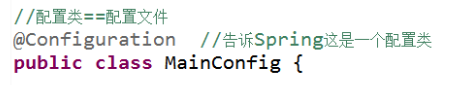
# spring注解驱动IOC

## @Configuration

修饰类 告诉spring这是一个配置类

spring的配置类==配置文件



### @Bean

给容器注册一个Bean,类型为返回值,id默认是方法名,id也可以在注解上进行配置value

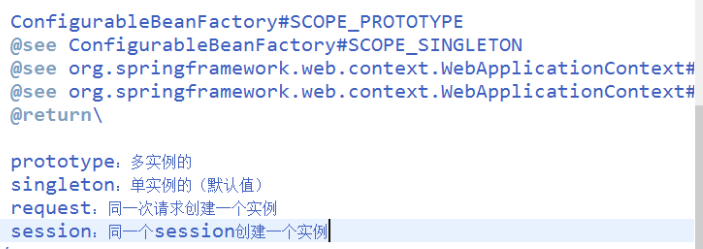


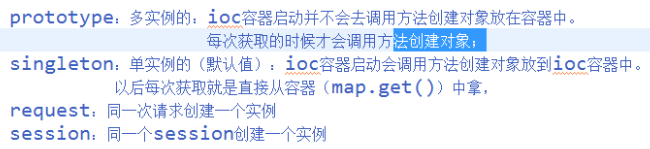
使用AnnotationConfigApplicationContext(配置类.class)来进行获取bean 

### @Scope

bean的作用范围 默认是单例的

可以通过Scope进行配置





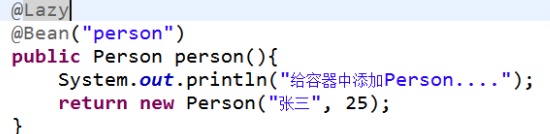
@Scope(**"prototype"**)

@Scope(**"singleton"**)

### @Lazy

懒加载,针对单例bean,默认在容器启动的时候创建对象

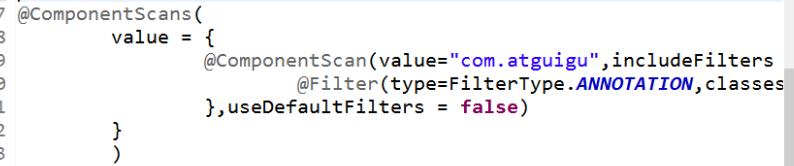
懒加载:容器启动不创建对象,第一次使用Bean的时候创建并初始化



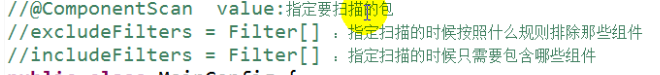
## @ComponentScan

如果java8可以使用重复注解

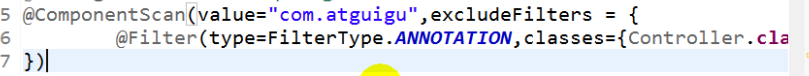
如果不是java8无法使用重复注解,可以使用ComponentScans里面是多个ComponentScan



包扫描:

