

## A

### @Accessors

链式编程使用。需要搭配@Getter和@Setter使用。主要有三个参数：

序号	参数名	介绍
1	chain	链式
2	fluent	流式（若无显示指定chain的值，也会把chain设置为true）
3	prefix	生成指定前缀的属性的getter与setter方法，并且生成的getter与setter方法时会去除前缀

我们发现prefix可以在生成get/set的时候，去掉xxx等prefix前缀，达到很好的一致性。但是，但是需要注意，因为此处age没有匹配上xxx前缀，所有根本就不给生成，所以使用的时候一定要注意。属性名没有一个以其中的一个前缀开头，则属性会被lombok完全忽略掉，并且还会产生一个警告。

### @Api

用在类上，该注解将一个Controller（Class）标注为一个swagger资源（API）。在默认情况下，Swagger-Core只会扫描解析具有@Api注解的类，而会自动忽略其他类别资源（JAX-RS endpoints, Servlets等等）的注解。该注解包含以下几个重要属性

tags API分组标签。具有相同标签的API将会被归并在一组内展示。

value 如果tags没有定义，value将作为Api的tags使用

description API的详细描述，在1.5.X版本之后不再使用，但实际发现在2.0.0版本中仍然可以使用

### @ApiOperation

在指定的（路由）路径上，对一个操作或HTTP方法进行描述。具有相同路径的不同操作会被归组为同一个操作对象。不同的HTTP请求方法及路径组合构成一个唯一操作。此注解的属性有：

value 对操作的简单说明，长度为120个字母，60个汉字。

notes 对操作的详细说明。

httpMethod HTTP请求的动作名，可选值有：“GET”，“HEAD”，“POST”，“PUT”，“DELETE”，“OPTIONS” and “PATCH”。

code 默认为200，有效值必须符合标准的[HTTP Status Code Definitions]。

### 实例

```

@AllArgsConstructor
@RestController
@RequestMapping("/api/category")
@Api(value = "/category", tags = "组件分类")
public class BizCategoryController {
    private IBizCategoryService bizCategoryService;
    @GetMapping("/list")
    @ApiOperation(value = "列表", notes = "分页列表")
    public R

```

## @Aspect

AOP为Aspect Oriented Programming的缩写，意为：面向切面编程，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术. AOP是OOP的延续，是软件开发中的一个热点，也是Spring框架中的一个重要内容，是函数式编程的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

在spring AOP中业务逻辑仅仅只关注业务本身，将日志记录，性能统计，安全控制，事务处理，异常处理等代码从业务逻辑代码中划分出来，通过对这些行为的分离，我们希望能将它们独立到非指导业务逻辑的方法中，进而改变这些行为的时候不影响业务逻辑的代码。

相关注解介绍：

**@Aspect:** 作用是把当前类标识为一个切面供容器读取

**@Pointcut:** Pointcut是植入Advice的触发条件。每个Pointcut的定义包括2部分，一是表达式，二是方法签名。方法签名必须是 public及void型。可以将Pointcut中的方法看作是一个被Advice引用的助记符，因为表达式不直观，因此我们可以通过方法签名的方式为 此表达式命名。因此Pointcut中的方法只需要方法签名，而不需要在方法体内编写实际代码。

**@Around:** 环绕增强，相当于MethodInterceptor

**@AfterReturning:** 后置增强，相当于AfterReturningAdvice，方法正常退出时执行

**@Before:** 标识一个前置增强方法，相当于BeforeAdvice的功能，相似功能的还有

**@AfterThrowing:** 异常抛出增强，相当于ThrowsAdvice

**@After:** final增强，不管是抛出异常或者正常退出都会执行  
使用pointcut代码

## @Aspect

```

public class AdviceTest {
    @Around("execution(* com.abc.service.*.many*(..))")
    public Object process(ProceedingJoinPoint point) throws Throwable {
        System.out.println("@Around: 执行目标方法之前...");
        //访问目标方法的参数：

