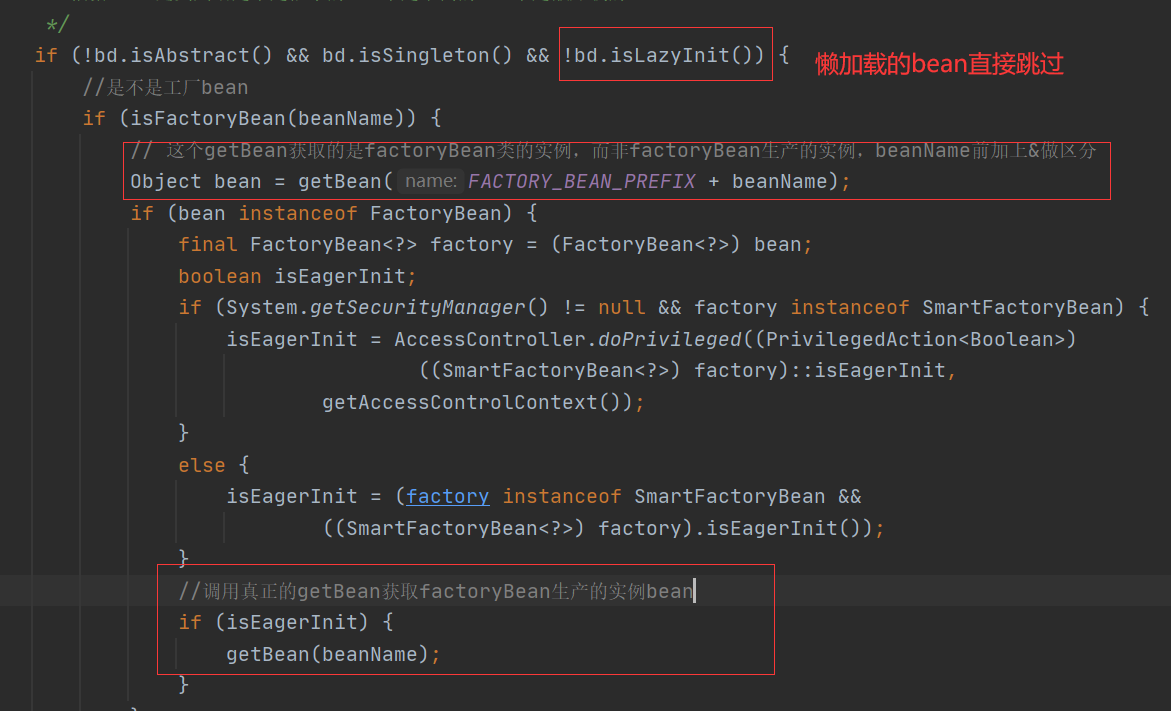
将事件监听器注册到多播器上，具体细节可以看spring event部分讲解

#### k、finishBeanFactoryInitialization(beanFactory)

实例化剩余的单例bean

如果bean是factoryBean类型的处理

首先获取factoryBean的类的实例，然后在通过getBean获取factoryBean生产的bean实例



调用getObject方法获取factoryBean生产的bean



#### l、finishRefresh()

最后容器刷新 发布刷新事件(Spring cloud也是从这里启动的)

其中LifecycleProcessor 负责管理ApplicationContext生命周期，此后置处理器的start方法启动的springcloud？

### 1.2 getBean

Spring bean的生命周期



1、实例化一个bean，也就是new；  
2、按照Spring上下文对Bean进行配置；  
3、如果Bean实现了BeanNameAware接口，会调用他的实现setBeaName(String)方法， 此处传递的就是Spring配置文件中的Bean id的值。

4、如果Bean实现了BeanFactoryAware接口，则会调用setBeanFactory(BeanFactory) 接口。

5、如果Bean实现了ApplicationCntextAware接口

setApplicationContext(ApplicationContext)方法，传入 Spring 上下文。  
6、如果Bean实现了BeanPostProcess接口，Spring将调用它们的

postProcessBeforeInitialization（预初始化）方法（作用是在Bean实例创建成功后对进行增强处理，如对Bean进行修改，增加某个功能）  
7、如果Bean实现了InitializingBean接口，Spring将调用它们的afterPropertiesSet方法，作用与在配置文件中对Bean使用init-method声明初始化的作用一样，都是在Bean的全部属性设置成功后执行的初始化方法。  
8、如果Bean实现了BeanPostProcess接口，Spring将调用它们

postProcessAfterInitialization（后初始化）方法（作用与6的一样，只不过6是在Bean初始化前执行的，而这个是在Bean初始化后执行的，时机不同 )  
9.经过以上的工作后，Bean将一直驻留在应用上下文中给应用使用，直到应用上下文被销毁

10.如果Bean实现了DispostbleBean接口，Spring将调用它的destory方法，作用与在配置文件中对Bean使用destory-method属性的作用一样，都是在Bean实例销毁前执行的方法。

InstantiationAwareBeanPostProcessor的postProcessBeforeInstantiation与postProcessAfterInstantiation方法CreateBean下resolveBeforeInstantiation会调用before方法，同时会缓存aop中的切面等对象，在doCreateBean下的populateBean调用after方法，