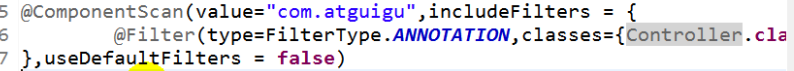


### excludeFilters

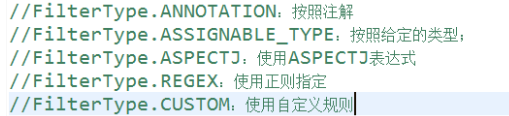


### includeFilters

注意要关闭使用默认filter



### FilterType规则:

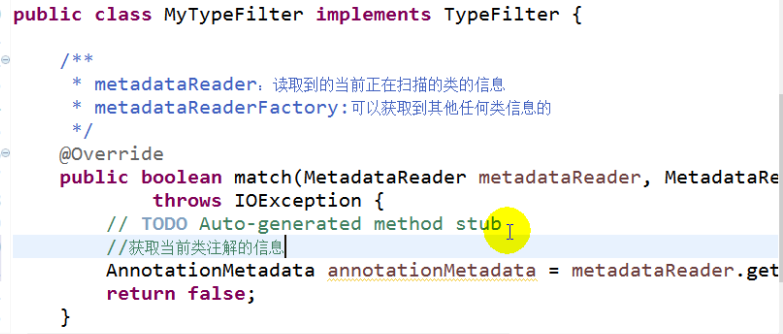


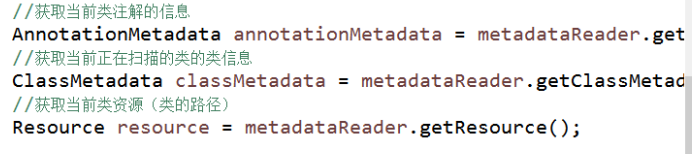
#### 自定义FilterType过滤规则

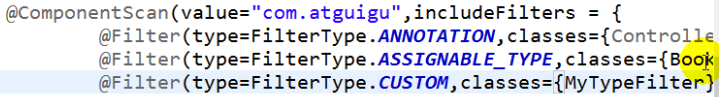
首先需要实现TypeFilter接口 并实现其中的方法

MatadataReader 读取的当前正在扫描的类的信息

MetadataReaderFactory: 可以获取到其他任何类的信息





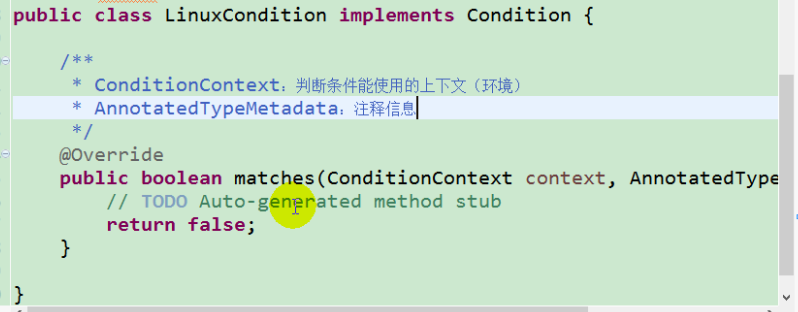
 

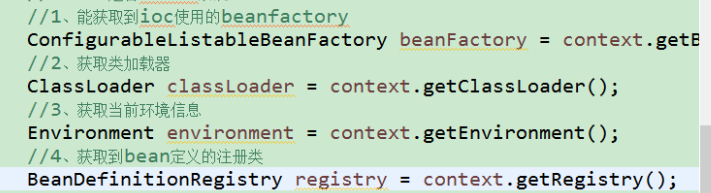
## @Conditional

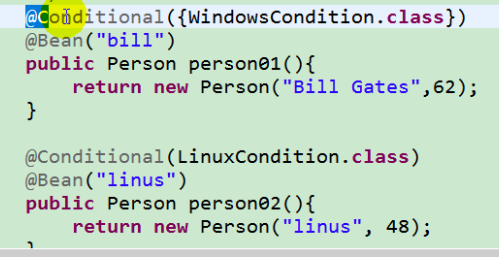
### 标注在方法上

按照一定的条件进行判断,满足条件给容器中注册bean

@Conditional中放的是Condition类型的数组

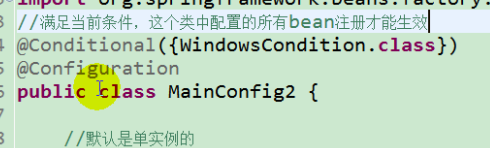






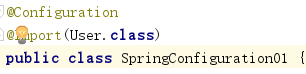
### 标注在类上:

满足当前的判断条件这个类上注册的bean才会生效



## @Import

快速给容器导入组件



id默认是全类名字符串



### ImportSelector

不止可以导入普通的class还可以导入ImportSelector类型的参数





## 给容器注册组件的方式:

### 1.包扫描+组件标注注解(@Controller,@Service,@Respository,@Component)

### 2.使用@Bean适用导入的第三方包

### 3.@Import 快速给容器中导入一个组件

#### (1)直接@import

@import({XXX.class,XXX.class})id默认是全类名

ApplicationContext application=**new** AnnotationConfigApplicationContext(SpringConfiguration01.**class**);  
User user=(User)application.getBean(**"com.langsin.bean.User"**);

#### (2)ImportSelector:

返回需要导入的组件的全类名数组

不止可以导入普通的class还可以导入ImportSelector类型的参数



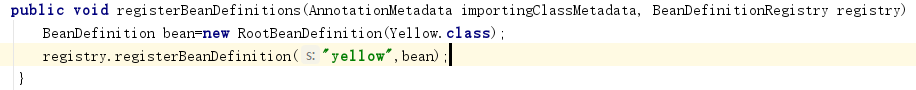


#### (3)ImportBeanDefinitionRegistrar

手动注册bean到容器中

这个可以指定bean的名称还有作用域等信息





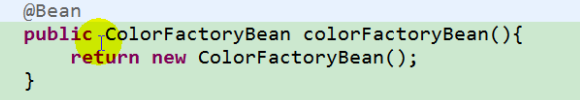


### 4使用spring提供的FactoryBean(工厂bean)

实现FactoryBean 并重写其中的方法



虽然注册的是ColorFactoryBena但是类型确实getObject创建的对象



但是当我们想获取ColorFactoryBean这个类型本身,需要通过&+id的方式获取



## Bean的生命周期

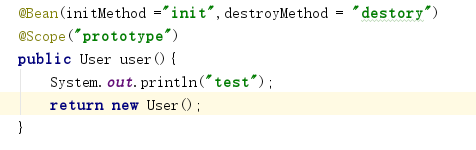
初始化:对象创建完成,并赋值好的时候,调用初始化方法

销毁:

