spring 注解

作者：Weixl

原文链接：https://ld246.com/article/1602154347698

来源网站：链滴

spring注解

- 之前学的spring注解常用无非就是那么几个 @SpringBootApplication，@ServletComponentSca，@Autowired，@Resource，@Mapper....等等等。
- 但是~~！但是！。进入工作之后，突然发现项目中的架构多了很多的 注解，以前学的时候都没有过，但是感觉很牛逼啊！
- 列如：@Import @EnableCaching， 一搜就是什么 缓存啊，导入啊什么的。裂开。
- 做个统计吧： 没有统计 spring-data-jpa注解

注解的功能

1. 采用纯java代码，不在需要配置繁杂的xml文件
2. 在配置中也可享受面向对象带来的好处
3. 类型安全对重构可以提供良好的支持
4. 减少复杂配置文件的同时亦能享受到springIoC容器提供的功能

spring注解

常用：

- @Component

@Component
是所有受Spring 管理组件的通用形式，@Component注解可以放在类的头上，@Component不推使用。

- @Controller - 表示层

@Controller对应表现层的Bean，也就是Action，Controller层

```
@Controller
public class UserAction extends BaseAction<User>{
    ......
}
```

使用@Controller注解标识一个 表现层Action后，就将 Action交给spring容器管理，在Spring容器会存在一个名为 'UserAction' 的action。
容器相当于一个 key-value的map集合。 key是唯一的。 key根据注册的类名来获取的（首字母小写。
@Controller(value='XXX') value可以指定要注册的key的名称
通常方法需要配合注解@RequestMapping

- @Scope - 切换单例还是多例

@Scope("prototype") 声明注册Bean的范围。
prototype： 多例，保证每个请求都有一个独特的 Bean对象来处理请求。避免线程安全问题
singleton： 单例(默认)。每次访问都是同一个Bean对象操作
在struts2 中是要求每次访问都必须是不同Bean对象 - prototype

- @Service - 业务层

@Service创建Bean的实例对象，注册到spring容器中
@Service("XXX") 可以指定注册到spring的名称

- @Repository - 数据层

@Repository是数据层的实例注解。
但是实际项目中一般都没有用到，项目融合Mybatis进行数据层操作，一般是写入@mapper注解

- @SpringBootApplication - 启动配置

申明让spring boot自动给程序进行必要的配置（有三个配置）：
@Configuration： 相当于传统的xml配置文件
@EnableAutoConfiguration： 自动配置,就是你添加了jar包，会自动配置相关接口到你的spring应中
@ComponentScan： 扫描目录下面的所有 @component注册Bean

- ResponseBody - 返回结果

方法的返回结果直接写入到HTTP response body中。用于构建 RESTFul 的API
一般配合@RequestMappping一起用，会直接返回JSON数据。
其实就是序列化pojo实体类数据，为json数据

- @RequestBody - 获取参数

RESTful风格的 获取请求参数 JSON格式。 可以自动封装成一个Bean对象

- RestController - 控制层注解

用于标注控制层组件(如struts中的action)，@ResponseBody和@Controller的合集。

- @RequestMapping - url映射

提供路由信息，负责URL到Controller中具体函数的映射

- @EnableAutoConfiguration

SpringBoot自动配置（auto-configuration）：尝试根据你添加的jar依赖自动配置你的Spring应用。

例如，如果你的classpath下存在HSQLDB，并且你没有手动配置任何数据库连接beans，那么我们将动配置一个内存型（in-memory）数据库”。

你可以将@EnableAutoConfiguration或者@SpringBootApplication注解添加到一个@Configuration类上来选择自动配置。

如果发现应用了你不想要的特定自动配置类，你可以使用@EnableAutoConfiguration注解的排除属来禁用它们。

- @ComponentScan

表示自动扫描搜索组件，如果有扫描到@controller @Service ...等就注册为Bean。
扫描到@Configuration就设置为配置类调用。

- @Configuration - 配置类

相当于传统的XML配置文件。
如果有些第三方库需要用到xml文件，建议仍然通过@Configuration类作为项目的配置主类——可使用@ImportResource注解加载xml配置文件

属性：
proBeanMethods指定@Bean注解标注的方法是否使用代理
proxyBeanMethods（默认true）： Full模式，也就是单例模式，通过方法调用指向的依旧是原来旧Bean
proxyBeanMethods=false： lite模式，直接返回新的实例对象，提高Spring的启动速度（官方推荐