BeanPostProcessor接口 在doCreateBean下initializeBean(beanName,

exposedObject, mbd)下applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName)方法内调用，该方法内首先调用aware接口，然后调用BeanPostProcessor的before 方法，然后调用InitializingBean，然后调用BeanPostProcessor的after方法

initializeBean(beanName, exposedObject, mbd)下同时调用

invokeAwareMethods(beanName, bean)该方法依次调用BeanNameAware、

BeanClassLoaderAware、BeanFactoryAware

1、实例化之前调用第一次bean的后置处理器

createBean下第一次调用bean的后置处理器，实现

InstantiationAwareBeanPostProcessor接口的后置处理器，before与after方法都会调用，用于缓存aop的切面对象

2、实例化bean

doCreateBean -->createBeanInstance实例化bean

第二次调用bean的后置处理器

SmartInstantiationAwareBeanPostProcessor的

determineCandidateConstructors 来决定我们的构造方法

3、第三次调用bean的后置处理器

实现MergedBeanDefinitionPostProcessor的后置处理器，用于解析@autowired，@value，@Inject注解的bean属性

1. 将实例化的早期对象加入到三级缓存中

第四次调用后置处理器

getEarlyBeanReference方法内调用InstantiationAwareBeanPostProcessor的 getEarlyBeanReference方法，主要用于解决aop对象的循环依赖问题，通过后置处理 器来创建代理对象

1. populateBean(beanName, mbd, instanceWrapper)属性赋值

调用set方法进行属性赋值

a、第五次调用后置处理器

属性填充前会调用实现InstantiationAwareBeanPostProcessor的后置处理器的postProcessAfterInstantiation方法，允许用户在属性填充前定义bean的属性值

b、第六次调用后置处理器

调用实现InstantiationAwareBeanPostProcessor的后置处理器的

postProcessPropertyValues方法修改bean的属性值

6、initializeBean初始化

属性赋值完成之后进行初始化操作

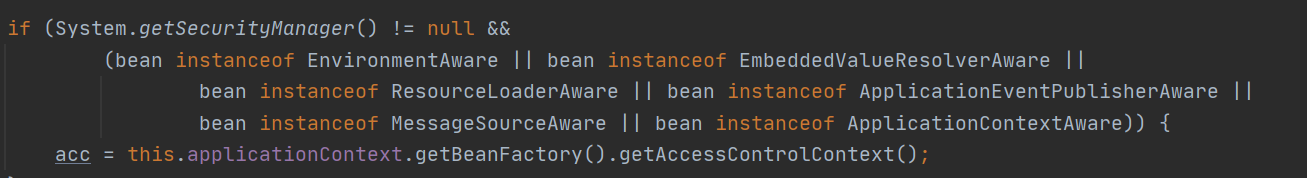
1. 调用aware接口

分别为BeanNameAware， BeanClassLoaderAware， BeanFactoryAware三个接 口

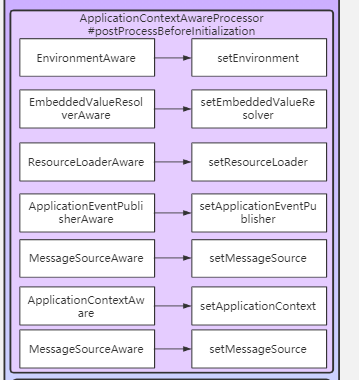
1. 第七次调用后置处理器

初始化前最后一次修改增强bean属性

调用所有的bean的后置处理器的postProcessorsBeforeInitialization方法 @PostCust注解的方法，本次调用会有ApplicationContextAwareProcessor后置处理器，在prepareBeanFactory时注册的，内的触发aware接口



各aware接口的作用



在ConfigurationClassPostProcessor的bean工厂的后置处理器中注册了

ImportAwareBeanPostProcessor后置处理器，用于实现ImportAware接口的bean调 用setImportMetadata方法

c、第八次调用后置处理器

postProcessAfterInitialization方法，该方法很重要，aop和事务都会在这里生存代理对 象我们AOP @EnableAspectJAutoProxy 为我们容器中导入了

AnnotationAwareAspectJAutoProxyCreator

我们事务注解@EnableTransactionManagement 为我们的容器导入了 InfrastructureAdvisorAutoProxyCreator

都是实现了我们的 BeanPostProcessor接口,

InstantiationAwareBeanPostProcessor,在这里实现的是BeanPostProcessor接口 的postProcessAfterInitialization来生成我们的代理对象

7、invokeInitMethods调用初始化方法

a、如果bean实现了InitializingBean接口则调用afterPropertiesSet()方法

b、调用bean的initMethod方法

c、注册销毁bean的接口，如果需要

## 2.2 aop

GetBean时调用第一次bean的后置处理器待用resolveBeforeInstantiation在初始化第一个自己配置的对象时会调用其后置处理器将切面信息解析放入缓存中

Aop的解析

<https://www.processon.com/view/link/5f1958a35653bb7fd24d0aad>

@EnableAspectJAutoProxy开启aop注解会引入AspectJAutoProxyRegistrar，该类实现了ImportBeanDefinitionRegistrar接口，用于注册bean定义，该类注册了AnnotationAwareAspectJAutoProxyCreator用于获取adviser，该类实现了bean后置处理器接口，在获取bean的第一次调用后置处理器时会触发，具体实现类为其父类AbstractAutoProxyCreator，同时解决aop对象循环依赖的getEarlyBeanReference方法也在该类内，获取advisor通过对aspect注解初筛加使用pointCut精晒两部获取到所有的advisor，然后放入缓存中