单例:容器关闭的时候调用销毁方法

多例:容器不会对bean进行管理,也不会调用销毁方法

### 1.通过@Bean注解



### 2.通过让bean实现InitializingBean(定义初始化逻辑),DisposableBean(定义销毁逻辑)

### 3.使用JSR250

使用注解@PostConstruct,在bean创建完成并且属性赋值完成,来执行初始化方法

使用注解@PreDestory,在容器销毁bean之前通知我们进行清理工作

这两个注解都是用来修饰方法

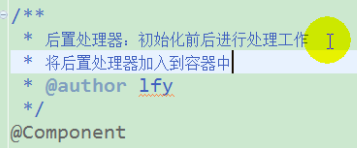
### 4.BeanPostProcessor接口,bean的后置处理器

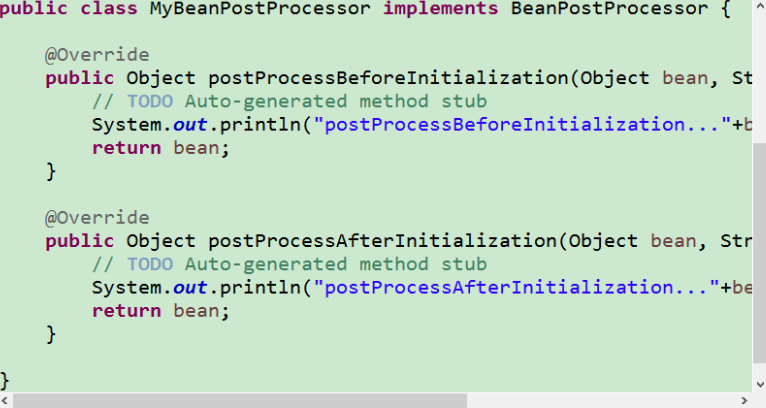
每一个bean都会经过后置处理器这两个方法

在bean初始化前后进行进行一些处理工作

postProcessBeforeInitialization:在初始化之前

postProcessAfterInitialization:在初始化之后

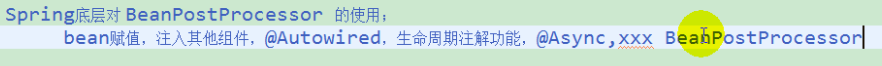




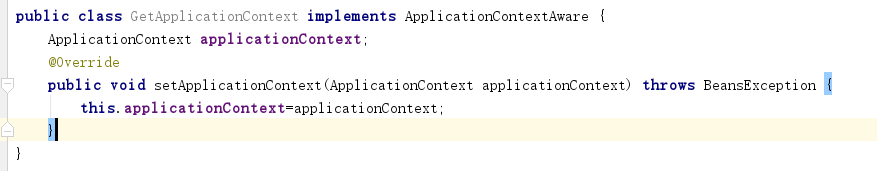
#### BeanPostProcessor的运行流程



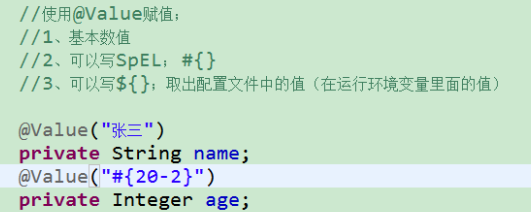
#### spring对BeanPostPrecessor的使用



可以通过继承ApplicationContextAware来获得ApplicationContext



## @Value



当我们需要使用${}取配置文件的值得时候需要配合注解@PropertySource来进行使用



## @PropertySource

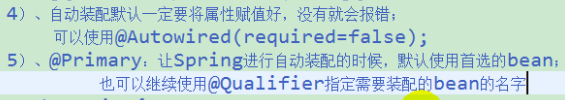
作用:读取外部配置文件中的k/v保存到运行的环境变量中



## 自动注入的注解

### @Autowired

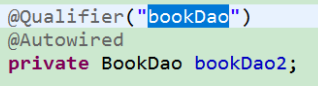




Autowired的属性中有一个required 可以指定是否必须装配 默认true

### @Qualifier(“id”)

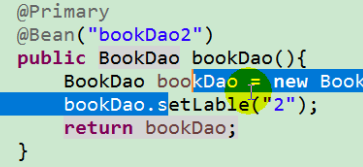
配合Autowired使用可以明确指定使用哪个id的组件来进行自动装配



### @Primary

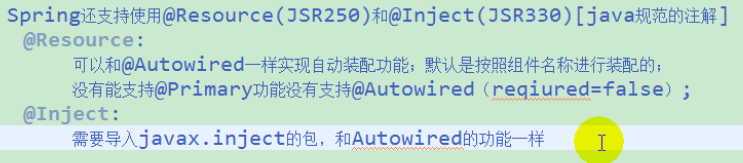
当存在多个相同类型的bean时指定首选装配哪个bean

如果存在@Qualifier明确指定使用哪个,那么就使用明确指定的id来获取



### @Resource 和@Inject(需要导包)

@Resource(name=””) name可以省略



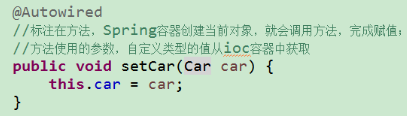


### @Autowired

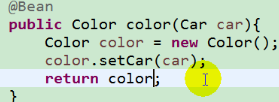
可以标注在属性 方法 构造器 参数上,都是从容器中获得组件

#### 标注在方法

spring容器创建当前对象就会调用方法完成赋值方法使用的参数,自定义类型的值从ioc容器中获取



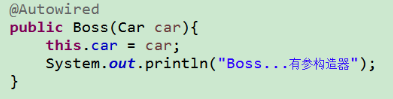
当使用@Bean的时候参数会自动从容器中获取可以不用写@Autowired,也可以在参数上自动获取



#### 标注在有参构造器

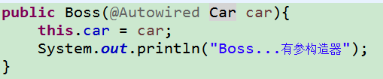
构造器要用的组件也是从容器中获取

如果组件只有一个有参构造器,@Autowired可以省略,参数的组件还是可以自动获取



#### 标注在参数上:

都是从容器中获取参数组件的值



## 使用spring底层的组件

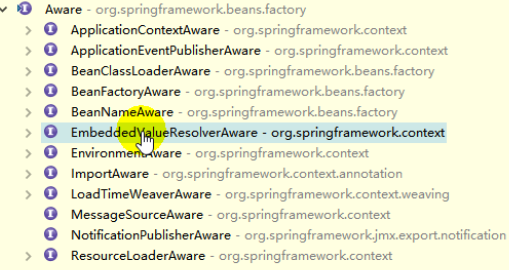
自定义的组件想要使用spring底层的一些组件例如:ApplicationContext,BeanFactory等等

如何使用呢:

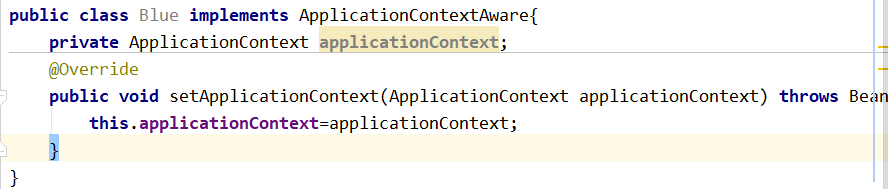
自定义的组件通过实现XXXAware:在创建对象的时候,会调用接口规定的方法注入相关的组件,把spring底层的一些组件注入到自定义的bean中

运行机制:XXXAware都会有相应的XXXAwareProcessor后置处理器来进行处理





### ApplicationContextAware

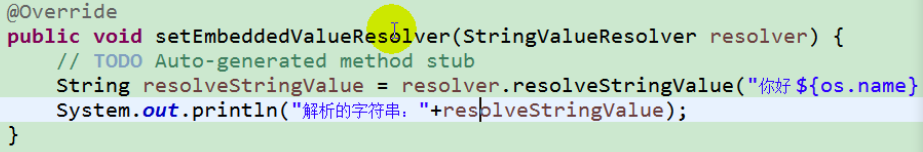


### BeanAware

可以获得在spring ioc容器中的名字

### EmbeddedValueResolverAware

可以解析特殊字符的组件 ${os.name} 可以取出运行环境中os.name名称的字符串



## @Profile

可以修饰配置类(只有指定的环境的时候整个配置类才能全部生效) 修饰方法

指定组件在哪个环境的情况下才能被注册到容器中,不指定的时候任何情况下都能注册这个组件当我们用@Profile修饰的时候,只有和当前这个环境相对应的@Profile修饰的组件才能激活