- @Autowired - 注入

DI依赖注入，自动导入依赖的bean
属性：required 默认 true
true： 注入的时候 对应的Bean必须存在，不然就报错
false： 注入的时候 对应Bean存在就注入，不存在就跳过

- @Bean

用@Bean标注方法等价于XML中配置的bean。
注册类

- @Resource(name="name",type="type")

没有括号内内容的话，默认Bean的类名称。与@Autowired干类似的事。

- @PathVariable：获取参数。

获取RequestMapping('/{a}') 请求路径中 a的数据

- @Primary - 设置主Bean

当我们在Config中配置了多个同父类下的 实现Bean时，可以使用@Primary来指定默认的Bean

- @Nullable - 为空

标注在方法，参数上，表示对应的值可以为空

- @NonNull - 不为空

标注在方法、字段、参数之上，表示对应的值不可以为空

- @Deprecated - 过期方法

标注在类 或 方法上，表示当前类 或 方法 已经过期 不建议使用，会在名称中间有一道横线

不常用

- @Import - 导入Bean

通过@Import可以将没有注册到 IOC 容器的Bean给注册到 spring中
方法1： 直接导入具体的Bean类，可以写成一个{}数组，写多个
@Import({Dog.class, Cat.class})

方法2： 写一个配置文件
public class MyConfig {

```
    @Bean
    public Dog getDog(){
```

```
        return new Dog();
    }

    @Bean
    public Cat getCat(){
        return new Cat();
    }

}

@Import(MyConfig.class) // 导入对应配置文件的class

使用Import导入之后，就可以使用Bean
ConfigurableApplicationContext context = SpringApplication.run(App.class, args);
        System.out.println(context.getBean(Dog.class));
        System.out.println(context.getBean(Cat.class));
        context.close();
```

- @Value

注入Spring boot application.properties 配置文件的 值
@Value(“${spring.redis.weight}”)
private int weight;

- @ ImportResource

用来加载xml配置文件

- @Inject

等价于默认的@Autowired，只是没有required属性

- @Qualifier - 与@Autowired配合

一般与@Autowired配合使用，当有多个同一类型的Bean时，可以用@Qualifier(“name”)来指定。
与@Autowired结合起来相当于是 @Resource(name="") 一个作用
可用在区分 父子继承关系中

```
@Service
public class AaaService implements IChangePassword { }

@Service
public class BbbService implements IChangePassword { }
```

想通过@Autowired引入AaaService时

```
@Autowired
@Qualifier("aaaService") // 注意默认开头小写
private IChangePassword aaaService; // 父类属性，子类对象
```

- @PostConstruct

@PostConstruct是javax.annotation的注解，在方法上加该注解，会在项目启动时执行此方法。

这个方法跟 Constructor 类的构造方法一致。但是区别在 执行顺序。

spring有一个@Autowired注入依赖的注解，如果需要使用spring的依赖，那么init初始的方法就需在 @Autowired 之后执行。 所以需要用@PostConstruct
Constructor >> @Autowired >> @PostConstruct

- @PreDestroy

@PreDestroy 在...销毁前执行，跟 @PostConstruct 一致。

pojo数据相关

- JSON序列化死循环
- json序列化的对象存在双向引用会导致的无线递归(infinite recursion)问题。
- 列如： A对象引用了B对象，B对象又引用了A对象数组，那么就会造成这个问题。
- 使用@JsonBackReference标记在有多对一或者多对多关系的属性上即可解决这个问题。
- @JsonManagedReference - 可序列化

标记的属性会被序列化，属性默认都是会被序列化的。 反序列化（将json数据转换为对象时）。如果有@JsonManagedReference ，则不会自动注入到@JsonBackReference标注的属性中。

- @JsonBackReference - 不可序列化

添加此注解后，该属性不会在进行序列化操作

- @JsonIgnore - 直接取消该属性

@JsonIgnore：直接忽略某个属性，以断开无限递归，序列化或反序列化均忽略。当然如果标注在ge、set方法中，则可以分开控制，序列化对应的是get方法，反序列化对应的是set方法。