创建动态代理对象

<https://www.processon.com/view/link/5f1e93f25653bb7fd2549b7c>

在调用最后一个bean的后置处理器postProcessInstitation时创建动态代理对象，流程先从第一步的缓存中拿出所有的advistor，筛选出匹配的advister创建动态代理，比如是否是循环依赖的，不是目标类的，需要忽略的，创建动态代理时会根据proxyTargetClass变量判断使用哪种动态代理，该变量可以在配置类配置，在创建动态代理前有方法会判断目标类是否为接口也会为该类赋值，所以如果目标类不是接口则创建动态代理时制定走cglib逻辑，并且在该逻辑中会判断目标类是否为接口，接口的话会走jdk动态代理，如果目标类为接口且配置类没有配置proxyTargetClass变量，则直接走jdk动态代理逻辑

动态代理对象调用

<https://www.processon.com/view/link/5f4dd513e0b34d1abc735998>

得到所有的动态代理对象，并且多了一个intercept对象用来提供proceed方法，然后将所有的通知全部装换位拦截器然后放入到拦截器链中，异常通知跟后置通知继承自拦截器可直接转换，其他的通知通过适配器转换，调用的顺序：前置通知-》环绕通知调用目标方法之前的方法-》目标方法-》后置通知-》返回通知，异常通知的try-catch包含整个调用链，后置通知的调用在finally语句块内，所以后置通知一定会执行，具体调用是先通过proceed方法以递归的形式一层层往下调用，不反射调用方法，知道最后一层前置通知，则跳出递归一层层返回，返回时调用目标方法

## 2.3 事物

### 3.1 事物源码入口分析

Spring事务三大接口介绍

PlatformTransactionManager： （平台）事务管理器

TransactionDefinition： 事务定义信息(事务隔离级别、传播行为、超时、只读、回滚规则

TransactionStatus： 事务运行状态

二:Spring事务三大接口介绍

2.1）PlatformTransactionManager： （平台）事务管理器

Spring并不直接管理事务，而是提供了多种事务管理器 ，他们将事务管理的职责委托给Hibernate或者JTA等持久化机制所提供的相关平台框架的事务来实现。

Spring事务管理器的接口是：

org.springframework.transaction.PlatformTransactionManager ，

通过这个接口，Spring为各个平台如JDBC、Hibernate等都提供了对应的事务管理器，但是具体的实现就是各个平台自己的事情了。

public interface PlatformTransactionManager {

/\*\*

\*获取事物状态

\*/

TransactionStatus getTransaction(@Nullable TransactionDefinition definition) throws TransactionException;

/\*\*

\*事物提交

\*/

void commit(TransactionStatus status) throws TransactionException;

/\*\*

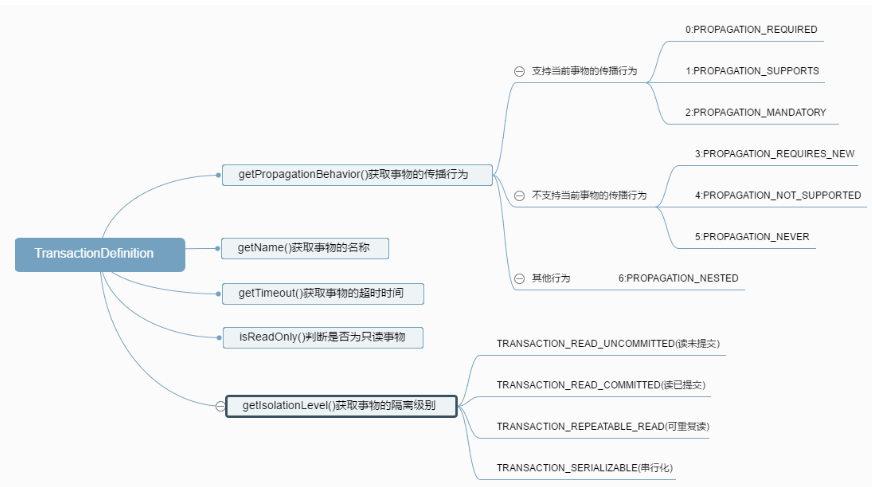
\*事物回滚

\*/

void rollback(TransactionStatus status) throws TransactionException;

}

2.2）TransactionDefinition： 事务定义信息(事务隔离级别、传播行为、超时、只读、回滚规则



TransactionDefinition接口中定义了5个方法以及一些表示事务属性的常量比如隔离级别、传播行为等等的常量。

我下面只是列出了TransactionDefinition接口中的方法而没有给出接口中定义的常量，该接口中的常量信息会在后面依次介绍到

public interface TransactionDefinition {

/\*\*

\* 支持当前事物，若当前没有事物就创建一个事物

\* \*/

int PROPAGATION\_REQUIRED = 0;

/\*\*

\* 如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行

\* \*/

int PROPAGATION\_SUPPORTS = 1;

/\*\*

\*如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常

\*/

int PROPAGATION\_MANDATORY = 2;

/\*\*

\*创建一个新的事务，如果当前存在事务，则把当前事务挂起

\*\*/

int PROPAGATION\_REQUIRES\_NEW = 3;

/\*\*

\* 以非事务方式运行，如果当前存在事务，则把当前事务挂起

\* \*/

int PROPAGATION\_NOT\_SUPPORTED = 4;

/\*\*

\* 以非事务方式运行，如果当前存在事务，则抛出异常。

\* \*/

int PROPAGATION\_NEVER = 5;