加了环境标识的bean只有这个环境被激活的时候才能注册到容器中,默认是default环境

如果没有指定环境的bean都会被加载

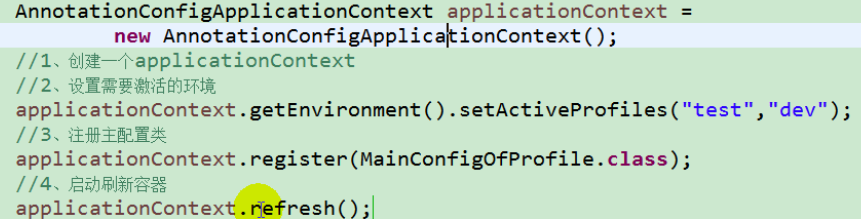
如何切换运行环境:

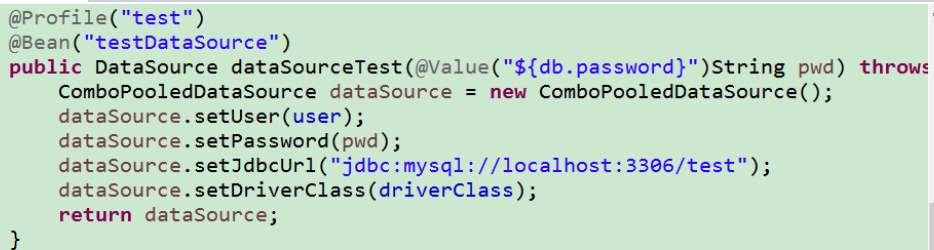
1. 命令行的方式设置 vm arguments:

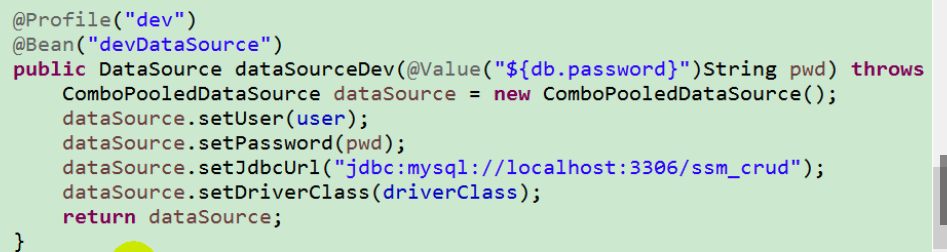
-Dspring-profiles.active=test

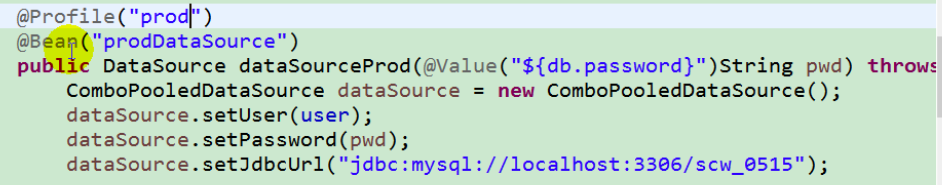
-Dspring-profiles.active=dev

2. 当创建ApplicatioContext的时候不使用有参的构造器,而是自己设置如何加载启动容器,可以设置多个运行环境





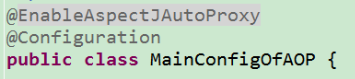




## @EnableAspectjAutoProxy

开启基于注解的aop模式

开启aop动态代理来代替xml中<aop:aspectj-autoproxy/>

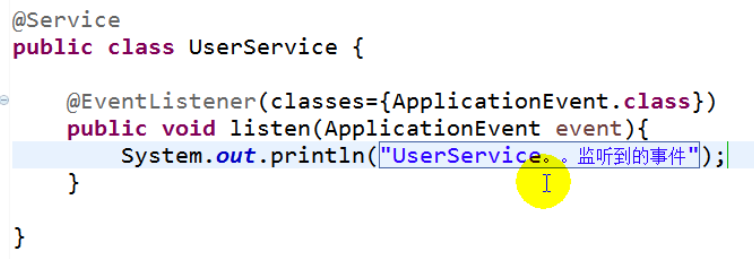


## @Aspectj

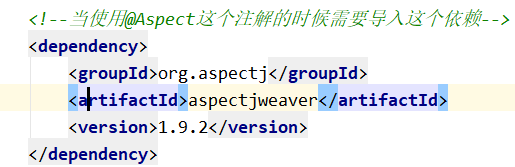
标注这是一个切面类

## @EventListener

标注在方法上可以用来监听指定事件



# spring注解驱动AOP



## 文字说明

|  |
| --- |
| /\*\*  \* AOP：【动态代理】  \* 指在程序运行期间动态的将某段代码切入到指定方法指定位置进行运行的编程方式；  \*  \* 1、导入aop模块；Spring AOP：(spring-aspects)  \* 2、定义一个业务逻辑类（MathCalculator）；在业务逻辑运行的时候将日志进行打印（方法之前、方法运行结束、方法出现异常，xxx）  \* 3、定义一个日志切面类（LogAspects）：切面类里面的方法需要动态感知MathCalculator.div运行到哪里然后执行；  \* 通知方法：  \* 前置通知(@Before)：logStart：在目标方法(div)运行之前运行  \* 后置通知(@After)：logEnd：在目标方法(div)运行结束之后运行（无论方法正常结束还是异常结束）  \* 返回通知(@AfterReturning)：logReturn：在目标方法(div)正常返回之后运行  \* 异常通知(@AfterThrowing)：logException：在目标方法(div)出现异常以后运行  \* 环绕通知(@Around)：动态代理，手动推进目标方法运行（joinPoint.procced()）  \* 4、给切面类的目标方法标注何时何地运行（通知注解）；  \* 5、将切面类和业务逻辑类（目标方法所在类）都加入到容器中;  \* 6、必须告诉Spring哪个类是切面类(给切面类上加一个注解：@Aspect)  \* [7]、给配置类中加 @EnableAspectJAutoProxy 【开启基于注解的aop模式】  \* 在Spring中很多的 @EnableXXX;  \*  \* 三步：  \* 1）、将业务逻辑组件和切面类都加入到容器中；告诉Spring哪个是切面类（@Aspect）  \* 2）、在切面类上的每一个通知方法上标注通知注解，告诉Spring何时何地运行（切入点表达式）  \* 3）、开启基于注解的aop模式；@EnableAspectJAutoProxy  \*  \* AOP原理：【看给容器中注册了什么组件，这个组件什么时候工作，这个组件的功能是什么？】  \* @EnableAspectJAutoProxy；  \* 1、@EnableAspectJAutoProxy是什么？  \* @Import(AspectJAutoProxyRegistrar.class)：给容器中导入AspectJAutoProxyRegistrar  \* 利用AspectJAutoProxyRegistrar自定义给容器中注册bean；BeanDefinetion  \* internalAutoProxyCreator=AnnotationAwareAspectJAutoProxyCreator  \*  \* 给容器中注册一个AnnotationAwareAspectJAutoProxyCreator；  \*  \* 2、 AnnotationAwareAspectJAutoProxyCreator：  \* AnnotationAwareAspectJAutoProxyCreator  \* ->AspectJAwareAdvisorAutoProxyCreator  \* ->AbstractAdvisorAutoProxyCreator  \* ->AbstractAutoProxyCreator  \* implements SmartInstantiationAwareBeanPostProcessor, BeanFactoryAware  \* 关注后置处理器（在bean初始化完成前后做事情）、自动装配BeanFactory  \*  \* AbstractAutoProxyCreator.setBeanFactory()  \* AbstractAutoProxyCreator.有后置处理器的逻辑；  \*  \* AbstractAdvisorAutoProxyCreator.setBeanFactory()-》initBeanFactory()  \*  \* AnnotationAwareAspectJAutoProxyCreator.initBeanFactory()  \*  \*  \* 流程：  \* 1）、传入配置类，创建ioc容器  \* 2）、注册配置类，调用refresh（）刷新容器；  \* 3）、registerBeanPostProcessors(beanFactory);注册bean的后置处理器来方便拦截bean的创建；  \* 1）、先获取ioc容器已经定义了的需要创建对象的所有BeanPostProcessor  \* 2）、给容器中加别的BeanPostProcessor  \* 3）、优先注册实现了PriorityOrdered接口的BeanPostProcessor；  \* 4）、再给容器中注册实现了Ordered接口的BeanPostProcessor；  \* 5）、注册没实现优先级接口的BeanPostProcessor；  \* 6）、注册BeanPostProcessor，实际上就是创建BeanPostProcessor对象，保存在容器中；  \* 创建internalAutoProxyCreator的BeanPostProcessor【AnnotationAwareAspectJAutoProxyCreator】  \* 1）、创建Bean的实例  \* 2）、populateBean；给bean的各种属性赋值  \* 3）、initializeBean：初始化bean；  \* 1）、invokeAwareMethods()：处理Aware接口的方法回调  \* 2）、applyBeanPostProcessorsBeforeInitialization()：应用后置处理器的postProcessBeforeInitialization（）  \* 3）、invokeInitMethods()；执行自定义的初始化方法  \* 4）、applyBeanPostProcessorsAfterInitialization()；执行后置处理器的postProcessAfterInitialization（）；  \* 4）、BeanPostProcessor(AnnotationAwareAspectJAutoProxyCreator)创建成功；--》aspectJAdvisorsBuilder  \* 7）、把BeanPostProcessor注册到BeanFactory中；  \* beanFactory.addBeanPostProcessor(postProcessor);  \* =======以上是创建和注册AnnotationAwareAspectJAutoProxyCreator的过程========  \*  \* AnnotationAwareAspectJAutoProxyCreator => InstantiationAwareBeanPostProcessor  \* 4）、finishBeanFactoryInitialization(beanFactory);完成BeanFactory初始化工作；创建剩下的单实例bean  \* 1）、遍历获取容器中所有的Bean，依次创建对象getBean(beanName);  \* getBean->doGetBean()->getSingleton()->  \* 2）、创建bean  \* 【AnnotationAwareAspectJAutoProxyCreator在所有bean创建之前会有一个拦截，InstantiationAwareBeanPostProcessor，会调用postProcessBeforeInstantiation()】  \* 1）、先从缓存中获取当前bean，如果能获取到，说明bean是之前被创建过的，直接使用，否则再创建；  \* 只要创建好的Bean都会被缓存起来  \* 2）、createBean（）;创建bean；  \* AnnotationAwareAspectJAutoProxyCreator 会在任何bean创建之前先尝试返回bean的实例  \* 【BeanPostProcessor是在Bean对象创建完成初始化前后调用的】  \* 【InstantiationAwareBeanPostProcessor是在创建Bean实例之前先尝试用后置处理器返回对象的】  \* 1）、resolveBeforeInstantiation(beanName, mbdToUse);解析BeforeInstantiation  \* 希望后置处理器在此能返回一个代理对象；如果能返回代理对象就使用，如果不能就继续  \* 1）、后置处理器先尝试返回对象；  \* bean = applyBeanPostProcessorsBeforeInstantiation（）：  \* 拿到所有后置处理器，如果是InstantiationAwareBeanPostProcessor;  \* 就执行postProcessBeforeInstantiation  \* if (bean != null) {  bean = applyBeanPostProcessorsAfterInitialization(bean, beanName);  }  \*  \* 2）、doCreateBean(beanName, mbdToUse, args);真正的去创建一个bean实例；和3.6流程一样；  \* 3）、  \*  \*  \* AnnotationAwareAspectJAutoProxyCreator【InstantiationAwareBeanPostProcessor】 的作用：  \* 1）、每一个bean创建之前，调用postProcessBeforeInstantiation()；  \* 关心MathCalculator和LogAspect的创建  \* 1）、判断当前bean是否在advisedBeans中（保存了所有需要增强bean）  \* 2）、判断当前bean是否是基础类型的Advice、Pointcut、Advisor、AopInfrastructureBean，  \* 或者是否是切面（@Aspect）  \* 3）、是否需要跳过  \* 1）、获取候选的增强器（切面里面的通知方法）【List<Advisor> candidateAdvisors】  \* 每一个封装的通知方法的增强器是 InstantiationModelAwarePointcutAdvisor；  \* 判断每一个增强器是否是 AspectJPointcutAdvisor 类型的；返回true  \* 2）、永远返回false  \*  \* 2）、创建对象  \* postProcessAfterInitialization；  \* return wrapIfNecessary(bean, beanName, cacheKey);//包装如果需要的情况下  \* 1）、获取当前bean的所有增强器（通知方法） Object[] specificInterceptors  \* 1、找到候选的所有的增强器（找哪些通知方法是需要切入当前bean方法的）  \* 2、获取到能在bean使用的增强器。  \* 3、给增强器排序  \* 2）、保存当前bean在advisedBeans中；  \* 3）、如果当前bean需要增强，创建当前bean的代理对象；  \* 1）、获取所有增强器（通知方法）  \* 2）、保存到proxyFactory  \* 3）、创建代理对象：Spring自动决定  \* JdkDynamicAopProxy(config);jdk动态代理；  \* ObjenesisCglibAopProxy(config);cglib的动态代理；  \* 4）、给容器中返回当前组件使用cglib增强了的代理对象；  \* 5）、以后容器中获取到的就是这个组件的代理对象，执行目标方法的时候，代理对象就会执行通知方法的流程；  \*  \*  \* 3）、目标方法执行 ；  \* 容器中保存了组件的代理对象（cglib增强后的对象），这个对象里面保存了详细信息（比如增强器，目标对象，xxx）；  \* 1）、CglibAopProxy.intercept();拦截目标方法的执行  \* 2）、根据ProxyFactory对象获取将要执行的目标方法拦截器链；  \* List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);  \* 1）、List<Object> interceptorList保存所有拦截器 5  \* 一个默认的ExposeInvocationInterceptor 和 4个增强器；  \* 2）、遍历所有的增强器，将其转为Interceptor；  \* registry.getInterceptors(advisor);  \* 3）、将增强器转为List<MethodInterceptor>；  \* 如果是MethodInterceptor，直接加入到集合中  \* 如果不是，使用AdvisorAdapter将增强器转为MethodInterceptor；  \* 转换完成返回MethodInterceptor数组；  \*  \* 3）、如果没有拦截器链，直接执行目标方法;  \* 拦截器链（每一个通知方法又被包装为方法拦截器，利用MethodInterceptor机制）  \* 4）、如果有拦截器链，把需要执行的目标对象，目标方法，  \* 拦截器链等信息传入创建一个 CglibMethodInvocation 对象，  \* 并调用 Object retVal = mi.proceed();  \* 5）、拦截器链的触发过程;  \* 1)、如果没有拦截器执行执行目标方法，或者拦截器的索引和拦截器数组-1大小一样（指定到了最后一个拦截器）执行目标方法；  \* 2)、链式获取每一个拦截器，拦截器执行invoke方法，每一个拦截器等待下一个拦截器执行完成返回以后再来执行；  \* 拦截器链的机制，保证通知方法与目标方法的执行顺序；  \*  \* 总结：  \* 1）、 @EnableAspectJAutoProxy 开启AOP功能  \* 2）、 @EnableAspectJAutoProxy 会给容器中注册一个组件 AnnotationAwareAspectJAutoProxyCreator  \* 3）、AnnotationAwareAspectJAutoProxyCreator是一个后置处理器；  \* 4）、容器的创建流程：  \* 1）、registerBeanPostProcessors（）注册后置处理器；创建AnnotationAwareAspectJAutoProxyCreator对象  \* 2）、finishBeanFactoryInitialization（）初始化剩下的单实例bean  \* 1）、创建业务逻辑组件和切面组件  \* 2）、AnnotationAwareAspectJAutoProxyCreator拦截组件的创建过程  \* 3）、组件创建完之后，判断组件是否需要增强  \* 是：切面的通知方法，包装成增强器（Advisor）;给业务逻辑组件创建一个代理对象（cglib）；  \* 5）、执行目标方法：  \* 1）、代理对象执行目标方法  \* 2）、CglibAopProxy.intercept()；  \* 1）、得到目标方法的拦截器链（增强器包装成拦截器MethodInterceptor）  \* 2）、利用拦截器的链式机制，依次进入每一个拦截器进行执行；  \* 3）、效果：  \* 正常执行：前置通知-》目标方法-》后置通知-》返回通知  \* 出现异常：前置通知-》目标方法-》后置通知-》异常通知  \*  \*  \*  \*/ |