```

```

Object[] args = point.getArgs();
if (args != null && args.length > 0 && args[0].getClass() == String.class) {
    args[0] = "改变后的参数1";
}
//用改变后的参数执行目标方法
Object returnValue = point.proceed(args);
System.out.println("@Around: 执行目标方法之后...");
System.out.println("@Around: 被织入的目标对象为: " + point.getTarget());
return "原返回值: " + returnValue + ", 这是返回结果的后缀";
}

@Before("execution(* com.abc.service.*.many*(..))")
public void permissionCheck(JoinPoint point) {
    System.out.println("@Before: 模拟权限检查...");
    System.out.println("@Before: 目标方法为: " +
        point.getSignature().getDeclaringTypeName() +
        "." + point.getSignature().getName());
    System.out.println("@Before: 参数为: " + Arrays.toString(point.getArgs()));
    System.out.println("@Before: 被织入的目标对象为: " + point.getTarget());
}

@AfterReturning(pointcut="execution(* com.abc.service.*.many*(..))",
    returning="returnValue")
public void log(JoinPoint point, Object returnValue) {
    System.out.println("@AfterReturning: 模拟日志记录功能...");
    System.out.println("@AfterReturning: 目标方法为: " +
        point.getSignature().getDeclaringTypeName() +
        "." + point.getSignature().getName());
    System.out.println("@AfterReturning: 参数为: " +
        Arrays.toString(point.getArgs()));
    System.out.println("@AfterReturning: 返回值为: " + returnValue);
    System.out.println("@AfterReturning: 被织入的目标对象为: " + point.getTarget());
}

@After("execution(* com.abc.service.*.many*(..))")
public void releaseResource(JoinPoint point) {
    System.out.println("@After: 模拟释放资源...");
    System.out.println("@After: 目标方法为: " +
        point.getSignature().getDeclaringTypeName() +
        "." + point.getSignature().getName());
    System.out.println("@After: 参数为: " + Arrays.toString(point.getArgs()));
    System.out.println("@After: 被织入的目标对象为: " + point.getTarget());
}

```

```
}
```

使用annotation代码

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.METHOD })
public @interface SMSAndMailSender {
    /*短信模板String格式化串*/
    String value() default "";
    String smsContent() default "";
    String mailContent() default "";
    /*是否激活发送功能*/
    boolean isActive() default true;
    /*主题*/
    String subject() default "";
}

//切面类
@Aspect
@Component("smsAndMailSenderMonitor")
public class SMSAndMailSenderMonitor {
    private Logger logger = LoggerFactory.getLogger(SMSAndMailSenderMonitor.class);
    /**
     * 在所有标记了@SMSAndMailSender的方法中切入
     * @param joinPoint
     * @param result
     */
    @AfterReturning(value="@annotation(com.trip.demo.SMSAndMailSender)",
returning="result")//有注解标记的方法，执行该后置返回
    public void afterReturning(JoinPoint joinPoint , Object result//注解标注的方法返回值) {
        MethodSignature ms = (MethodSignature) joinPoint.getSignature();
        Method method = ms.getMethod();
        boolean active = method.getAnnotation(SMSAndMailSender.class).isActive();
        if (!active) {
            return;
        }

        String smsContent = method.getAnnotation(SMSAndMailSender.class).smsContent();
        String mailContent = method.getAnnotation(SMSAndMailSender.class).mailContent();
        String subject = method.getAnnotation(SMSAndMailSender.class).subject();
    }
    /**
```

```

    * 在抛出异常时使用
    * @param joinPoint
    * @param ex
    */
@AfterThrowing(value="@annotation(com.trip.order.monitor.SMSAndMailSender)", throwing =
"ex")
    public void afterThrowing(JoinPoint joinPoint, Throwable ex//注解标注的方法抛出的异常) {
        MethodSignature ms = (MethodSignature) joinPoint.getSignature();
        Method method = ms.getMethod();
        String subject = method.getAnnotation(SMSAndMailSender.class).subject();
    }
}
//实体类中使用该注解标注方法
@Service("testService ")
public class TestService {
    @Override
    @SMSAndMailSender(smsContent = "MODEL_SUBMIT_SMS", mailContent =
"MODEL_SUPPLIER_EMAIL", subject = "MODEL_SUBJECT_EMAIL")
    public String test(String param) {
        return "success";
    }
}

```

注意，记得在配置文件中加上：

```
<aop:aspectj-autoproxy proxy-target-class="true"/>
```

## @Autowired

顾名思义，就是自动装配。其作用是替代Java代码里面的getter/setter与bean属性中的property。如果有属性需要对外提供的话，getter应当予以保留。引入@Autowired注解，先看一下spring配置文件怎么写：

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xmlns="http://www.springframework.org/schema/beans"
4     xmlns:context="http://www.springframework.org/schema/context"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6         http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
7         http://www.springframework.org/schema/context
8         http://www.springframework.org/schema/context/spring-context-4.2.xsd">
9