/\*\*

\* 表示如果当前正有一个事务在运行中，则该方法应该运行在 一个嵌套的事务中，

被嵌套的事务可以独立于封装事务进行提交或者回滚(保存点)，

如果封装事务不存在,行为就像 PROPAGATION\_REQUIRES NEW

\* \*/

int PROPAGATION\_NESTED = 6;

/\*\*

\*使用后端数据库默认的隔离级别，Mysql 默认采用的 REPEATABLE\_READ隔离级别 Oracle 默认采用的 READ\_COMMITTED隔离级别

\*/

int ISOLATION\_DEFAULT = -1;

/\*\*

\*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

\*/

int ISOLATION\_READ\_UNCOMMITTED = Connection.TRANSACTION\_READ\_UNCOMMITTED;

/\*\*

\*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

\*/

int ISOLATION\_READ\_COMMITTED = Connection.TRANSACTION\_READ\_COMMITTED;

/\*\*

\*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

\*/

int ISOLATION\_REPEATABLE\_READ = Connection.TRANSACTION\_REPEATABLE\_READ;

/\*\*

\*最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，

\* 也就是说，该级别可以防止脏读、不可重复读以及幻读。但是这将严重影响程序的性能通常情况下也不会用到该级别

\*/

int ISOLATION\_SERIALIZABLE = Connection.TRANSACTION\_SERIALIZABLE;

/\*\*

\*使用默认的超时时间

\*/

int TIMEOUT\_DEFAULT = -1;

/\*\*

\*获取事物的传播行为

\*/

int getPropagationBehavior();

/\*\*

\*获取事物的隔离级别

\*/

int getIsolationLevel();

/\*\*

\*返回事物的超时时间

\*/

int getTimeout();

/\*\*

\*返回当前是否为只读事物

\*/

boolean isReadOnly();

/\*\*

\*获取事物的名称

\*/

@Nullable

String getName();

}

TransactionDefinition接口中定义了5个方法以及一些表示事务属性的常量比如隔离级别、传播行为等等的常量。

我下面只是列出了TransactionDefinition接口中的方法而没有给出接口中定义的常量，该接口中的常量信息会在后面依次介绍到

public interface TransactionDefinition {

/\*\*

\* 支持当前事物，若当前没有事物就创建一个事物

\* \*/

int PROPAGATION\_REQUIRED = 0;

/\*\*

\* 如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行

\* \*/

int PROPAGATION\_SUPPORTS = 1;

/\*\*

\*如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常

\*/

int PROPAGATION\_MANDATORY = 2;

/\*\*

\*创建一个新的事务，如果当前存在事务，则把当前事务挂起

\*\*/

int PROPAGATION\_REQUIRES\_NEW = 3;

/\*\*

\* 以非事务方式运行，如果当前存在事务，则把当前事务挂起

\* \*/

int PROPAGATION\_NOT\_SUPPORTED = 4;

/\*\*

\* 以非事务方式运行，如果当前存在事务，则抛出异常。

\* \*/

int PROPAGATION\_NEVER = 5;

/\*\*

\* 表示如果当前正有一个事务在运行中，则该方法应该运行在 一个嵌套的事务中，

被嵌套的事务可以独立于封装事务进行提交或者回滚(保存点)，

如果封装事务不存在,行为就像 PROPAGATION\_REQUIRES NEW

\* \*/

int PROPAGATION\_NESTED = 6;

/\*\*

\*使用后端数据库默认的隔离级别，Mysql 默认采用的 REPEATABLE\_READ隔离级别 Oracle 默认采用的 READ\_COMMITTED隔离级别

\*/

int ISOLATION\_DEFAULT = -1;

/\*\*

\*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

\*/

int ISOLATION\_READ\_UNCOMMITTED = Connection.TRANSACTION\_READ\_UNCOMMITTED;

/\*\*

\*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

\*/

int ISOLATION\_READ\_COMMITTED = Connection.TRANSACTION\_READ\_COMMITTED;

/\*\*

\*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

\*/

int ISOLATION\_REPEATABLE\_READ = Connection.TRANSACTION\_REPEATABLE\_READ;

/\*\*

\*最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，

\* 也就是说，该级别可以防止脏读、不可重复读以及幻读。但是这将严重影响程序的性能通常情况下也不会用到该级别

\*/

int ISOLATION\_SERIALIZABLE = Connection.TRANSACTION\_SERIALIZABLE;

/\*\*

\*使用默认的超时时间

\*/

int TIMEOUT\_DEFAULT = -1;

/\*\*

\*获取事物的传播行为

\*/

int getPropagationBehavior();

/\*\*

\*获取事物的隔离级别

\*/

int getIsolationLevel();

/\*\*

\*返回事物的超时时间

\*/

int getTimeout();

/\*\*

\*返回当前是否为只读事物

\*/

boolean isReadOnly();

/\*\*

\*获取事物的名称

\*/

@Nullable

String getName();

}

2.3）TransactionStatus： 事务运行状态

TransactionStatus接口用来记录事务的状态 该接口定义了一组方法,用来获取或判断事务的相应状态信息.