## 使用

[动态代理] 值在程序运行期间动态的将某段代码切入到指定方法指定位置进行运行的编程方式

导入aop模块,spring aop(spring aspectj )

通知方法:

前置通知(@Before):目标方法运行之前

后置通知(@After):目标方法运行结束之后,无论方法有没有发生异常

返回通知(@AfterReturning):目标方法正常返回之后

异常通知(@AfterThrowing):目标方法出现异常之后运行

环绕通知(@Around):动态代理,手动推进目标方法运行(joinPoint.preced())

切面类和目标方法所在类都需要加入到容器中,并且需要给切面类加入@Aspect注解

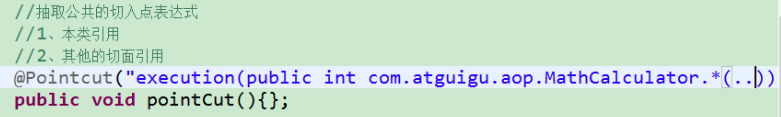
并且需要开启aspect自动代理需要在配置类上加入注解@EnableAspectjAutoProxy

注意:pom.xml中需要引入aspectjweaver,否则@Aspect无法使用

**dependency**>  
 <**groupId**>org.aspectj</**groupId**>  
 <**artifactId**>aspectjweaver</**artifactId**>  
 <**version**>1.9.2</**version**>  
</**dependency**>

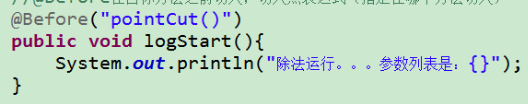
参数使用JoinPoint可以获得一些方法的属性,多个参数的时候必须出现在参数列表的第一位