```

```

10    <context:component-scan base-package="com.zxt" />
11
12    <bean id="school" class="com.zxt.bean.School" />
13    <bean id="teacher" class="com.zxt.uu.Teacher" />
14    <bean id="student" class="com.zxt.uu.Student" />
15
16 </beans>

```

注意第10行，为了实现bean的自动载入，必须配置spring的扫描器。在base-package指明一个包：

```
<context:component-scan base-package= "com.zxt" />
```

表明com.zxt包及其子包中，如果某个类的头上带有特定的注解@Component、@Repository、@Service或@Controller，就会将这个对象作为Bean注入进spring容器。

看到第12行，原来school里面应当注入两个属性teacher、student，现在不需要注入了。再看下，School.java也很简练，把getter/setter都可以去掉：

```

public class School{
    @Autowired
    private Teacher teacher;
    @Autowired
    private Student student;
    public String toString(){
        return teacher + "\n" + student;
    }
}

```

这里@Autowired注解的意思就是，当Spring发现@Autowired注解时，将自动在代码上下文中找到与其匹配（默认是类型匹配）的Bean，并自动注入到相应的地方去。

B

### @Bean

@Bean是一个方法级别上的注解，主要用在@Configuration注解的类里，也可以用在@Component注解的类里。作用为注册bean对象。

@Bean注解在返回实例的方法上，如果未通过@Bean指定bean的名称，则默认与标注的方法名相同；@Bean注解默认作用域为单例singleton作用域，可通过@Scope(“prototype”)设置为原型作用域；既然@Bean的作用是注册bean对象，那么完全可以使用@Component、@Controller、@Service、@Repository等注解注册bean，当然需要配置@ComponentScan注解进行自动扫描。

### @Resource

作用：用来装配bean，可以写在字段上或者setter方法上。默认按照名称来装配注入，找不到名称时按照类型来装配注入。获取bean，测试代码：

```
@Service
public class DemoService {
    @Bean(name = "userDemo")
    public User getUser() {
        User user = new User("付恒", "男", 22);
        return user;
    }
}
```

首先用@Bean注入一个bean方法，名称为：userDemo

```
@RestController
public class UserController {
    @Resource(name = "userDemo")
    User user;
    @RequestMapping(value = "/index", method = RequestMethod.GET)

    public String index() {
        return user.getName()+"--"+user.getSex()+"--"+user.getAge();
    }
}
```

通过@Resource获取名称为userDemo的值。

## C

### @cacheable

@Cacheable可以标记在一个方法上，也可以标记在一个类上。当标记在一个方法上时表示该方法是支持缓存的，当标记在一个类上时则表示该类所有的方法都是支持缓存的。对于一个支持缓存的方法，Spring会在其被调用后将其返回值缓存起来，以保证下次利用同样的参数来执行该方法时可以直接从缓存中获取结果，而不需要再次执行该方法。Spring在缓存方法的返回值时是以键值对进行缓存的，值就是方法的返回结果，至于键的话，Spring又支持两种策略，默认策略和自定义策略，这个稍后会进行说明。需要注意的是当一个支持缓存的方法在对象内部被调用时是不会触发缓存功能的。@Cacheable可以指定三个属性，value、key和condition。

value属性指定Cache名称。value属性是必须指定的，其表示当前方法的返回值是会被缓存在哪个Cache上的，对应Cache的名称。其可以是一个Cache也可以是多多个Cache，当需要指定多个Cache时其是一个数组。

@Cacheable("cache1")//Cache是发生在cache1上的

```
public User find(Integer id) {
    return null;
}
```

@Cacheable({ “cache1”, “cache2” })//Cache是发生在cache1和cache2上的

```
public User find(Integer id) {
    return null;
}
```

使用key属性自定义key。ey属性是用来指定Spring缓存方法的返回结果时对应的key的。该属性支持SpringEL表达式。当我们没有指定该属性时，Spring将使用默认策略生成key。

@Cacheable(value=“ users” , key=“ #id” )

```
public User find(Integer id) {
    return null;
}
```

@Cacheable(value=“ users” , key=“ #p0” )

```
public User find(Integer id) {
    return null;
}
```

@Cacheable(value=“ users” , key=“ #user.id” )

```
public User find(User user) {
    return null;
}
```