PlatformTransactionManager.getTransaction(…) 方法返回一个 TransactionStatus 对象。返回的TransactionStatus 对象可能代表一个新的或已经存在的事务（如果在当前调用堆栈有一个符合条件的事物

public interface TransactionStatus extends SavepointManager, Flushable {

/\*\*

\* 是否为新事物

\* \*/

boolean isNewTransaction();

/\*\*

\*是否有保存点

\*/

boolean hasSavepoint();

/\*\*

\*设置为只回滚

\*/

void setRollbackOnly();

/\*\*

\*是否为只回滚

\*/

boolean isRollbackOnly();

/\*\*

\*属性

\*/

@Override

void flush();

/\*\*

\*判断当前事物是否已经完成

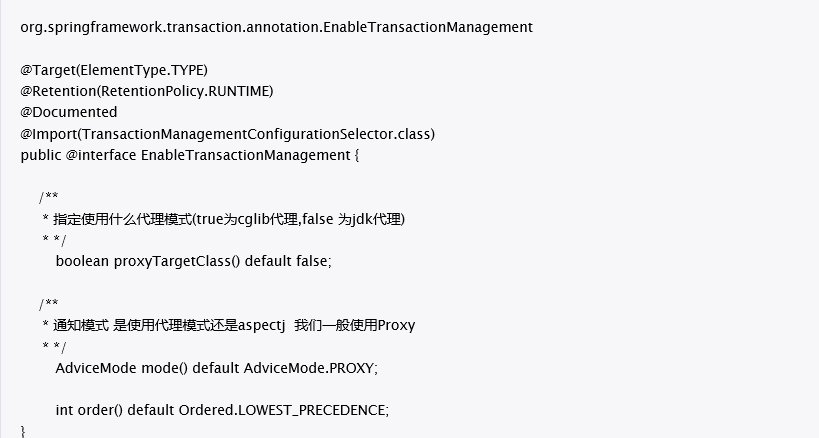
\*/

boolean isCompleted();

}

​

#### **@EnableTransactionManagement开始分析**



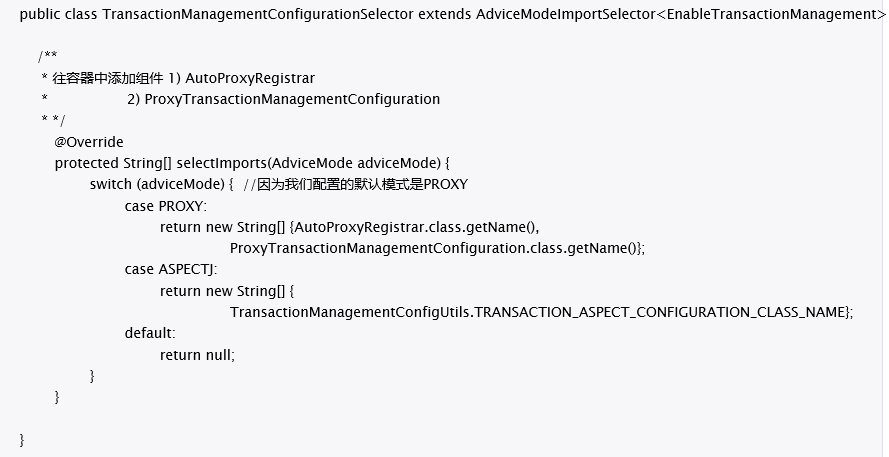
他通过@Import导入了TransactionManagementConfigurationSelector组件

