# 注解

## 声明bean的注解

| **注解** | **描述** |
| --- | --- |
| @Component | 没有明确的对应 |
| @Service | @Component的派生子类，在业务逻辑层(service层)使用 |
| @Repository | @Component的派生子类，在数据访问层(dao层)使用. |
| @Controller | @Component的派生子类，在控制层(controller/web)层使用 |

## 注入bean的注解

| **注解** | **描述** |
| --- | --- |
| @Inject | 1、@Inject是JSR330 (Dependency Injection for Java)中的规范，需要导入javax.inject.Inject;实现注入。 2、@Inject是根据类型进行自动装配的，如果需要按名称进行装配，则需要配合@Named； 3、@Inject可以作用在变量、setter方法、构造函数上 |
| @Named | @Named("XXX") 中的 XX是 Bean 的名称，所以 @Inject和 @Named结合使用时，自动注入的策略就从 byType 转变成 byName 了。 |
| @Autowired | 1、@Autowired是spring自带的注解，通过‘AutowiredAnnotationBeanPostProcessor’ 类实现的依赖注入 2、@Autowired是根据类型进行自动装配的，如果需要按名称进行装配，则需要配合@Qualifier 3、@Autowired有个属性为required，可以配置为false，如果配置为false之后，当没有找到相应bean的时候，系统不会抛错 4、@Autowired可以作用在变量、setter方法、构造函数上 |
| @Qualifier | @Qualifier("XXX") 中的 XX是 Bean 的名称，所以 @Autowired 和 @Qualifier 结合使用时，自动注入的策略就从 byType 转变成 byName 了。不过需要注意的是@Autowired 可以对成员变量、方法以及构造函数进行注释，而 @Qualifier 的标注对象是成员变量、方法入参、构造函数入参。 |
| @Resource | 1、@Resource是JSR250规范的实现，需要导入javax.annotation实现注入。 2、@Resource是根据名称进行自动装配的，一般会指定一个name属性3、@Resource可以作用在变量、setter方法上。 |

## 配置文件的注解

| **注解** | **描述** |
| --- | --- |
| @Configuration | 声明配置类，相等于xml配置文件 |
| @ComponentScan (com.lifeibai) | 自动扫描包及子包的所有 @Component @Service @Repository @Controller 的类,并注册为Bean |
| @WiselyConfiguration | 组合注解 可以替代 @Configuration和@ComponentScan |
| @Bean | 用在方法上，声明为一个bean，不写的值的话，默认bean的名称为方法名称 |

## AOP切面编程注解

| **注解** | **描述** |
| --- | --- |
| @Aspect | 声明这是一个切面 |
| @After @Before. @Around | 定义切面,可以直接将拦截规则(切入点 PointCut)作为参数 |
| @PointCut | 专门定义拦截规则 然后在 @After @Before. @Around 中调用 |
| @Transcational | 事务处理 |
| @Cacheable | 数据缓存 |
| @EnableAaspectJAutoProxy | 开启Spring 对 这个切面(Aspect )的支持 |
| @Target (ElementType.TYPE) | 元注解,用来指定注解修饰类的那个成员 -->指定拦截规则 |
| @Retention(RetentionPolicy.RUNTIME) | 当定义的注解的@Retention为RUNTIME时，才能够通过运行时的反射机制来处理注解.-->指定拦截规则 |

## Spring 常用配置

| **注解** | **描述** |
| --- | --- |
| @import | 导入配置类 |
| @Scope | 新建Bean的实例 |
| @PostConstruct | 在构造函数执行完之后执行 |
| @PreDestroy | 在 Bean 销毁之前执行 |
| @ActiveProfiles | 用来声明活动的 profile |
| @profile | 为不同环境下使用不同的配置提供了支持 @Profile("dev") .......对方法名为 dev-xxxx的方法提供实例化Bean |
| @EnableAsync | 开启异步任务的支持(多线程) |
| @Asyns | 声明这是一个异步任务,可以在类级别 和方法级别声明. |
| @EnableScheduling | 开启对计划任务的支持(定时器) |
| @Scheduled | 声明这是一个计划任务 支持多种计划任务,包含 cron. fixDelay fixRate @Scheduled (dixedDelay = 5000) 通过注解 定时更新 |
| @Conditional | 条件注解,根据满足某一特定条件创建一个特定的Bean |
| @WebAppCofiguration | 指定加载 ApplicationContext是一个WebApplicationContext |
| @ContextConfiguration | 加载配置文件@ContextConfiguration(classes = {TestConfig.class})@ContextConfiguration用来加载ApplicationContext classes属性用来加载配置类 |
| @Value | 放置在字段、方法和方法/构造函数参数上，以指定默认值。  @Value("字符串") -->注入字符串 @Vaule("#{systemProperties['os.name']}")-->注入操作系统属性 @Value ("#{ T (java.lang.Math).random()  \* 100.0 }") --> 注入表达式结果 @Value ("#{demoService.another}") --> 注入其他Bean属性 @Value ( "classpath:com/wisely/highlight\_spring4/ch2/el/test.txt" )     注入文件资源 @Value ("http://www.baidu.com")-->注入网址资源 @Value ("${book.name}" ) --> 注入配置文件  注意: 使用的是$ 而不是 # |