@Cacheable(value=“ users” , key=“ #p0.id” )

```
public User find(User user) {
    return null;
}
```

除了上述使用方法参数作为key之外，Spring还为我们提供了一个root对象可以用来生成key。通过该root对象我们可以获取到以下信息。

属性名称	描述	示例
methodName	当前方法名	#root.methodName
method	当前方法	#root.method.name
target	当前被调用的对象	#root.target
targetClass	当前被调用的对象的class	#root.targetClass
args	当前方法参数组成的数组	#root.args[0]
caches	当前被调用的方法使用的Cache	#root.caches[0].name

当我们要使用root对象的属性作为key时我们也可以将“#root”省略，因为Spring默认使用的就是root对象的属性。如：



```
@Cacheable(value={ "users" , "xxx" }, key=" caches[1].name" )
public User find(User user) {
    return null;
}
```

condition属性指定发生的条件. 有的时候我们可能并不希望缓存一个方法所有的返回结果。通过condition属性可以实现这一功能。condition属性默认为空，表示将缓存所有的调用情形。其值是通过SpringEL表达式来指定的，当为true时表示进行缓存处理；当为false时表示不进行缓存处理，即每次调用该方法时该方法都会执行一次。

```
@Cacheable(value={ "users" }, key=" #user.id" , condition=" #user.id%2==0" )
public User find(User user) {
    System.out.println( "find user by user " + user);
    return user;
}
```

## @CacheEvict

@CacheEvict是用来标注在需要清除缓存元素的方法或类上的。当标记在一个类上时表示其中所有的方法的执行都会触发缓存的清除操作。@CacheEvict可以指定的属性有value、key、condition、allEntries和beforeInvocation。其中value、key和condition的语义与@Cacheable对应的属性类似。即value表示清除操作是发生在哪些Cache上的（对应Cache的名称）；key表示需要清除的是哪个key，如未指定则会使用默认策略生成的key；condition表示清除操作发生的条件。下面我们来介绍一下新出现的两个属性allEntries和beforeInvocation。

allEntries属性.allEntries是boolean类型，表示是否需要清除缓存中的所有元素。默认为false，表示不需要。当指定了allEntries为true时，Spring Cache将忽略指定的key。有的时候我们需要Cache一下清除所有的元素，这比一个一个清除元素更有效率。

```
@CacheEvict(value=" users" , allEntries=true)
public void delete(Integer id) {
    System.out.println( "delete user by id: " + id);
}
```

beforeInvocation属性. 清除操作默认是在对应方法成功执行之后触发的，即方法如果因为抛出异常而未能成功返回时也不会触发清除操作。使用beforeInvocation可以改变触发清除操作的时间，当我们指定该属性值为true时，Spring会在调用该方法之前清除缓存中的指定元素。

```
@CacheEvict(value=" users" , beforeInvocation=true)
public void delete(Integer id) {
    System.out.println( "delete user by id: " + id);
}
```

```
}
```

### @Caching

@Caching注解可以让我们在一个方法或者类上同时指定多个Spring Cache相关的注解。其拥有三个属性：cacheable、put和evict，分别用于指定@Cacheable、@CachePut和@CacheEvict。

```
@Caching(cacheable = @Cacheable( "users" ), evict = { @CacheEvict( "cache2" ),
@CacheEvict(value = "cache3", allEntries = true) })
public User find(Integer id) {
    return null;
}
```

### @CachePut

在支持Spring Cache的环境下，对于使用@Cacheable标注的方法，Spring在每次执行前都会检查Cache中是否存在相同key的缓存元素，如果存在就不再执行该方法，而是直接从缓存中获取结果进行返回，否则才会执行并将返回结果存入指定的缓存中。@CachePut也可以声明一个方法支持缓存功能。与@Cacheable不同的是使用@CachePut标注的方法在执行前不会去检查缓存中是否存在之前执行过的结果，而是每次都会执行该方法，并将执行结果以键值对的形式存入指定的缓存中。

@CachePut也可以标注在类上和方法上。使用@CachePut时我们可以指定的属性跟@Cacheable是一样的。

@CachePut( "users" )//每次都会执行方法，并将结果存入指定的缓存中

```
public User find(Integer id) {
    return null;
}
```

