

**Spring注解**

# Java注解

## 定义

注解通过 @interface 关键字进行定义。

public @interface MyAnnotation {}

## 元注解

元注解是可以注解到注解上的注解，或者说元注解是一种基本注解，但是它能够应用到其它的注解上面。

如果难于理解的话，你可以这样理解。元注解也是一张标签，但是它是一张特殊的标签，它的作用和目的就是给其他普通的标签进行解释说明的。

元标签有 @Retention、@Documented、@Target、@Inherited、@Repeatable 5 种。

### Retention

Retention 的英文意为保留期的意思。当 @Retention 应用到一个注解上的时候，它解释说明了这个注解的的存活时间。

它的取值如下：   
- RetentionPolicy.SOURCE 注解只在源码阶段保留，在编译器进行编译时它将被丢弃忽视。   
- RetentionPolicy.CLASS 注解只被保留到编译进行的时候，它并不会被加载到 JVM 中。   
- RetentionPolicy.RUNTIME 注解可以保留到程序运行的时候，它会被加载进入到 JVM 中，所以在程序运行时可以获取到它们。

### Documented

顾名思义，这个元注解肯定是和文档有关。它的作用是能够将注解中的元素包含到 Javadoc 中去

### Target

Target 是目标的意思，@Target 指定了注解运用的地方。

你可以这样理解，当一个注解被 @Target 注解时，这个注解就被限定了运用的场景。

类比到标签，原本标签是你想张贴到哪个地方就到哪个地方，但是因为 @Target 的存在，它张贴的地方就非常具体了，比如只能张贴到方法上、类上、方法参数上等等。@Target 有下面的取值

ElementType.ANNOTATION\_TYPE 可以给一个注解进行注解

ElementType.CONSTRUCTOR 可以给构造方法进行注解

ElementType.FIELD 可以给属性进行注解

ElementType.LOCAL\_VARIABLE 可以给局部变量进行注解

ElementType.METHOD 可以给方法进行注解

ElementType.PACKAGE 可以给一个包进行注解

ElementType.PARAMETER 可以给一个方法内的参数进行注解

ElementType.TYPE 可以给一个类型进行注解，比如类、接口、枚举

### Inherited

Inherited 是继承的意思，但是它并不是说注解本身可以继承，而是说如果一个超类被 @Inherited 注解过的注解进行注解的话，那么如果它的子类没有被任何注解应用的话，那么这个子类就继承了超类的注解。

## 注解的属性

注解的属性也叫做成员变量。注解只有成员变量，没有方法。注解的成员变量在注解的定义中以“无形参的方法”形式来声明，其方法名定义了该成员变量的名字，其返回值定义了该成员变量的类型

## Java内置注解

主要说一下这个FunctionalInterface

@FunctionalInterface  
**private interface** DocumentFilter {  
  
 **boolean** match(Document document);  
  
}

它注解了一个单抽象函数接口，在使用lambda表达式的时候，可以正确的抽象一个实例出来。

比如：

**private** DocumentFilter test(String s){  
 System.***out***.println(s);  
 **return** (Document document) ->{  
 System.***out***.println(s);  
 **return true**;  
 };  
}

该函数返回了一个lambda表达式，类似于javascript中的lambda一样，该表达式会在虚拟机中填充到match函数中，使用动态代理，生成一个DocumentFilter实例化对象。调用match函数的时候，再正确的调用到表达式里去。如果有非抽象函数，也可以使用该对象调用，详细的表达式知识，后续可能会有额外的专题描述。

# Spring注解

有些不知道怎么翻译合适的，原文也放上了。

## 基础组件

### Component

注解一个class是一个组件，这个组件会被自动发现（扫描到），其他的类级注解可以看成是一个特殊的Component。

### Service

类似于Component，这个组件更多的用于标识当前class是接口定义型组件。

### Controller

也是一个特殊的Component，一般都和RequestMapping一起用。

### Repository

标识一个DAO，也是一个特殊的…

## Web组件

### Mapping

暗示该注解是一个web mapping注解，用作注解的注解

### RequestMapping

*Annotation for mapping web requests onto methods in request-handling classes with flexible method signatures*

