注解SpringBoot/spring

@SpringBootApplication:

包含@Configuration、@EnableAutoConfiguration、@ComponentScan通常用在主类上；

@Repository:

用于标注数据访问组件，即DAO组件；

@Service:

用于标注业务层组件；

@RestController:

用于标注控制层组件(如struts中的action)，包含@Controller和@ResponseBody；

@Controller:

用于标注是控制层组件，需要返回页面时请用@Controller而不是@RestController；

@Component:

泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注；

@ResponseBody:

表示该方法的返回结果直接写入HTTP response body中，一般在异步获取数据时使用，在使用@RequestMapping后，返回值通常解析为跳转路径，

加上@responsebody后返回结果不会被解析为跳转路径，而是直接写入HTTP response body中；比如异步获取json数据，加上@responsebody后，会直接返回json数据；

@RequestBody:

参数前加上这个注解之后，认为该参数必填。表示接受json字符串转为对象 List等；

@ComponentScan:

组件扫描。个人理解相当于，如果扫描到有@Component @Controller @Service等这些注解的类，则把这些类注册为bean*；

@Configuration:

指出该类是 Bean 配置的信息源，相当于XML中的，一般加在主类上；

@Bean:

相当于XML中的,放在方法的上面，而不是类，意思是产生一个bean,并交给spring管理；

@EnableAutoConfiguration:

让 Spring Boot 根据应用所声明的依赖来对 Spring 框架进行自动配置，一般加在主类上；

@AutoWired:

byType方式。把配置好的Bean拿来用，完成属性、方法的组装，它可以对类成员变量、方法及构造函数进行标注，完成自动装配的工作；

当加上（required=false）时，就算找不到bean也不报错；

@Qualifier:

当有多个同一类型的Bean时，可以用@Qualifier(“name”)来指定。与@Autowired配合使用；

@Resource(name=”name”,type=”type”):

没有括号内内容的话，默认byName。与@Autowired干类似的事；

@RequestMapping:

RequestMapping是一个用来处理请求地址映射的注解，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；

> 该注解有六个属性:
> 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。
> value:指定请求的实际地址，指定的地址可以是URI Template 模式
> method:指定请求的method类型， GET、POST、PUT、DELETE等

> consumes:指定处理请求的提交内容类型（Content-Type），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。

@GetMapping、@PostMapping等:

相当于@RequestMapping（value=”/”，method=RequestMethod.GetPostPutDelete等）。是个组合注解；

@RequestParam:

用在方法的参数前面。相当于 request.getParameter；

@PathVariable:

路径变量。如 RequestMapping(“user/get/mac/{macAddress}”)；

```
public String getByMacAddress(
@PathVariable(“macAddress”) String macAddress){
    //do something;
}
```

参数与大括号里的名字相同的话，注解后括号里的内容可以不填。

@Retention({RetentionPolicy.Runtime})

RetentionPolicy这个枚举类型的常量描述保留注释的各种策略，它们与元注释(@Retention)一起指定注释要保留多长时间

```
public enum RetentionPolicy {
    /**
     * 注释只在源代码级别保留，编译时被忽略
     */
    SOURCE,
    /**
     * 注释将被编译器在类文件中记录
     * 但在运行时不需要JVM保留。这是默认的
     * 行为.
     */
    CLASS,
    /**
     *注释将被编译器记录在类文件中
     *在运行时保留VM，因此可以反读。
     * @see java.lang.reflect.AnnotatedElement
```

```
     */
    RUNTIME
}
```

@Documented

Documented注解表明这个注释是由 javadoc记录的，在默认情况下也有类似的记录工具。 如果一个类型声明被注释了文档化，它的注释成为公共API的一部分。

@Inherited

允许子类继承父类的注解。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Inherited
public @interface DBTable {
    public String name() default "";
}
```

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface DBTable2 {
    public String name() default "";
}
```

```
@DBTable
class Super {
    private int superPrivateF;
    public int superPublicF;

    public Super(){
    }

    private int superPrivateM(){
        return 0;
    }
    public int superPubliceM(){
        return 0;
    }
}

@DBTable2
class Sub extends Super {
    private int subPrivateF;
```

```
    public int subPublicF;

    private Sub(){
    }
    public Sub(int i){
    }

    private int subPrivateM(){
        return 0;
    }
    public int subPubliceM(){
        return 0;
    }
}
```

```
public class DeclaredOrNot {
    public static void main(String[] args) {
        Class<Sub> clazz = Sub.class;

 System.out.println("============================Field=================
========");
        //public + 继承
        System.out.println(Arrays.toString(clazz.getFields()));//获取pub
属性+继承
        //all + 自身
        System.out.println(Arrays.toString(clazz.getDeclaredFields()));//
获取吱声声明

 System.out.println("============================Method================
=========");
        //public + 继承
        System.out.println(Arrays.toString(clazz.getMethods()));
        //all + 自身
        System.out.println(Arrays.toString(clazz.getDeclaredMethods()));

 System.out.println("============================Constructor===========
==============");
        //public + 自身
        System.out.println(Arrays.toString(clazz.getConstructors()));
        //all + 自身

 System.out.println(Arrays.toString(clazz.getDeclaredConstructors()));
```

```
 System.out.println("============================AnnotatedElement========
==================");
        //注解DBTable2是否存在于元素上
        System.out.println(clazz.isAnnotationPresent(DBTable2.class));
        //如果存在该元素的指定类型的注释DBTable2，则返回这些注释，否则返回
null。
        System.out.println(clazz.getAnnotation(DBTable2.class));
        //继承
        System.out.println(Arrays.toString(clazz.getAnnotations()));
        //自身

 System.out.println(Arrays.toString(clazz.getDeclaredAnnotations()));

    }
}
```