### @Cleanup

这个注解用在变量前面，可以保证此变量代表的资源会被自动关闭，默认是调用资源的close()方法，如果该资源有其它关闭方法，可使用@Cleanup( "methodName" )来指定要调用的方法，就用输入输出流来举个例子吧：

```
public static void main(String[] args) throws IOException {
    @Cleanup InputStream in = new FileInputStream(args[0]);
    @Cleanup OutputStream out = new FileOutputStream(args[1]);
    byte[] b = new byte[1024];
    while (true) {
        int r = in.read(b);
        if (r == -1) break;
        out.write(b, 0, r);
    }
}
```

```
}
```

实际效果相当于：

```
public static void main(String[] args) throws IOException {
    InputStream in = new FileInputStream(args[0]);
    try {
        OutputStream out = new FileOutputStream(args[1]);
        try {
            byte[] b = new byte[10000];
            while (true) {
                int r = in.read(b);
                if (r == -1) break;
                out.write(b, 0, r);
            }
        } finally {
            if (out != null) {
                out.close();
            }
        }
    } finally {
        if (in != null) {
            in.close();
        }
    }
}
```

## @Component

注解                      含义

@Component      最普通的组件，可以被注入到spring容器进行管理

@Repository              作用于持久层

@Service                      作用于业务逻辑层

@Controller              作用于表现层（spring-mvc的注解）

实例

```
@Component("userManager")
```

```
public class UserManagerImpl implements UserManager {
    private UserDao userDao;
    public UserDao getUserDao() {
        return userDao;
    }
}
```

```

    }

    @Resource
    public void setUserDao(UserDao userDao) {
        this.userDao = userDao;
    }

    public boolean exists(Users u) {
        return userDao.checkUserExistsWithName(u.getUsername());
    }

    public void add(Users u) {
        userDao.save(u);
    }
}

```

在持久层、业务层和控制层分别采用 @Repository、@Service 和 @Controller 对分层中的类进行注释，而用 @Component 对那些比较中立的类进行注释。这里就是说把这个类交给Spring管理，重新起个名字叫 userManager，由于不好说这个类属于哪个层面，就用@Component

### @ComponentScan

根据定义的扫描路径，把符合扫描规则的类装配到spring的bean容器中。

Spring ComponentScan注解有以下特性：

自定扫描路径下边带有@Controller, @Service, @Repository, @Component注解加入spring容器

通过includeFilters加入扫描路径下没有以上注解的类加入spring容器

通过excludeFilters过滤出不用加入spring容器的类

自定义增加了@Component注解的注解方式

源码

```

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
@Documented
@Repeatable(ComponentScans.class)
public @interface ComponentScan {
    @AliasFor("basePackages")
    String[] value() default {};
    @AliasFor("value")
    String[] basePackages() default {};
    Class[] basePackageClasses() default {};
    Class nameGenerator() default BeanNameGenerator.class;
    Class scopeResolver() default AnnotationScopeMetadataResolver.class;
    ScopedProxyMode scopedProxy() default ScopedProxyMode.DEFAULT;
}

```

```

String resourcePattern() default "**/*.class";
boolean useDefaultFilters() default true;
ComponentScan.Filter[] includeFilters() default {};
ComponentScan.Filter[] excludeFilters() default {};
boolean lazyInit() default false;
@Retention(RetentionPolicy.RUNTIME)
@Target({})
public @interface Filter {
    FilterType type() default FilterType.ANNOTATION;
    @AliasFor("classes")
    Class[] value() default {};
    @AliasFor("value")
    Class[] classes() default {};
    String[] pattern() default {};
}
}

```

basePackages与value:? 用于指定包的路径, 进行扫描

basePackageClasses: 用于指定某个类的包的路径进行扫描

nameGenerator: bean的名称的生成器

useDefaultFilters: 是否开启对@Component, @Repository, @Service, @Controller的类进行检测

includeFilters: 包含的过滤条件 FilterType.ANNOTATION: 按照注解过

滤, FilterType.ASSIGNABLE\_TYPE: 按照给定的类型, FilterType.ASPECTJ: 使用ASPECTJ表达

式, FilterType.REGEX: 正则, FilterType.CUSTOM: 自定义规则

excludeFilters: 排除的过滤条件, 用法和includeFilters一样

## 注解方式

### 1. 扫描包

```
@ComponentScan(basePackages = " ") //单个
```

```
@ComponentScan(basePackages = { "xxx", "aaa", "..."} ) //多个
```