映射web请求到函数中。详细映射过程参照*RequestMappingHandlerAdapter*，后面Spring MVC章节也会详细说明。

#### name属性

本来没什么用，就是一个标识

结果后面4以后，spring在解析请求的时候加入了个新规则。可以根据类+函数的方式来调用。

你可以自定义name属性，默认制是通过类的大写字符+#+方法名构建的，比如CustomerController的getCustomer方法，name值默认为CC#getCustomer。然后在构建请求的时候，使用CC#getCustomer（没有测试过）。详细知识可以参照（*MvcUriComponentsBuilder，HandlerMethodMappingNamingStrategy*）

#### value、path属性

互为别名，不解释了，就是配置映射的，例如（/getUser）

当然，也可以使用*/${connect}，*在参数里，使用PathVariable获取connect

#### method属性

http请求*GET, POST, HEAD, OPTIONS, PUT, PATCH, DELETE, TRACE*

params

#### headers属性

就是headers

#### consumes属性

接受的contentType

#### produces属性

返回的contentType

### PostMapping

### GetMapping

### PatchMapping

### PutMapping

### DeleteMapping

以上同RequestMapping，使用别名方式。

### RequestHeader

### RequestMethod

以上不解释

### RequestAttribute

请求函数参数属性，校验性质

### RequestBody

请求使用body传递参数，并且由*HttpMessageConverter*转换参数到函数指定类型，若转换失败会提示异常。

### RequestParam

类似RequestBody，不过可以在servlet中使用。将参数转换成Map（String,String）形式,当时用的时候，会从那个map中获取数据，如@RequestParam(name=”userId”) String userId。

### RequestPart

处理*multipart/form-data*请求（RequestParam也可以），根据*Content-Type*属性使用*HttpMessageConverter*整合参数（这点和RequestBody类似，而RequestParam使用Converter或者*PropertyEditor*获取参数）。一般情况下，RequestParam适合name-value形式的数据，RequestPart或者RequestBody适合JSON、XML形式的数据

public JsonResult uploadFile(@RequestPart("file") MultipartFile file, @RequestParam String bucket){

public JsonResult uploadStringFile(@RequestParam("stringFile") String stringFile, @RequestParam("bucket") String bucket){

RequestPart的类型不限定，param只会转移成String

### ResponseStatus

可以转换返回码，意义不是特别大，详细参照*ResponseStatusExceptionResolver*

### RestController

新版本，和Controller区别不大了。

### ExceptionHandler

http请求异常后的处理类

### CrossOrigin

处理跨域的注解，一般我们使用全局的。设定之后，这个类或者函数的请求就可以跨域了

@Bean

public WebMvcConfigurer corsConfigurer() {

return new WebMvcConfigurerAdapter() {

@Override

public void addCorsMappings(CorsRegistry registry) {

registry.addMapping("/\*\*")

.allowedOrigins(alloweredOrigins)

.allowCredentials(true)

.maxAge(3600)

.allowedHeaders("Origin", "X-Requested-With", "Content-Type", "Accept");

}

};

}

### CookieValue

设定cookie的东西

### InitBinder

数据类型转换，意义不是特别大

public class BaseController {

@InitBinder

protected void initBinder(WebDataBinder binder) {

binder.registerCustomEditor(Date.class, new MyDateEditor());

binder.registerCustomEditor(Double.class, new DoubleEditor());

binder.registerCustomEditor(Integer.class, new IntegerEditor());

}

private class MyDateEditor extends PropertyEditorSupport {

@Override

public void setAsText(String text) throws IllegalArgumentException {

SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

Date date = null;

try {

date = format.parse(text);

} catch (ParseException e) {

format = new SimpleDateFormat("yyyy-MM-dd");

try {

date = format.parse(text);

} catch (ParseException e1) {

}

}

setValue(date);