## @Enable\*注解

| **注解** | **描述** |
| --- | --- |
| @EnableAsync | 开启异步任务的支持(多线程) |
| @EnableScheduling | 开启对计划任务的支持(定时器) |
| @EnableWebMVC | 开启对Web MVC 的配置支持 |
| @EnableAaspectJAutoProxy | 开启Spring 对 这个切面(Aspect )的支持 |
| @EnableConfigurationProperties | 开启对@ConfigurationProperties注解配置Bean的支持 |
| @EnableJpaRepositories | 开启对Spring Data JAP Repository 的支持 |
| @EnableTransactionManagement | 开启对注解式事物的支持 |
| @EnableCaching | 开启注解是缓存的支持. |
| @EnableDiscoveryClient | 让服务发现服务器,使用服务器.Spring cloud 实现服务发现 |
| @EnableEurekaServer | 注册服务器 spring cloud 实现服务注册 |
| @EnableScheduling | 让spring可以进行任务调度,功能类似于spring.xml文件中的命名空间 |
| @EnableCaching | 开启Cache缓存支持 |

## springboot注解

| **注解** | **描述** |
| --- | --- |
| @SpringBootConfiguration | 包含了@ComponentScan、@Configuration和@EnableAutoConfiguration注解。 |
| @EnableAutoConfiguration | 自动配置 |
| @ComponentScan | 让spring Boot扫描到Configuration类并把它加入到程序上下文 |
| @PropertySource | 是spring3.1开始引入的基于java config的注解。 通过@PropertySource注解将properties配置文件中的值存储到Spring的 Environment中，Environment接口提供方法去读取配置文件中的值，参数是properties文件中定义的key值。 |
| @ConditionalOnBean | 当容器里面有指定的 Bean 的条件下 |
| @ConditionalOnClass | 当类路径下有指定的类的条件下 |
| @ConditionalOnExpression | 基于SpEL表达式作为判断条件 |
| @ConditionalOnJava | 基于JVM版本作为判断条件 |
| @ConditionalOnJndi | 在JNDI存在的条件下查找指定的位置 |
| @ConditionalOnMissingBean | 当容器里没有指定的 Bean 的情况加 |
| @ConditionalOnMissingClass | 当类路径下没有指定的类的条件下 |
| @ConditionalOnNotWebApplication | 当前项目不是web项目的条件下 |
| @ConditionalOnProperty | 指定的属性是否具有指定的值 |
| @ConditionalOnResourcce | 类路径下是否有指定的值 |

## springmvc注解

| **注解** | **描述** |
| --- | --- |
| @RequestMapping | 请求到处理器功能方法的映射规则；  RequestMapping是一个用来处理请求地址映射的注解（将请求映射到对应的控制器方法中），可用于类或方法上。用于类上，表示类中的所有响应请求的方法都是以该地址作为父路径。  RequestMapping请求路径映射，如果标注在某个controller的类级别上，则表明访问此类路径下的方法都要加上其配置的路径；最常用是标注在方法上，表明哪个具体的方法来接受处理某次请求。  RequestMapping注解有六个属性，下面我们把她分成三类进行说明（下面有相应示例）。 1、 value， method； value：     指定请求的实际地址，指定的地址可以是URI Template 模式（后面将会说明）； method：  指定请求的method类型， GET、POST、PUT、DELETE等； 2、consumes，produces consumes： 指定处理请求的提交内容类型（Content-Type），例如application/json, text/html; produces:    指定返回的内容类型，仅当request请求头中的(Accept)类型中包含该指定类型才返回； 3、params，headers params： 指定request中必须包含某些参数值是，才让该方法处理。 headers： 指定request中必须包含某些指定的header值，才能让该方法处理请求。 |
| @RequestParam | 请求参数到处理器功能处理方法的方法参数上的绑定； |
| @ModelAttribute | 请求参数到命令对象的绑定； |
| @SessionAttributes | 用于声明session级别存储的属性，放置在处理器类上，通常列出模型属性（如@ModelAttribute）对应的名称，则这些属性会透明的保存到session中； |
| @InitBinder | 自定义数据绑定注册支持，用于将请求参数转换到命令对象属性的对应类型；  如果希望某个属性编辑器仅作用于特定的 Controller ，  可以在 Controller 中定义一个标注 @InitBinder 注解的方法，  可以在该方法中向 Controller 了注册若干个属性编辑器 |
| @CookieValue | cookie数据到处理器功能处理方法的方法参数上的绑定； |
| @RequestHeader | 请求头（header）数据到处理器功能处理方法的方法参数上的绑定； |
| @RequestBody | 请求的body体的绑定（通过HttpMessageConverter进行类型转换）； |
| @ResponseBody | 处理器功能处理方法的返回值作为响应体（通过HttpMessageConverter进行类型转换）；  通过HttpMessageConverter接口转换为指定格式的数据 |
| @ResponseStatus | 定义处理器功能处理方法/异常处理器返回的状态码和原因； |
| @ExceptionHandler | 注解式声明异常处理器； |
| @PathVariable | 请求URI中的模板变量部分到处理器功能处理方法的方法参数上的绑定 |
| @RestContrller | 相当于@ResponseBody和@Controller的结合 : |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# 动态代理