注意: 可以省略 "basePackages ="

```
@Configuration
```

```
@ComponentScan("com.5lgjie.spring.service")
```

```
public class MyConfig {}
```

```
@Configuration
```

```
@ComponentScan("com.5lgjie.spring.dao", "com.5lgjie.spring.service")
```

```
public class MyConfig {}
```

```
@Configuration
```

```
@ComponentScan("com.5lgjie.spring.*") //通配符匹配所有的包
```

```
public class MyConfig {}
```

## 2. 扫描类

```
@ComponentScan(basePackageClasses = “”) //单个
```

```
@ComponentScan(basePackageClasses = { “aaa”, “bbb”, “...” }) //多个
```

注意：不可以省略 “basePackageClasses =”

```
@Configuration
```

```
@ComponentScan(basePackageClasses = HelloController.class)
```

```
public class MyConfig {  
}
```

### @Configuration

从Spring3.0, @Configuration用于定义配置类, 可替换xml配置文件, 被注解的类内部包含有一个或多个被@Bean注解的方法, 这些方法将会被AnnotationConfigApplicationContext或AnnotationConfigWebApplicationContext类进行扫描, 并用于构建bean定义, 初始化Spring容器。

注意: @Configuration注解的配置类有如下要求:

@Configuration不可以是final类型;

@Configuration不可以是匿名类;

嵌套的@Configuration必须是静态类。

@Configuration等价于<Beans></Beans>

@Bean等价于<Bean></Bean>

@ComponentScan等价于<context:component-scan base-package=” com.dxz.demo” />

### @ConfigurationProperties

在 Spring Boot 项目中, 我们将大量的参数配置在application.properties 或 application.yml 文件中, 通过 @ConfigurationProperties 注解, 我们可以方便的获取这些参数值在Spring Boot 中, @ConfigurationProperties有三种使用场景, 而通常情况下我们使用的最多的只是其中的一种场景。

#### 场景一

使用@ConfigurationProperties和@Component注解到bean定义类上, 这里@Component代指同一类实例化Bean的注解。基本使用实例如下:

```
// 将类定义为一个bean的注解, 比如 @Component, @Service, @Controller, @Repository
```

```
// 或者 @Configuration
```

```
@Component
```

```
// 表示使用配置文件中前缀为user1的属性的值初始化该bean定义产生的bean实例的同名属性
```

```
// 在使用时这个定义产生的bean时，其属性name会是Tom
@ConfigurationProperties(prefix = "user1")
public class User {
    private String name;
    // 省略getter/setter方法
}
```

对应application.properties配置文件内容如下：

```
user1.name=Eden
```

在此种场景下，当Bean被实例化时，@ConfigurationProperties会将对应前缀的后面的属性与Bean对象的属性匹配。符合条件则进行赋值。

## 场景二

使用@ConfigurationProperties和@Bean注解在配置类的Bean定义方法上。以数据源配置为例：

```
@Configuration
public class DataSourceConfig {
    @Primary
    @Bean(name = "primaryDataSource")
    @ConfigurationProperties(prefix="spring.datasource.primary")
    public DataSource primaryDataSource() {
        return DataSourceBuilder.create().build();
    }
}
```

这里便是将前缀为“spring.datasource.primary”的属性，赋值给DataSource对应的属性值。

@Configuration注解的配置类中通过@Bean注解在某个方法上将方法返回的对象定义为一个Bean，并使用配置文件中相应的属性初始化该Bean的属性。

## 场景三

使用@ConfigurationProperties注解到普通类，然后再通过@EnableConfigurationProperties定义为Bean。

```
@ConfigurationProperties(prefix = "user1")
public class User {
    private String name;
    // 省略getter/setter方法
}
```

```
}
```

这里User对象并没有使用@Component相关注解。而该User类对应的使用形式如下：

```
@SpringBootApplication
@EnableConfigurationProperties({User.class})
public class Application {
    public static void main(String[] args) throws Exception {
        SpringApplication.run(Application.class, args);
    }
}
```

上述代码中，通过@EnableConfigurationProperties对User进行实例化时，便会使用到@ConfigurationProperties的功能，对属性进行匹配赋值。

## @CrossOrigin

出于安全原因，浏览器禁止Ajax调用驻留在当前原点之外的资源。例如，当你在一个标签中检查你的银行账户时，你可以在另一个选项卡上拥有EVILL网站。来自EVILL的脚本不能够对你的银行API做出Ajax请求（从你的帐户中取出钱！）使用您的凭据。跨源资源共享（CORS）是由大多数浏览器实现的W3C规范，允许您灵活地指定什么样的跨域请求被授权，而不是使用一些不太安全和不太强大的策略，如IFRAME或JSONP。