}

}

public class DoubleEditor extends PropertiesEditor {

@Override

public void setAsText(String text) throws IllegalArgumentException {

if (text == null || text.equals("")) {

text = "0";

}

setValue(Double.parseDouble(text));

}

@Override

public String getAsText() {

return getValue().toString();

}

}

public class IntegerEditor extends PropertiesEditor {

@Override

public void setAsText(String text) throws IllegalArgumentException {

if (text == null || text.equals("")) {

text = "0";

}

setValue(Integer.parseInt(text));

}

@Override

public String getAsText() {

return getValue().toString();

}

}

}

### ControllerAdvice*、*RestControllerAdvice

可以全局处理，比如添加上面的类型转换、或者添加属性，addAttribute，异常处理。例如：

@ControllerAdvice  
**public class** WoBSSWebExceptionHandler {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(WoBSSWebExceptionHandler.**class**);  
  
 @ExceptionHandler(value = UserException.**class**)  
 @ResponseBody  
 **public** Object baseErrorHandler(HttpServletRequest req, Exception e) **throws** Exception {  
 ***logger***.error(e.getMessage(), e);  
 UserException userException = (UserException)e;  
 JSONObject result = **new** JSONObject();  
 result.put(**"code"**, userException.getErrorCode());  
 result.put(**"msg"**, userException.getErrorMessage());  
 **return** result.toJSONString();  
 }  
  
 @ExceptionHandler(value = Exception.**class**)  
 @ResponseBody  
 **public** Object defaultErrorHandler(HttpServletRequest req, Exception e) **throws** Exception {  
 ***logger***.error(e.getMessage(), e);  
 JSONObject result = **new** JSONObject();  
 result.put(**"code"**, **"8888"**);  
 result.put(**"msg"**, **"系统异常："** + e.getMessage());  
 **return** result.toJSONString();  
 }  
}

### SessionAttribute、SessionAttributes

Session属性，用处不是特别大。

## Context组件

### ComponentScan

扫描注解*Configuration，*功能和xml中的*<context:component-scan>*类似

和扫描那个函数类似，提供了includeFilters，excludeFilters，useDefaultFilters等等这些配置。

注意：ScopedProxyMode，这个属性，设置bean实例方式（代理类型，有，或者没有，使用jdk还是cglib），如果这里设置了，会覆盖*ClassPathBeanDefinitionScanner*中的设置。

注解ComponentScans，配合用，使ComponentScan可重复

### Bean

不解释

### Conditional

放在Class或者函数之前，可以判定该函数或者Class是否启用，接收一个参数，就是一个Class，这个Class实现了org.springframework.context.annotation.Condition接口，实现matches函数

### Configuration

用来声明Bean，如：

#### 声明Bean

@Configuration

public class AppConfig {

@Bean

public MyBean myBean() {

// instantiate, configure and return bean ...

}

}

过去的话，我们使用这种模式

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();

ctx.register(AppConfig.class);

ctx.refresh();

MyBean myBean = ctx.getBean(MyBean.class);

// use myBean ...

#### 唯一参数构造函数

也可以声明自己的构造函数的参数，里面使用@Bean声明

@Configuration

public class AppConfig {

private final SomeBean someBean;

public AppConfig(SomeBean someBean) {

this.someBean = someBean;

}

// @Bean definition using "SomeBean"

}

#### 使用自定义扫描器

@Configuration

@ComponentScan("com.acme.app.services")

public class AppConfig {

// various @Bean definitions ...

}

#### 使用环境变量

@Configuration

public class AppConfig {

@Autowired Environment env;

@Bean

public MyBean myBean() {

MyBean myBean = new MyBean();

myBean.setName(env.getProperty("bean.name"));

return myBean;

}

}

#### 使用配置文件

@Configuration

@PropertySource("classpath:/com/acme/app.properties")

public class AppConfig {

@Inject Environment env;

@Bean

public MyBean myBean() {

return new MyBean(env.getProperty("bean.name"));

}

}

#### 使用注解获取配置文件参数

@Configuration

@PropertySource("classpath:/com/acme/app.properties")

public class AppConfig {

@Value("${bean.name}") String beanName;

