## 基础

@component == @Named Spring扫描到后会创建并管理此Bean

@ComponentScan注解启用了组件扫描，默认扫描与配置类相同的包

@ComponentScan注解== <context:component-scan>XML配置

<context:annotation-config> 和 <context:component-scan>的区别：

<context:annotation-config> 是用于激活那些已经在spring容器里注册过的bean上面的注解，默认会注册AutowiredAnnotationBeanPostProcessor、CommonAnnotationBeanPostProcessor、

PersistenceAnnotationBeanPostProcessor，RequiredAnnotationBeanPostProcessor 这四个bean，

这4个Bean都实现了BeanPostProcessor接口，例如AutowiredAnnotationBeanPostProcessor会在spring启动，也就是Bean实例化的时候去查询带有@Autowired注解的属性、set方法或者构造方法，将对象通过反射注入到带有注解的方法或属性中去，

而<context:component-scan>不仅有<context:annotation-config>所具有的功能外，还有将指定包下的类交给spring管理，会扫描@Component,以下他的子注解@Controller,@Respository,@Service

@Autowired(required=false) == @Inject 为null会报空指针

@Configuration 表名此类是一个配置类，包含在spring应用上下文中如何创建bean的细节

@Bean 告诉spring这个方法将返回一个上下文中的bean

@profile(“dev”) 3.2开始可以应用于方法上

@Conditional 4.0后才有（条件化创建bean。用在带有@Bean的方法上）

@Conditional(AA.class) AA implement Condition 实现Condition的matches方法，如果返回true。才会创建此bean

@Primary 可以和@Component 一起用，表示此bean为首选bean（当注入的接口有三个实现类，又没有标示注入id时就会奏效）

@Qualifier(“zfx”) 和@Autowired一起用，表示注入的bean ID为zfx

和@Bean、@Component一起用限定bean ID，默认是将类名首字母小写

@Scope作用于类上表示Bean的作用域 Singleton(默认)、Prototype、Session、Request

@Scope(value=”” ,proxyMode=””);

当单例的Bean中将注入Scope是session或request的bean时候，因为session或request的bean在请求的时候才会有，所以spring会将目标bean的代理注入，等真正调用的时候才会委托给真正的bean。proxyMode的值是ScopedProxyMode.interfaces或者ScopedProxyMode.target\_class 表示基于接口代理或者基于类生成代理（@Scope是类的话只能使用CGLib生成代理）

<aop:scoped-proxy> == @Scope的proxyMode属性 ，默认使用CGLib,可以将 proxy-target-class=”false”

运行时注入外部的值

@Configuration

@PropertySource(“classpath:/aa/bb/app.properties”)

会加载到spring的Enviroment中，所以我们可以

@Autowired

Enviroment env ;

env.getProperty(“xx”)

属性占位符：

XML中，${xx} , java中例如构造参数 @Value(${xx}) 获取properties中的值

使用占位符之前必须配置一个Spring的Bean

PropertySourcesPlaceholderConfigurer（荐，基于Enviroment）或PropertyPlaceholderConfigurer

*PropertySourcesPlaceholderConfigurer* == <context:property-placeholder>

SPel表达式:

字面值：#{1}

引用bean、属性、方法：#{beanId}、#{beanId.name}、#{beanId.say()}

类型：#{T(java.lang.Math.random())}

运算符：#{1\*1}

正则表达式：#{admin.email matches ‘[a-Za-z]’}

## 切面编程

springAOP只支撑方法级别的拦截

spring在运行期把切面织入到spring管理的bean中，直到应用程序需要用到被代理的bean时，spring才会创建代理对象，ApplicationContext从BeanFactory加载所有bean的时候，spring就会创建被代理的对象

SpringAOP使用Aspect切点表达式语言定义切点

例：切点表达式 execution(\* concert.Performance.perform(..))

\*：返回值

concert.Performance：类全路径

perform：方法名

..：参数

@Aspect标注一个POJO类是一个切面。

声明切面中的方法为通知：@After、@AfterReturning、@AfterThrowing、@Around、@Before

切面中声明公共的切点：@PointCut(“execution(\* xxx(..))”)

Public void zfx(){}

在方法上应用切点：

@Before(“zfx()”)

Public void before(){xxxxx}

Aspect需要配置之后才会自动代理：

@Configuration

@EnableAspectJAutoProxy //启动Aspect自动代理

@ComponentScan

Public class Config{

@Bean

Xxxxx 返回切面类的方法

}考虑；‘,l

@EnableAspectAutoProxy == <aop:aspect-autoproxy>

Spring只是使用@Aspect作为创建切面的指导，切面依然是基于代理的。

因为Aspect切面是由AspectJ在运行时创建的，不是由spring创建管理的，但是所有的AspectJ切面都提供了一个静态的aspectOf()方法，该方法返回切面的一个单例，此时这个切面就相当于一个bean交给spring管理，我们也可以将其他bean注入到此切面中，例如：

<bean class=’com.Hello’ factory-method=’aspectOf’ >

<property name=’’ ref=’’></ property>

</bean>

<aop:declare-parents> 通过切面引入新功能

## Spring持久化

## springMVC

ServletContainerInitializer

SpringServletContainerInitializer

WebApplicationInitializer

我们希望DispatcherServlet加载包含Web组件的bean，如控制器、试图解析器以及处理器映射，而ContextLoaderListener要加载应用中其他bean。这些bean通常是驱动应用后端的中间层和数据层组件

springMvc的配置类

@Configuration

@EnableWebMvc

Public class WebConfig{

}

//接收参数名称

Public String spittle(@RequestParam(“spittleId”)long spittleId){

}

//属性占位符

Public String spittle(@PathVariable(“spittleId”)long spittleId , Model model){

}