一、跨域(CORS)支持：Spring Framework 4.2 GA为CORS提供了第一类支持，使您比通常的基于过滤器的解决方案更容易和更强大地配置它。所以springMVC的版本要在4.2或以上版本才支持@CrossOrigin

二、使用方法：

### 1、controller配置CORS

1.1、controller方法的CORS配置，您可以向@RequestMapping注解处理程序方法添加一个@CrossOrigin注解，以便启用CORS（默认情况下，@CrossOrigin允许在@RequestMapping注解中指定的所有源和HTTP方法）：

```
@RestController
@RequestMapping("/account") public class AccountController {

    @CrossOrigin
    @GetMapping("/{id}") public Account retrieve(@PathVariable Long id) { // ...
    }

    @DeleteMapping("/{id}") public void remove(@PathVariable Long id) { // ...
    }
}
```



```
}  
}
```

其中@CrossOrigin中的2个参数:

origins : 允许可访问的域列表

maxAge: 准备响应前的缓存持续的最大时间（以秒为单位）。

### 1.2、为整个controller启用@CrossOrigin

```
@CrossOrigin(origins = "http://domain2.com", maxAge = 3600)  
@RestController  
@RequestMapping("/account") public class AccountController {  
  
    @GetMapping("/{id}") public Account retrieve(@PathVariable Long id) { // ...  
}  
  
    @DeleteMapping("/{id}") public void remove(@PathVariable Long id) { // ...  
}  
}
```

在这个例子中, 对于retrieve()和remove()处理方法都启用了跨域支持, 还可以看到如何使用@CrossOrigin属性定制CORS配置。

### 1.3、同时使用controller和方法级别的CORS配置, Spring将合并两个注释属性以创建合并的CORS配置。

```
@CrossOrigin(maxAge = 3600)  
@RestController  
@RequestMapping("/account") public class AccountController {  
  
    @CrossOrigin(origins = "http://domain2.com")  
    @GetMapping("/{id}") public Account retrieve(@PathVariable Long id) { // ...  
}  
  
    @DeleteMapping("/{id}") public void remove(@PathVariable Long id) { // ...  
}  
}
```

### 1.4、如果您正在使用Spring Security, 请确保在Spring安全级别启用CORS, 并允许它利用Spring MVC级别定义的配置。

```

@EnableWebSecurity public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override protected void configure(HttpSecurity http) throws Exception {
        http.cors().and()...
    }
}

```

## 2、全局CORS配置

除了细粒度、基于注释的配置之外，您可能需要定义一些全局CORS配置。这类似于使用筛选器，但可以声明为Spring MVC并结合细粒度@CrossOrigin配置。默认情况下，所有origins and GET, HEAD and POST methods是允许的。

JavaConfig使整个应用程序的CORS简化为：

```

@Configuration
@EnableWebMvc public class WebConfig extends WebMvcConfigurerAdapter {

    @Override public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**");
    }
}

```

如果您正在使用Spring Boot，建议将WebMvcConfigurer bean声明如下：

```

@Configuration public class MyConfiguration {

    @Bean public WebMvcConfigurer corsConfigurer() { return new WebMvcConfigurerAdapter() {
        @Override public void addCorsMappings(CorsRegistry registry) {
            registry.addMapping("/**");
        }
    };
}

```

您可以轻松地更改任何属性，以及仅将此CORS配置应用到特定的路径模式：

```

@Override public void addCorsMappings(CorsRegistry registry) {

```

```

registry.addMapping("/api/**")
    .allowedOrigins("http://domain2.com")
    .allowedMethods("PUT", "DELETE")
    .allowedHeaders("header1", "header2", "header3")
    .exposedHeaders("header1", "header2")
    .allowCredentials(false).maxAge(3600);
}

```

如果您正在使用Spring Security，请确保在Spring安全级别启用CORS，并允许它利用Spring MVC级别定义的配置。

### 3、XML命名空间

还可以将CORS与MVC XML命名空间配置。

a、如果整个项目所有方法都可以访问，则可以这样配置；此最小XML配置使CORS在/\*\*路径模式具有与JavaConfig相同的缺省属性：

```

<mvc:cors>
    <mvc:mapping path="/**" />
</mvc:cors>