@Bean

public MyBean myBean() {

return new MyBean(beanName);

}

}

#### 嵌套使用

@Configuration

public class DatabaseConfig {

@Bean

public DataSource dataSource() {

// instantiate, configure and return DataSource

}

}

@Configuration

@Import(DatabaseConfig.class)

public class AppConfig {

private final DatabaseConfig dataConfig;

public AppConfig(DatabaseConfig dataConfig) {

this.dataConfig = dataConfig;

}

@Bean

public MyBean myBean() {

// reference the dataSource() bean method

return new MyBean(dataConfig.dataSource());

}

}

#### 根据开发（development），测试（test），生产（production）区分配置

@Profile("development")

@Configuration

public class EmbeddedDatabaseConfig {

@Bean

public DataSource dataSource() {

// instantiate, configure and return embedded DataSource

}

}

@Profile("production")

@Configuration

public class ProductionDatabaseConfig {

@Bean

public DataSource dataSource() {

// instantiate, configure and return production DataSource

}

}

Alternatively, you may also declare profile conditions at the @Bean method level, e.g. for alternative bean variants within the same configuration class:

@Configuration

public class ProfileDatabaseConfig {

@Bean("dataSource")

@Profile("development")

public DataSource embeddedDatabase() { ... }

@Bean("dataSource")

@Profile("production")

public DataSource productionDatabase() { ... }

}

#### 使用XML配置文件

@Configuration

@ImportResource("classpath:/com/acme/database-config.xml")

public class AppConfig {

@Inject DataSource dataSource; // from XML

@Bean

public MyBean myBean() {

// inject the XML-defined dataSource bean

return new MyBean(this.dataSource);

}

}

#### 内部配置

@Configuration

public class AppConfig {

@Inject DataSource dataSource;

@Bean

public MyBean myBean() {

return new MyBean(dataSource);

}

@Configuration

static class DatabaseConfig {

@Bean

DataSource dataSource() {

return new EmbeddedDatabaseBuilder().build();

}

}

}

### DependsOn

@DependsOn(value = {**"cde"**,**"cde"**})

没什么可说的，就是判断，在谁之后才实例化，如果没有，直接抛出

Consider defining a bean named '123' in your configuration

### Description

没毛用

### Import

引用未被扫描的@Configuration，或者[ImportSelector](file:///D:\\maven\\repository\\org\\springframework\\spring-context\\5.0.8.RELEASE\\spring-context-5.0.8.RELEASE-javadoc\\org\\springframework\\context\\annotation\\ImportSelector.html" \o "interface in org.springframework.context.annotation) and [ImportBeanDefinitionRegistrar](file:///D:\maven\repository\org\springframework\spring-context\5.0.8.RELEASE\spring-context-5.0.8.RELEASE-javadoc\org\springframework\context\annotation\ImportBeanDefinitionRegistrar.html) implementations实现类。

如果要引用XML配置，或者不是在Configuration中配置的Bean，请使用**ImportResource**

### ImportResource

引用XML配置，可以配置自己实现的BeanDefinitionReader。

### ImportAware

*Set the annotation metadata of the importing @{****@code*** *Configuration} class*

有时我们可能配置里想使用注解中的属性值，如下：

查看@EnableRedissonHttpSession源码

@Retention(RetentionPolicy.RUNTIME)

@Target({ElementType.TYPE})

@Import({RedissonHttpSessionConfiguration.class})

@Configuration

public @interface EnableRedissonHttpSession {

int maxInactiveIntervalInSeconds() default 1800;

String keyPrefix() default "";

}

@Configuration

public class RedissonHttpSessionConfiguration extends SpringHttpSessionConfiguration implements ImportAware {

private Integer maxInactiveIntervalInSeconds;

private String keyPrefix;

public void setImportMetadata(AnnotationMetadata importMetadata){

Map<String, Object> map = importMetadata.getAnnotationAttributes(EnableRedissonHttpSession.class.getName());