@JsonIgnore不会自动注入被忽略的属性值（父或子），这是它跟@JsonBackReference和@JsonManagedReference最大的区别。

- @Transient

表示该属性并非一个到数据库表的字段的映射,ORM框架将忽略该属性。

- @JsonInclude

JsonJsonInclude.Include.ALWAYS
默认策略，任何情况下都序列化该字段，和不写这个注解是一样的效果

JsonJsonInclude.Include.NON_NULL
最常用，即如果加该注解的字段为null,那么就不序列化这个字段了

JsonJsonInclude.Include.NON_ABSENT
布吉岛

JsonJsonInclude.Include.NON_EMPTY
这个属性包含NON_NULL，NON_ABSENT之后还包含如果字段为空也不序列化

JsonJsonInclude.Include.NON_DEFAULT
字段是默认值的话就不序列化

springmvc注解

- @RequestMapping

请求url的映射。 属性：
value： 请求的url地址 支持通配符匹配@RequestMapping(value="getUser/*");
即 localhost/getUser/hahaha 什么都都可以访问

path： 和 value属性使用一致

name： 注释，方便理解属性

method： 表示该方法仅仅处理哪些HTTP请求方式， 可以写一个数组 method={xx,xx}

params： 请求中必须包含params属性规定的参数时，才能执行该请求
@RequestMapping(value = "getUser",params = "flag")
请求中必须包含flag参数才能执行该请求，flag参数值不做要求
getUser?flag=xxx 此URL能够正常访问getUser()方法

header： 求中必须包含某些指定的header值，才能够让该方法处理请求
@RequestMapping(value = "getUser",headers = "Referer=http://www.xxx.com")
必须满足请求的header中包含了指定的"Referer"请求头和值为"http://www.xxx.com"时，才能执行请求

consumer： 指定处理请求提交内容类型(Content-Type) 列如：application/json、text/html时，能够让该方法处理请求

produces： 指定返回的内容类型，返回的内容类型必须是request请求头（Accept）中所包含的类型
produces属性还可以指定返回值的编码
produces = "application/json，charset=utf-8" 则指明返回utf-8编码

- @RequestParam： 用在方法参数前面

在方法参数中获取 请求参数 /user?abc=xxx 获取abc参数

- @ PathVariable - 路径变量

获取请求的路径变量参数 /user/{a} 获取a这个路径变量参数

全局异常注解

- @ControllerAdvice - 统一处理异常。

包含@Component。可以被扫描到。统一处理异常。
需要配合@ExceptionHandler使用。
当将异常抛到controller时,可以对异常进行统一处理,规定返回的json格式或是跳转到一个错误页面

- @ExceptionHandler（Exception.class） - 方法异常处理

用在方法上面表示遇到这个异常就执行以下Exception方法

```
//全局异常捕捉处理
@ControllerAdvice
public class CustomExceptionHandler {

    @ResponseBody
```

```
    @ExceptionHandler(value = Exception.class)
    public Map errorHandler(Exception ex) {
        Map map = new HashMap();
        map.put("code", 400);
        //判断异常的类型,返回不一样的返回值
        if(ex instanceof MissingServletRequestParameterException){
            map.put("msg","缺少必需参数：  "+((MissingServletRequestParameterException) ex).get
arameterName());
        }
        else if(ex instanceof MyException){
            map.put("msg","这是自定义异常");
        }
        return map;
    }
}
```

使用环境注解

APO切面注解

- @Aspect

把当前表示类 标识为一个切面供容器读取

- @Before

前置通知，在方法执行前执行，切面方法

- @After

后置通知，在方法执行之后进行 执行切面方法
无论是 代码正常返回，还是抛出错误。 这个后置通知 都会执行

- @AfterRunning

返回通知，在方法返回结果之后执行
执行成功调用
属性：
pointcut： 原始的切入点表达式需要写在里面
returning： 方法返回值

- @AfterThrowing

异常通知，在方法抛出错误之后 调用切面方法
执行失败调用
属性：
pointcut： 原始的切入点表达式需要写在里面
throwing： 错误信息返回值

- @Around

环绕通知， 围绕着方法执行
注入参数: ProceedingJoinPoint
pjp.proceed(); 表示切入点方法的运行
需要有返回值，Object ， 一般来代替切入点方法的返回值，或者直接

return pjp.proceed(); 来返回切入点方法的返回值

- @Pointcut

定义切入点
@Pointcut("execution(public java.util.Map com.example.demo.controller.TestService.getMap(
tring, Integer))")
private void p() {}