抽取公共表达式

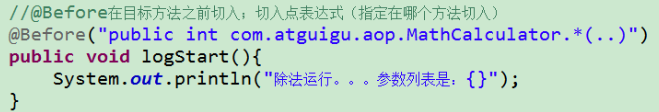


Before

调用公共切面表达式

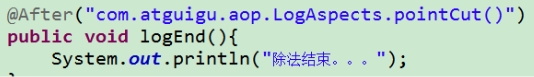


自定义切面表达式



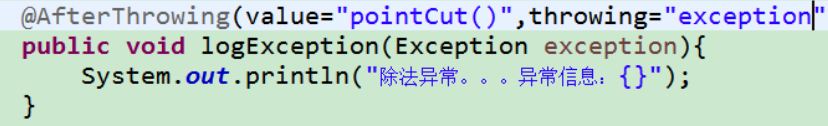
After 无论有没有异常都会调用

如果是其他类的切面表达式需要全类名.方法的形式调用

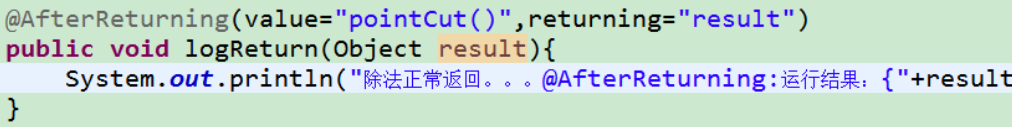


AfterThrowing

需要throwing异常



@AfterReturing 把返回值绑定到result

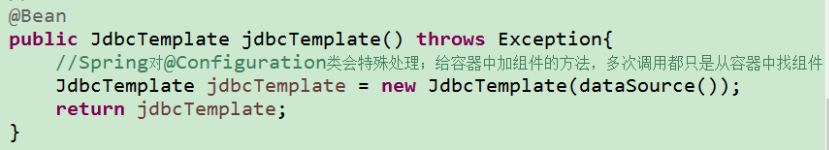


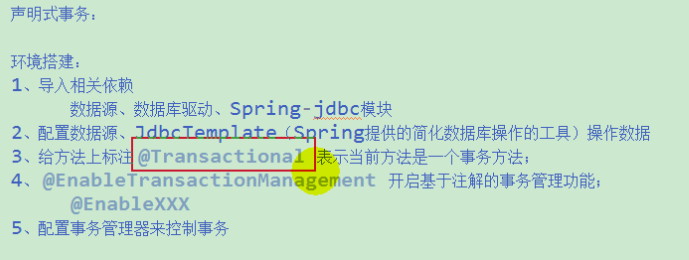
## 注意:

1.如果我们要切入的目标方法是基本类型 返回值会发生类型不匹配的问题 ,所以尽量使用包装类来代替基本类型

# 声明式事物

spring对@Configuration类会特殊处理,给容器中加组件的方法,多次调用会 从容器中寻找组件,而不会再次运行该方法

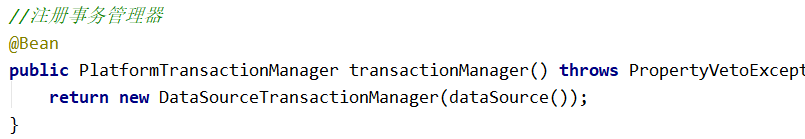




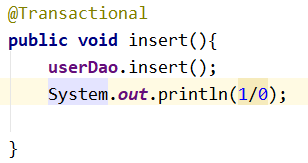
在配置类上开启声明式事务



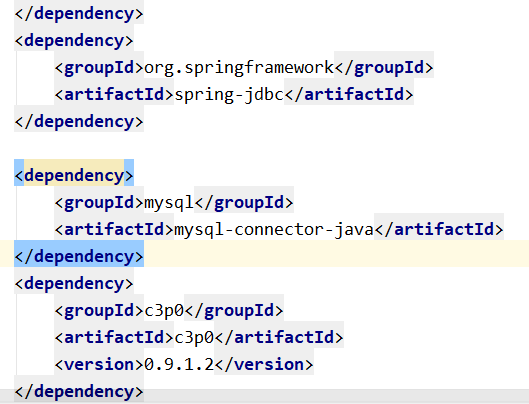
将事务管理类加入到容器中



使用@Transactional标注在方法上出现异常回滚

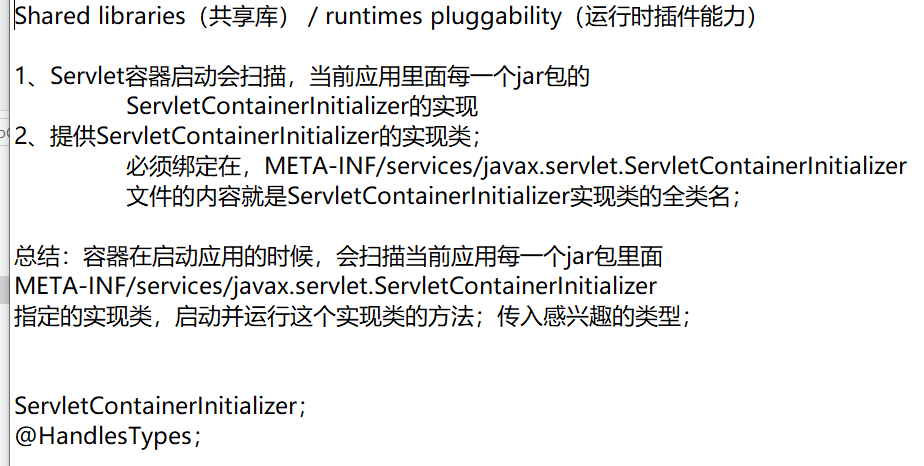


需要导入的依赖:

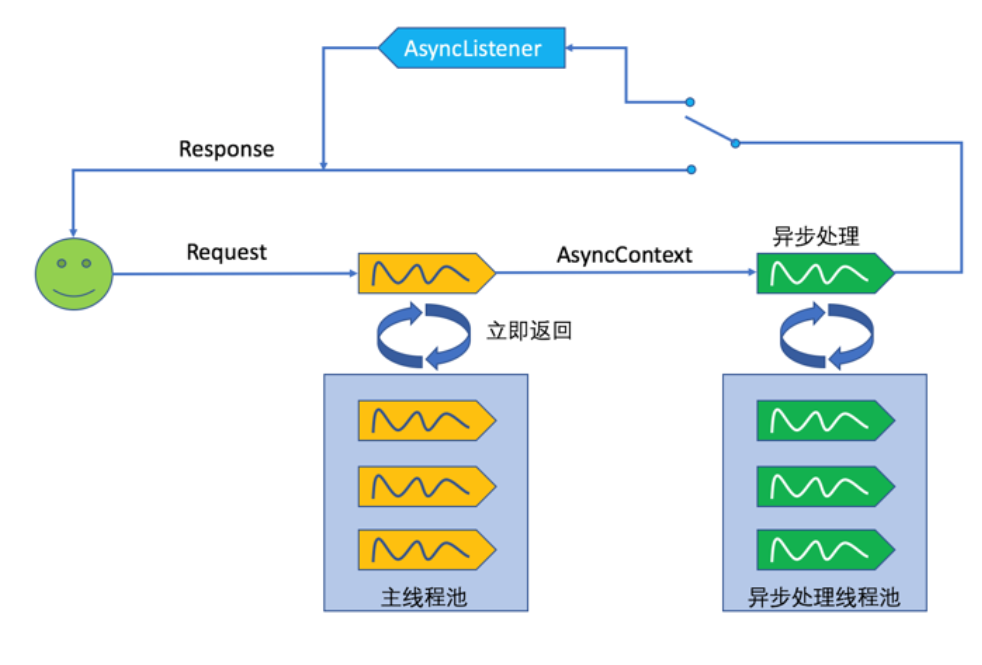


# Servlet3.0





## servlet异步请求处理





# 整合springmvc

## 文字说明

|  |
| --- |
| 1、web容器在启动的时候，会扫描每个jar包下的META-INF/services/javax.servlet.ServletContainerInitializer  2、加载这个文件指定的类SpringServletContainerInitializer  3、spring的应用一启动会加载感兴趣的WebApplicationInitializer接口的下的所有组件；  4、并且为WebApplicationInitializer组件创建对象（组件不是接口，不是抽象类）  1）、AbstractContextLoaderInitializer：创建根容器；createRootApplicationContext()；  2）、AbstractDispatcherServletInitializer：  创建一个web的ioc容器；createServletApplicationContext();  创建了DispatcherServlet；createDispatcherServlet()；  将创建的DispatcherServlet添加到ServletContext中；  getServletMappings();  3）、AbstractAnnotationConfigDispatcherServletInitializer：注解方式配置的DispatcherServlet初始化器  创建根容器：createRootApplicationContext()  getRootConfigClasses();传入一个配置类  创建web的ioc容器： createServletApplicationContext();  获取配置类；getServletConfigClasses();    总结：  以注解方式来启动SpringMVC；继承AbstractAnnotationConfigDispatcherServletInitializer；  实现抽象方法指定DispatcherServlet的配置信息；  ===========================  定制SpringMVC；  1）、@EnableWebMvc:开启SpringMVC定制配置功能；  <mvc:annotation-driven/>；  2）、配置组件（视图解析器、视图映射、静态资源映射、拦截器。。。）  extends WebMvcConfigurerAdapter |

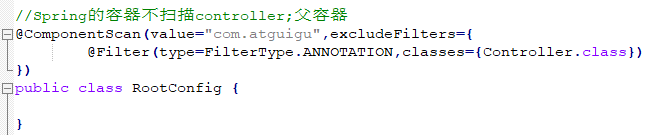
## 配置启动器

如果是注解版需要配置当前这个

如果是xml配置版需要继承另一个抽象类



## 配置spring



## 配置springmvc

继承mvc配置的适配器 根据需要选择重写其中的方法,进行配置具体可以参考spring官方的文档



## springmvc异步请求

### 返回参数写Callable<>()

|  |
| --- |
| @Controller  public class AsyncController {  @ResponseBody  @RequestMapping("/createOrder")  public DeferredResult<Object> createOrder(){  DeferredResult<Object> deferredResult = new DeferredResult<>((long)3000, "create fail...");  DeferredResultQueue.save(deferredResult);  return deferredResult;  }  @ResponseBody  @RequestMapping("/create")  public String create(){  //创建订单  String order = UUID.randomUUID().toString();  DeferredResult<Object> deferredResult = DeferredResultQueue.get();  deferredResult.setResult(order);  return "success===>"+order;  }  /\*\*  \* 1、控制器返回Callable  \* 2、Spring异步处理，将Callable 提交到 TaskExecutor 使用一个隔离的线程进行执行  \* 3、DispatcherServlet和所有的Filter退出web容器的线程，但是response 保持打开状态；  \* 4、Callable返回结果，SpringMVC将请求重新派发给容器，恢复之前的处理；  \* 5、根据Callable返回的结果。SpringMVC继续进行视图渲染流程等（从收请求-视图渲染）。  \*  \* preHandle.../springmvc-annotation/async01  主线程开始...Thread[http-bio-8081-exec-3,5,main]==>1513932494700  主线程结束...Thread[http-bio-8081-exec-3,5,main]==>1513932494700  =========DispatcherServlet及所有的Filter退出线程============================    ================等待Callable执行==========  副线程开始...Thread[MvcAsync1,5,main]==>1513932494707  副线程开始...Thread[MvcAsync1,5,main]==>1513932496708  ================Callable执行完成==========    ================再次收到之前重发过来的请求========  preHandle.../springmvc-annotation/async01  postHandle...（Callable的之前的返回值就是目标方法的返回值）  afterCompletion...    异步的拦截器:  1）、原生API的AsyncListener  2）、SpringMVC：实现AsyncHandlerInterceptor；  \* @return  \*/  @ResponseBody  @RequestMapping("/async01")  public Callable<String> async01(){  System.out.println("主线程开始..."+Thread.currentThread()+"==>"+System.currentTimeMillis());    Callable<String> callable = new Callable<String>() {  @Override  public String call() throws Exception {  System.out.println("副线程开始..."+Thread.currentThread()+"==>"+System.currentTimeMillis());  Thread.sleep(2000);  System.out.println("副线程开始..."+Thread.currentThread()+"==>"+System.currentTimeMillis());  return "Callable<String> async01()";  }  };    System.out.println("主线程结束..."+Thread.currentThread()+"==>"+System.currentTimeMillis());  return callable;  } |

### 返回参数使用DeferredResult<T>

用来整合消息队列等

可以把请求放在DeferredResult<T>中在其他地方进行处理 如中间件等等

当处理完成后在处理完成的地方调用deferredResult.setResult(data);