AnnotationAttributes attrs = AnnotationAttributes.fromMap(map);

this.keyPrefix = attrs.getString("keyPrefix");

this.maxInactiveIntervalInSeconds = (Integer)attrs.getNumber("maxInactiveIntervalInSeconds");

}

}

这时，值就进去了。

### Lazy

延迟加载，不细说。

### Profile

使用哪种配置文件，略。

### PropertySource

Configuration引入配置文件

@Configuration

@PropertySource("classpath:/com/myco/app.properties")

public class AppConfig {

@Autowired

Environment env;

@Bean

public TestBean testBean() {

TestBean testBean = new TestBean();

testBean.setName(env.getProperty("testbean.name"));

return testBean;

}

}

使用配置了值的配置路径

@Configuration

@PropertySource(**"classpath:/${my.placeholder.default}/a.properties"**)

public class AppConfig {

@Autowired

Environment env;

@Bean

public TestBean testBean() {

TestBean testBean = new TestBean();

testBean.setName(env.getProperty("testbean.name"));

return testBean;

}

}

当我们使用了多个配置文件的时候，某一个key唯一的取值，取决于哪个Configuration先注入

@Configuration

@PropertySource("classpath:/com/myco/a.properties")

public class ConfigA { }

@Configuration

@PropertySource("classpath:/com/myco/b.properties")

public class ConfigB { }

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();

ctx.register(ConfigA.class);

ctx.register(ConfigB.class);

ctx.refresh();