可以定义在类变量上， 然后在 @Before..等通知中 就可以调用。
@Before("p()")

ConditionalOnXXX

- 一般适合@Configuration一起使用
- @Configuration是配置一个配置类。而配置类中则进行了spring的一些配置。
- 配置类会自动被spring扫描并且加载到配置中，那么有时可能不是非要给配置成功，需要有一个条判断下是否加加载

@ConditionalOnWebApplication()
如果想要当前配置文件生效，必须有Web运行环境

@ConditionalOnNotWebApplication
当前不是web环境

@ConditionalOnClass(CharacterEncodingFilter.class)
如果想要当前配置文件生效，必须有CharacterEncodingFilter这个类

@ConditionalOnJava
系统的java版本是否符合要求

@ConditionalOnResource
类路径下是否存在指定资源文件

@ConditionalOnJava 系统的java版本是否符合要求
@ConditionalOnBean 容器中存在指定Bean；
@ConditionalOnMissingBean 容器中不存在指定Bean；
@ConditionalOnExpression 满足SpEL表达式指定
@ConditionalOnClass 系统中有指定的类
@ConditionalOnMissingClass 系统中没有指定的类
@ConditionalOnSingleCandidate 容器中只有一个指定的Bean，或者这个Bean是首选Bean
@ConditionalOnProperty 系统中指定的属性是否有指定的值
@ConditionalOnResource 类路径下是否存在指定资源文件
@ConditionalOnWebApplication 当前是web环境
@ConditionalOnNotWebApplication 当前不是web环境
@ConditionalOnJndi JNDI存在指定项

- 每一个中都有属性进行判断操作。具体操作请百度吧~

@ConfigurationProperties

- @ConfigurationProperties(prefix = "redis.config")

- 使用在实体类上，可以将实体类的信息 与 properties的属性对应起来

application.properties：

redis.config.maxTotal=5000

Redis配置类中：

```
@Component
@ConfigurationProperties(prefix = "redis.config")
@Data
public class RedisConfiguration {
    private int maxTotal;
}
```

maxTotal就可以跟 properties配置文件的 maxTotal对应起来

@EnableConfigurationProperties

- @EnableConfigurationProperties 用来将将 @ConfigurationProperties 的配置文件实体类进行册到IOC容器内。
- 这样@ConfigurationProperties 注解的实体类就不需要@Component来进行注册IOC

如果一个配置类只配置@ConfigurationProperties注解，而没有使用@Component，那么在IOC容中是获取不到properties 配置文件转化的bean。
@EnableConfigurationProperties 是把指定类的属性又注入了一次。

```
-----------------------------------------
@Data
@ConfigurationProperties(prefix = "yyzx.properties")
public class AnnotationDesc {
  // 该书写方式，属性值注入成功
}

@Slf4j
@Configuration
@EnableConfigurationProperties({
        AnnotationDesc.class,
        YyzxProperties2.class
})
public class SpringBootPlusConfig {

}
```

Cache缓存

- @EnableCaching

完成一些简单的缓存操作 - 加载Application启动类上
一般用于不会改动的数据上。如果数据容易改变，那么可能查的数据是之前缓存的，而造成数据不一错误

- @Cacheable

进行缓存，可以用在方法（返回值被缓存） 或 类上（所有方法缓存），
下次使用相同方法和相同参数都会直接从缓存中查去，不会再走方法
常用属性：
value： 设置一个组， 即各个方法通过value组来进行共同缓存。必填
key： 设置一个组件标识，用于区分同组下的不同缓存，不填系统默认生成
condition： 条件判断，可以填写SpEL表达式，进行条件判断，是否去查询缓存

- @CachePut

操作后缓存，一般用在Post，put方法中。方法一定会去执行，执行后返回的数据会进行缓存
三大属性与@cacheable基本一致

- @CacheEvict

@CacheEvict是用来标注在需要清除缓存元素的方法或类上的
常用属性：
value： 设置一个组， 即各个方法通过value组来进行共同缓存。必填
key： 设置一个组件标识，用于区分同组下的不同缓存，不填系统默认生成
condition： 条件判断，可以填写SpEL表达式，进行条件判断
allEntries： boolean类型，表示是否需要清除缓存中所有的元素，默认为false
beforeInvocation： 方法前后执行清除，默认为false在方法执行之后进行清除，