 TransactionManagementConfigurationSelector源码分析

我们可以分析处向容器中导入了二个组件

1)AutoProxyRegistrar

2)ProxyTransactionManagementConfiguration



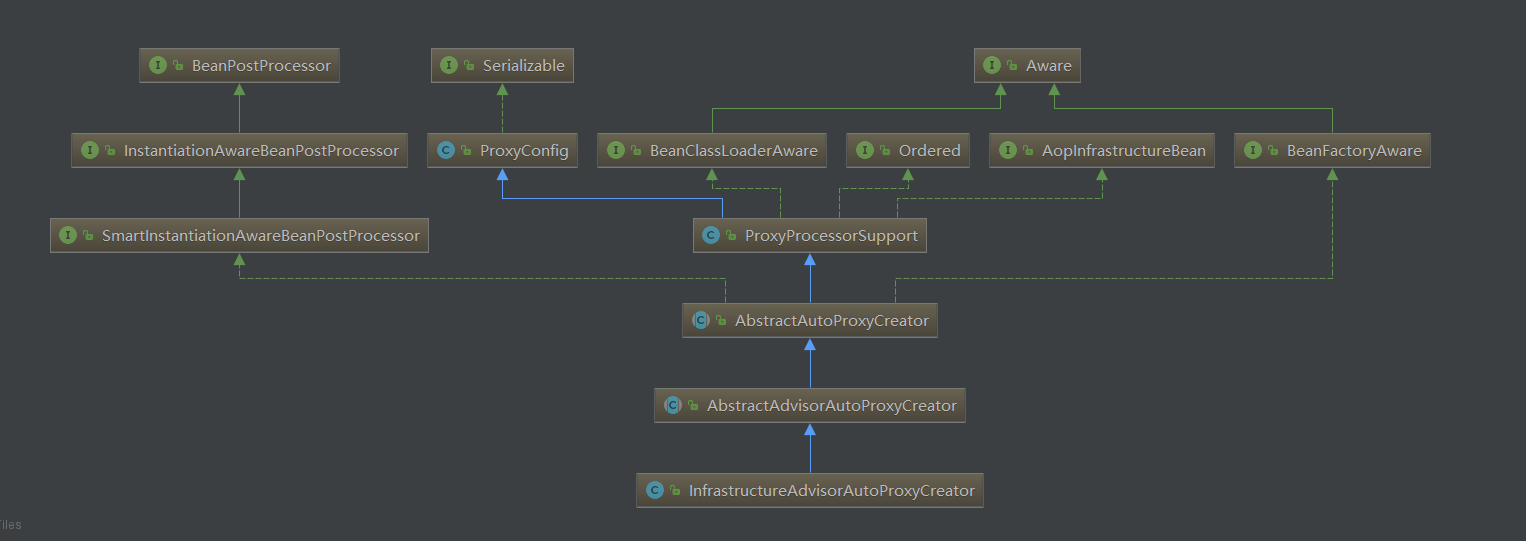
#### **AutoProxyRegistrar**

AutoProxyRegistrar为我们容器注册了一个InfrastructureAdvisorAutoProxyCreator组件





我们来看下InfrastructureAdvisorAutoProxyCreator继承图,有没有一点熟悉的味道?



所以我们来分析InfrastructureAdvisorAutoProxyCreator 实现了如下的接口

①:实现了Aware接口(具体代表 BeanFactoryAware接口)

做了什么事情）

a：把我们的BeanFacotry容器设置到了InfrastructureAdvisorAutoProxyCreator组件中去

b:  创建了一个advisorRetrievalHelper组件   增强器检索工具

AbstractAutoProxyCreator 实现了BeanFactoryAware接口，但是马上又被AbstractAdvisorAutoProxyCreator给重写了; 

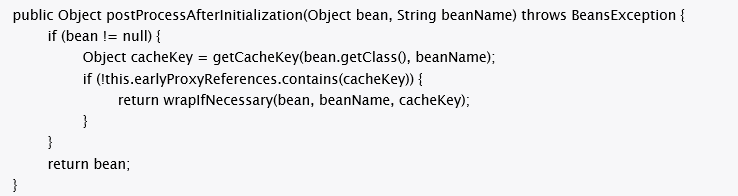
实现了我们的接口 InstantiationAwareBeanPostProcessor类型的后置处理器，为我们容器中做了什么事情

org.springframework.aop.framework.autoproxy.AbstractAutoProxyCreator#postProcessBeforeInitialization

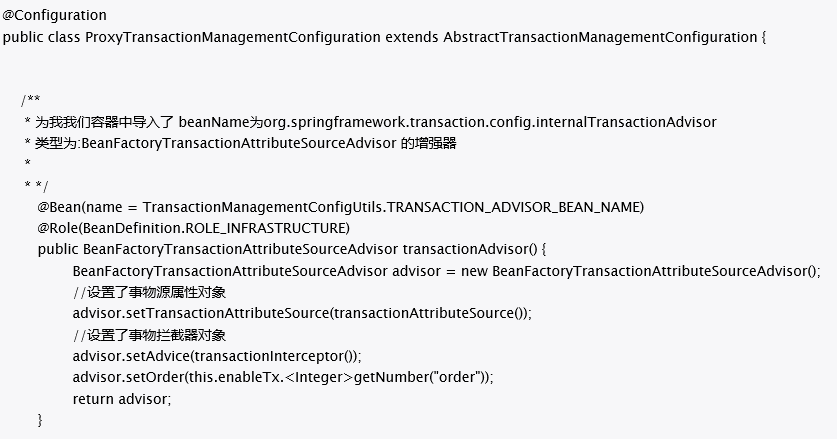
postProcessBeforeInstantiation方法



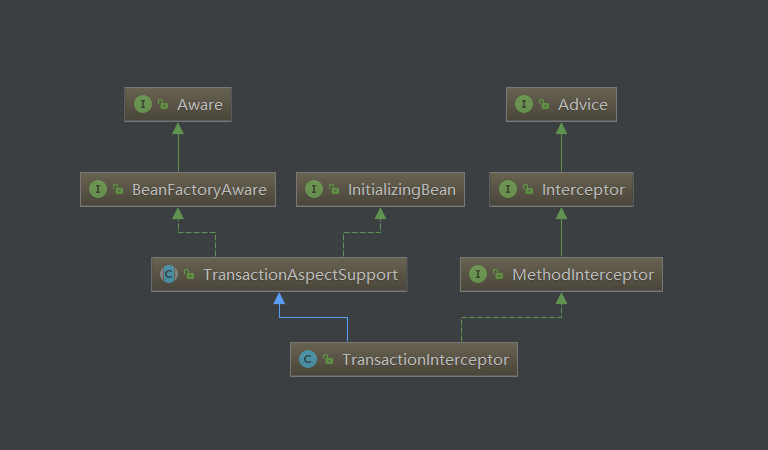
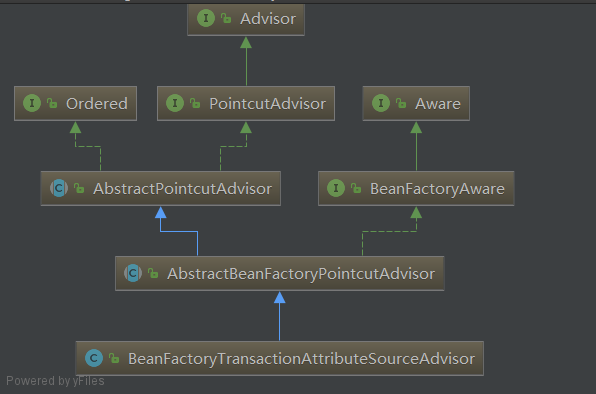
postProcessAfterInitialization 为我们做了事情