当我们很难判定谁先进入系统的时候，最好不要这么配置。

### Role

没啥用，不解释。

### Scope

使用哪一种策略，单例，请求，还是非单例。

注意，不管是请求、还是非单例（prototype），都不是做多个实例，还是一个实例，使用代理模式实现，所以这个注解还有个参数，动态代理模式（cglib，jdk）

### PropertySources

配合多个PropertySource

### EnableMBeanExport

支撑JMX的，没遇到过。

### EnableLoadTimeWeaving

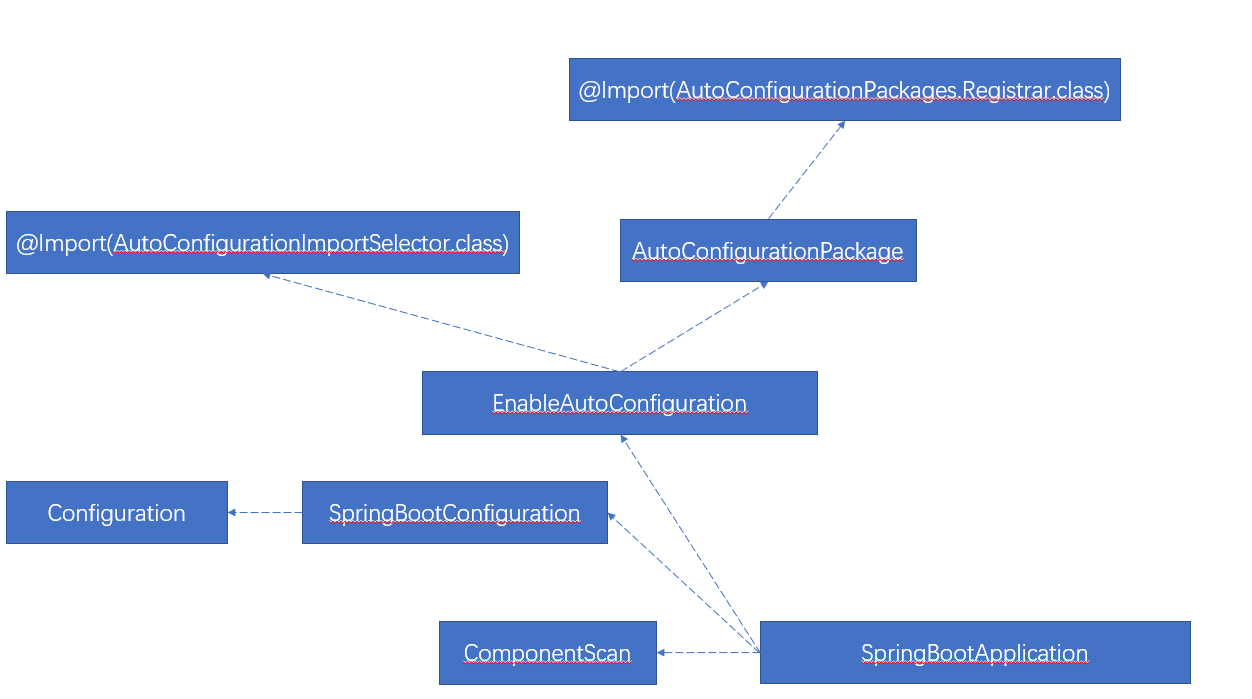
没遇到过，不知道干嘛用的

### EnableAspectJAutoProxy

开启apectj代理，不需要这个注解，自动开启。除非需要设置代理模式等，否则不需要主动开启。

## AutoConfiguration组件

### SpringBootApplication



1. @ComponentScan(excludeFilters = {

@Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),

@Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })

1. SpringBootConfiguration

标志当前应用为SpringBootApplication

1. Configuration
2. AutoConfigurationPackage

AutoConfigurationPackages.Registrar 的作用是添加对@AutoConfigurationPackage注解作用的类所在包名的扫描，并将该包名下扫描到的需要注册的bean注册

### EnableConfigurationProperties

Import（EnableConfigurationPropertiesImportSelector）

value 为需要绑定参数的class。必须与Configuration配合使用。

当系统启动执行到BeanDefinitionRegistryPostProcessor的时候，获取该ConfigurationClass的importSelector的时候执行EnableConfigurationPropertiesImportSelector，将注解了EnableConfigurationProperties属性的Class加入到BeanDefinition中。

当系统调用BeanPostProcessor中的ConfigurationPropertiesBindingPostProcessor的时候，如果该bean使用注解ConfigurationProperties，将会自动根据系统参数，绑定该bean中的属性，不需要@Value等，直接就反射绑定了。

# 注解读取

## AnnotationConfigUtils

### attributesFor

获取注解属性，参数为（AnnotatedTypeMetadata metadata, String annotationClassName）

所有注解几乎都是使用（StandardAnnotationMetadata）初始化的注解源，StandardAnnotationMetadata实现了AnnotatedTypeMetadata。

StandardAnnotationMetadata🡪getAnnotationAttributes

里面调用AnnotatedElementUtils.*getMergedAnnotationAttributes*。

### attributesForRepeatable

获取可重复的属性，比如PropertySource，参数和上面类似，多了个复数的

参数为PropertySources.class,target--MetaData

1. *addAttributesIfNotNull*
2. metadata.getAnnotationAttributes

最终还是调用到AnnotatedElementUtils.*getMergedAnnotationAttributes*

## AnnotatedElementUtils

### getMergedAnnotationAttributes

参数为AnnotatedElement element,  
 String annotationName, **boolean** classValuesAsString, **boolean** nestedAnnotationsAsMap

其中AnnotatedElement所有Class，Annotation等都实现了这个接口

1. searchWithGetSemantics

获取当前对象（第一次调用该函数的时候，对象就是类本身，之后可能是该类所含注解等）

1. searchWithGetSemanticsInAnnotations

不处理Java注解

获取方式可以用名称、类型、或者processor强制。

当匹配成功后，调用processor的process函数

最终调用AnnotationUtils.retrieveAnnotationAttributes函数获取属性

1. postProcessAnnotationAttributes

调用AnnotationUtils. *postProcessAnnotationAttributes*

这个函数在获取完数据后，最后一次修改处理。

## AnnotationUtils

1. retrieveAnnotationAttributes

获取注解参数值，很简单，获取值，没有就用default

value是个基本单位的时候，不处理

注解的话，判定是否将内部注解的值转换成Map

1. *postProcessAnnotationAttributes*

默认没有做太多事情，主要是判断默认值什么的。

## PropertiesLoaderUtils