@Target:

@Target说明了Annotation所修饰的对象范围：Annotation可被用于 packages、types（类、接口、枚举、Annotation类型）、类型成员（方法、构造方法、成员变量、枚举值）、方法参数和本地变量（如循环变量、catch参数）。在Annotation类型的声明中使用了target可更加明晰其修饰的目标。

作用：用于描述注解的使用范围（即：被描述的注解可以用在什么地方）

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Target {
    /**
     * Returns an array of the kinds of elements an annotation type
     * can be applied to.
     * @return an array of the kinds of elements an annotation type
     * can be applied to
     */
    ElementType[] value();
}
```

取值(ElementType)有

```
public enum ElementType {
     /**用于描述类、接口(包括注解类型) 或enum声明 Class, interface
(including annotation type), or enum declaration */
    TYPE,

    /** 用于描述域 Field declaration (includes enum constants) */
    FIELD,

    /** 用于描述方法 Method declaration */
    METHOD,

    /** 用于描述参数 Formal parameter declaration */
    PARAMETER,

    /** 用于描述构造器 Constructor declaration */
    CONSTRUCTOR,

    /** 用于描述局部变量 Local variable declaration */
    LOCAL_VARIABLE,

    /** 用于注解类型上 Annotation type declaration */
    ANNOTATION_TYPE,

    /** 用于描述包 Package declaration */
    PACKAGE,

    /**
     * 能标注任何类型名称 用来标注类型参数 Type parameter declaration
     *
     * @since 1.8
     */
    TYPE_PARAMETER,

    /**
     * Use of a type
     *
     * @since 1.8
     */
    TYPE_USE
}
```

二.Jpa

@Entity:

@Table(name=”“):

表明这是一个实体类。一般用于jpa ，这两个注解一般一块使用，但是如果表名和实体类名相同的话，@Table可以省略；

@MappedSuperClass:

用在确定是父类的entity上。父类的属性子类可以继承；

@NoRepositoryBean:

一般用作父类的repository，有这个注解，spring不会去实例化该repository；

@Column:

如果字段名与列名相同，则可以省略；

@Id:

表示该属性为主键；

@GeneratedValue(strategy=GenerationType.SEQUENCE,generator = “repair_seq”):

表示主键生成策略是sequence（可以为Auto、IDENTITY、native等，Auto表示可在多个数据库间切换），指定sequence的名字是repair_seq；

@SequenceGeneretor(name = “repair_seq”, sequenceName = “seq_repair”, allocationSize = 1):

name为sequence的名称，以便使用，sequenceName为数据库的sequence名称，两个名称可以一致；

@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.

如果一个属性并非数据库表的字段映射,就务必将其标示为@Transient,否则,ORM框架默认其注解为@Basic；

@Basic(fetch=FetchType.LAZY):

标记可以指定实体属性的加载方式；

@JsonIgnore:

作用是json序列化时将java bean中的一些属性忽略掉,序列化和反序列化都受影响；

@JoinColumn(name="loginId"):

一对一：本表中指向另一个表的外键。

一对多：另一个表指向本表的外键。

@OneToOne

@OneToMany

@ManyToOne:

对应Hibernate配置文件中的一对一，一对多，多对一。

三.全局异常处理

@ControllerAdvice:

包含@Component。可以被扫描到。统一处理异常；

@ExceptionHandler(Exception.class):

用在方法上面表示遇到这个异常就执行以下方法。

四.springcloud

@EnableEurekaServer:

用在springboot启动类上，表示这是一个eureka服务注册中心；

@EnableDiscoveryClient:

用在springboot启动类上，表示这是一个服务，可以被注册中心找到；

@LoadBalanced:

开启负载均衡能力；

@EnableCircuitBreaker:

用在启动类上，开启断路器功能；

@HystrixCommand(fallbackMethod=”backMethod”):

用在方法上，fallbackMethod指定断路回调方法；

@EnableConfigServer:

用在启动类上，表示这是一个配置中心，开启Config Server；

@EnableZuulProxy:

开启zuul路由，用在启动类上；

@SpringCloudApplication:

包含

- @SpringBootApplication
- @EnableDiscovertyClient
- @EnableCircuitBreaker

分别是SpringBoot注解、注册服务中心Eureka注解、断路器注解。对于SpringCloud来说，这是每一微服务必须应有的三个注解，所以才推出了@SpringCloudApplication这一注解集合。