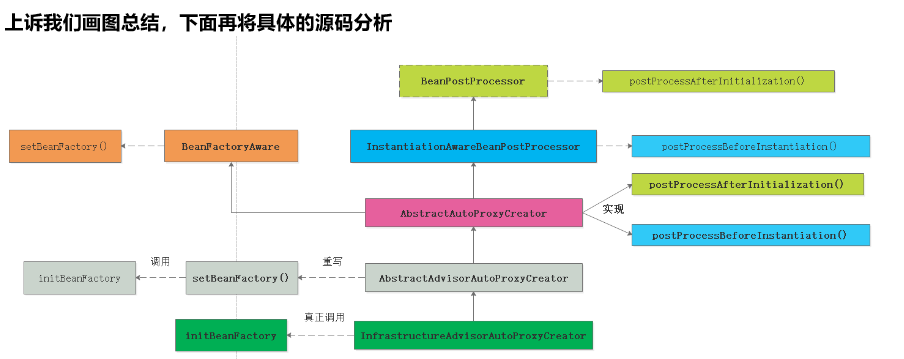


#### ProxyTransactionManagementConfiguration









## 2.4 spring event

在事件监听器中，ContextRefreshedEvent是在容器加载完成，所有bean创建成功后调用，可以使用该事件做项目启动后需要初始化调用的代码，同时实现SmartInitializingSingleton的接口也会在容器初始化所有的单例bean之后调用该接口的afterSingletonsInstantiated()方法这两种方式据可以用作项目初始化成功后触发自定义的逻辑业务

### 3.1 事件的使用

使用观察者模式

Spring事件体系包括三个组件：事件，事件监听器，事件广播器

**事件**（ApplicationEvent) 负责对应相应监听器 事件源发生某事件是特定事件监听器

被触发的原因。

**监听器(**ApplicationListener) 对应于观察者模式中的**观察者**。监听器监听特定事件,并

在内部定义了事件发生后的响应逻辑。

**事件发布器**（ApplicationEventMulticaster ）对应于观察者模式中的**被观察者/主**

**题， 负责通知观察者** 对外提供发布事件和增删事件监听器的接口,维护事件和事件监听

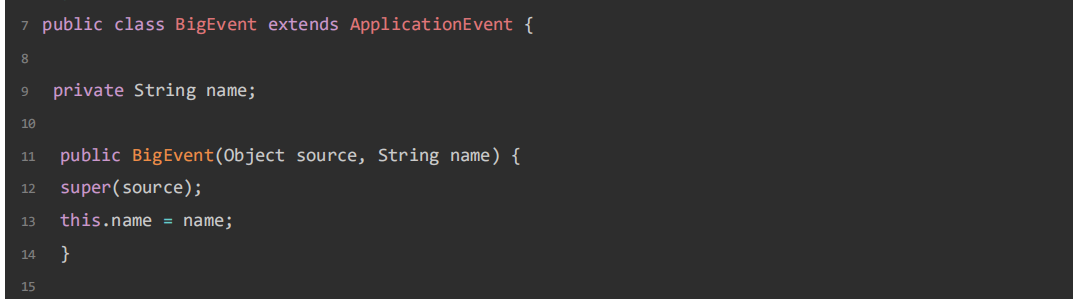
器之间的映射关系,并在事件发生时负责通知相关监听器。

事件可以有多个监听器，事件与监听可以同步执行也可异步



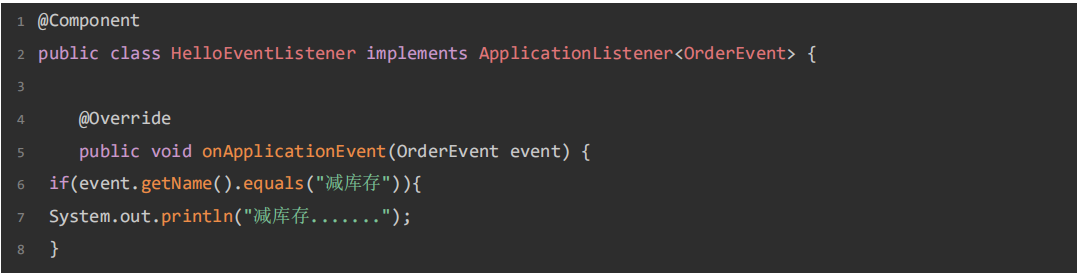
ContextRefreshedEvent是在容器加载完成，所有bean创建成功后调用，可以使用该事件做项目启动后需要初始化调用的代码