## 基于接口的动态代理

提供者：JDK官方的Proxy类

要求：被代理类至少实现一个接口

## 基于子类的动态代理

提供者：CGLIB

要求：被代理类不能用final修饰

# AOP

## Aop相关术语

### Jionpoint（连接点）

指那些被拦截的点。在spring中，这些点指的是方法

### Pointcut（切入点）

指要对哪些Joinpoint进行拦截的定义

### Advice（通知/增强）

指拦截到Joinpoint之后要做的事情

通知的类型：前置通知，后置通知，异常通知，最终通知，环绕通知

### Introduction（引介）

引介是一种特殊的通知在不修改类代码的前提下，Introduction可以在运行时为类动态的添加一些方法或Field.

### Target（目标对象）

代理的目标对象

### Weaving（织入）

指把增强应用到目前对象来创建新的代理对象的过程。

Spring采用动态代理织入，而AspectJ采用编译期织入和类装载期织入。

### Proxy（代理）

一个类被AOP织入增强后，就产生一个结果代理类

### Aspect（切面）

是切入点和通知（引介）的结合

## 常用标签

### <aop:config>

作用：用于声明开始aop配置

### <aop:aspect>

作用：用于配置切面

属性：id给切面提供一个唯一标识；ref引用配置好的通知类bean的id。

### <aop:pointcut>

作用：用于配置切点表达式

属性：expression用于定义切点表达式；id用于给切点表达式提供唯一标识。

### <aop:before>

作用：用于配置前置通知

属性：method指定通知方法的名称；pointct定义切入点表达式；pointcut-ref指定切入点表达式的引用。

### <aop:after-returning>

作用：用于配置后置通知

属性：method指定通知方法的名称；pointct定义切入点表达式；pointcut-ref指定切入点表达式的引用

### <aop:after>

作用：用于配置最终通知

属性：method指定通知方法的名称；pointct定义切入点表达式；pointcut-ref指定切入点表达式的引用

### <aop:around>

作用：用于配置环绕通知

属性：method指定通知方法的名称；pointct定义切入点表达式；pointcut-ref指定切入点表达式的引用

## 常用注解

@Aspect

作用：把当前类声明为切面类

@Before

作用：把当前方法看成是前置通知

属性：value用于指定切入点表达式，还可以指定切入点表达式的引用

@AfterReturning

作用：把当前方法看成后置通知

属性：value用于指定切入点表达式，还可以指定切入点表达式的引用

@AfterThrowing

作用：把当前方法看成异常通知

属性：value用于指定切入点表达式，还可以指定切入点表达式的引用

@After

作用：把当前方法看成最终通知

属性：value用于指定切入点表达式，还可以指定切入点表达式的引用

@Around

作用：把当前方法看成环绕通知

属性：value用于指定切入点表达式，还可以指定切入点表达式的引用

@Pointcut

作用：指定切入点表达式

属性：value指定表达式内容

@EnableAspectJAutoProxy

作用：开启spirng对注解AOP的支持

## 切入点表达式

语法：execution([修饰符] 返回值类型 包名.类名.方法名(参数))

### 全匹配方式

public void com.house.Class.method()

### 访问修饰符可以省略

void com.house.Class.method()

### 返回值可以用\*，表示任意返回值

\* com.house.Class.method()

### 包名可以使用\*，表示任意包，一个层级用一个\*

\*﷒\*.\*.Class.method()

### 使用..表示当前包及其子包

\* com..Class.method()

### 类名可以使用\*，表示任意类

\* com..\*.method()

### 方法名可以用\*，表示任意方法

\* com..\*.\*()

### 参数列表可以使用\*，表示参数可以是任意数据类型，但是必须有参数

\* com..\*.\*(\*)

### 参数列表可以用..，表示有无参数均可，有参数可以是任意类型

\* com..\*.\*(..)

### 全通配模式

\* \*..\*.\*(..)

## 通知类型

### 前置通知

切入点方法执行之前执行

### 后置通知

切入点方法正常执行之后执行。它和异常通知只能有一个执行

### 异常通知

切入点方法执行产生异常后执行。它和后置通知只能有一个执行

### 最终通知

无论切入点方法执行时是否产生异常，它都会在其后面执行

### 环绕通知

Sping框架提供的一种可以在代码中手工控制增强部分什么时候执行的方式

Spring提供一个接口：ProceedingJoinPoint，它可以作为环绕通知的参数，在环绕通知执行时，spring框架会提供接口的实现类对象

该接口有一个方法proceed()，此方法相当于method.invoke()