**自定义事件**

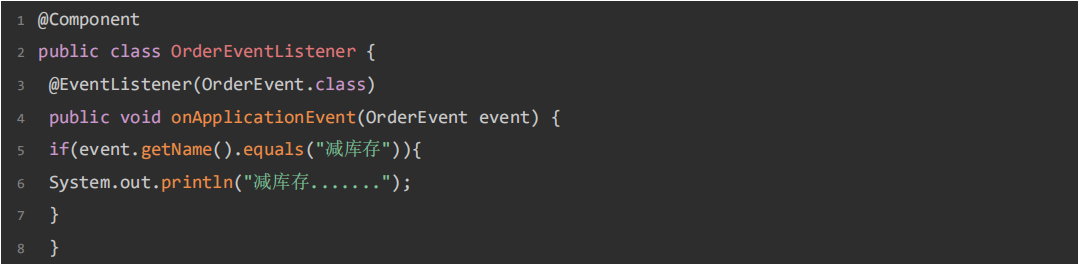


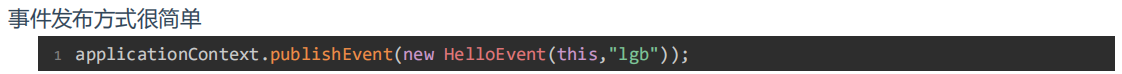
事件类是一种很简单的pojo，除了需要继承ApplicationEvent也没什么了，这个类有一个构造方法需要super。

事件监听器有两种实现，基于接口与注解

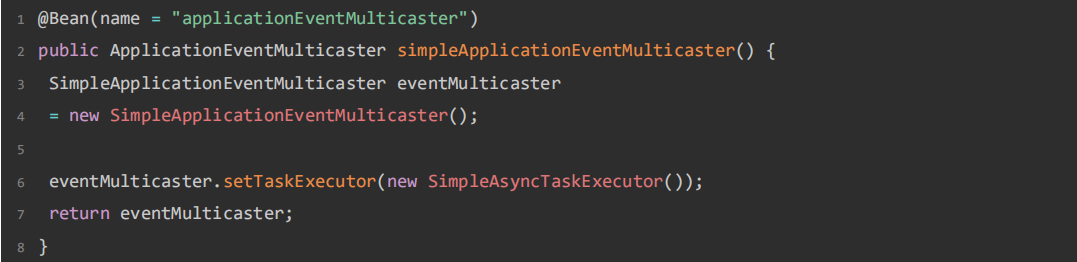


事件监听器需要实现ApplicationListener接口，这是个泛型接口，泛型类类型就是事件类型，其次需要是spring容器托管的bean，所以这里加了@component，只有一个方法，就是onApplicationEvent。

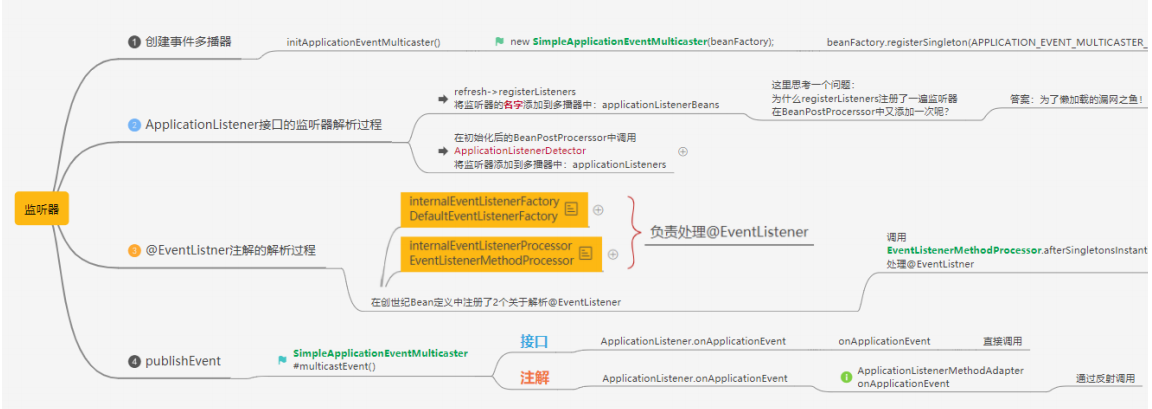




配置监听异步执行



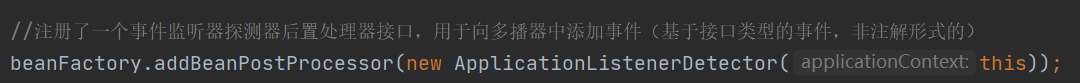
### 3.2 event 源码



在容器的refresh方法中，initApplicationEventMulticaster()方法创建多播器，用于存储自定义的事件监听器

**1、基于接口的事件监听器注册**

registerListeners()注册自定义的实现接口的事件监听器到多播器上，并且执行早期事件监听器，即在registerListeners()之前容器注册的事件监听器。

registerListeners()方法只处理接口类型事件监听器，该方法将只是将事件监听器的beanName注册到多播器上，之后调用的时候需要通过getBean获取实例，该方法调用后会触发后置处理器在prepareBeanFactory(beanFactory)中注册的该后置处理器，该后置处理器会创建接口类型的非懒加载的事件监听器，为完整的bean，之后直接调用监听器方法即可

这里同时使用两种方式获取监听器的原因？

registerListeners()方法可以获取到所有的事件监听器，但是监听器只有beanName，没有真正创建，需要getBean获取实例之后才能调用，而后置处理器不能注册懒加载的监听器

1. **基于注解的事件监听器注册**

容器初始化时注册了DefaultEventListenerFactory，与EventListenerMethodProcessor两个实例，EventListenerMethodProcessor实现SmartInitializingSingleton接口，会在容器创建完成所有的单例bean后调用EventListenerMethodProcessor里的afterSingletonsInstantiated()方法，在该方法内获取容器中所有的bean，循环查找出标注@EventListener注解的bean，通过DefaultEventListenerFactory工厂创建ApplicationListenerMethodAdapter对象，然后将其存储到多播器上