```

其中\* 表示匹配到下一层；\*\*\*\* 表示后面不管有多少层，都能匹配。\*\*如：

```

<mvc:cors>
    <mvc:mapping path="/api/*/"/>
</mvc:cors>

```

这个可以匹配到的路径有：

```

/api/aaa
/api/bbbb

```

不能匹配的：

```

/api/aaa/bbb

```

因为\* 只能匹配到下一层路径，如果想后面不管多少层都可以匹配，配置如下：

```

<mvc:cors>

```

```
    <mvc:mapping path="/api/**"/>
</mvc:cors>
```

注：其实就是一个(\*)变成两个(\*\*)

b、也可以用定制属性声明几个CORS映射：

```
<mvc:cors>
    <mvc:mapping path="/api/**" allowed-origins="http://domain1.com, http://domain2.com"
allowed-methods="GET, PUT" allowed-headers="header1, header2, header3" exposed-
headers="header1, header2" allow-credentials="false" max-age="123" />
    <mvc:mapping path="/resources/**" allowed-origins="http://domain1.com" />
</mvc:cors>
```

请求路径有/api/，方法示例如下：

```
@RequestMapping("/api/crossDomain")
@ResponseBody public String crossDomain(HttpServletRequest req, HttpServletResponse res,
String name) {
    .....
    .....
}
```

c、如果使用Spring Security，不要忘记在Spring安全级别启用CORS：

```
<http>
    <!-- Default to Spring MVC's CORS configuration -->
    <cors /> ... </http>
```

4、How does it work?

CORS请求（包括预选的带有选项方法）被自动发送到注册的各种HandlerMapping。它们处理CORS准备请求并拦截CORS简单和实际请求，这得益于CorsProcessor实现（默认情况下默认DefaultCorsProcessor处理器），以便添加相关的CORS响应头（如Access-Control-Allow-Origin）。CorsConfiguration 允许您指定CORS请求应该如何处理：允许origins, headers, methods等。

a、AbstractHandlerMapping#setCorsConfiguration() 允许指定一个映射，其中有几个CorsConfiguration 映射在路径模式上，比如/api/\*\*。

b、子类可以通过重写AbstractHandlerMapping类的getCorsConfiguration(Object, HttpServletRequest)方法来提供自己的CorsConfiguration。

c、处理程序可以实现 CorsConfigurationSource接口（如ResourceHttpRequestHandler），以便为每个请求提供一个CorsConfiguration。

## 5、基于过滤器的CORS支持

作为上述其他方法的替代，Spring框架还提供了CorsFilter。在这种情况下，不使用@CrossOrigin或WebMvcConfigurer#addCorsMappings(CorsRegistry)，例如，可以在Spring Boot应用程序中声明如下的过滤器：

```
@Configuration public class MyConfiguration {
    @Bean public FilterRegistrationBean corsFilter() {
        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        CorsConfiguration config = new CorsConfiguration();
        config.setAllowCredentials(true);
        config.addAllowedOrigin("http://domain1.com");
        config.addAllowedHeader("*");
        config.addAllowedMethod("*");
        source.registerCorsConfiguration("/*", config);
        FilterRegistrationBean bean = new FilterRegistrationBean(new CorsFilter(source));
        bean.setOrder(0); return bean;
    }
}
```

## 三、spring注解@CrossOrigin不起作用的原因

1、是springMVC的版本要在4.2或以上版本才支持@CrossOrigin

2、非@CrossOrigin没有解决跨域请求问题，而是不正确的请求导致无法得到预期的响应，导致浏览器端提示跨域问题。

3、在Controller注解上方添加@CrossOrigin注解后，仍然出现跨域问题，解决方案之一就是：

在@RequestMapping注解中没有指定Get、Post方式，具体指定后，问题解决。

类似代码如下：

```
@CrossOrigin
@RestController public class person{
```

```

    @RequestMapping(method = RequestMethod.GET) public String add() { // 若干代码
    }
}

```

## D

### @Data

@Data 注解的主要作用是提高代码的简洁，使用这个注解可以省去代码中大量的get()、set()、toString()等方法；要使用 @Data 注解要先引入lombok，lombok 是什么，它是一个工具类库，可以用简单的注解形式来简化代码，提高开发效率。

在maven中添加依赖

```

<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.4</version>
    <scope>provided</scope>
</dependency>

```

在编译器中添加插件

这里以IDEA为例，在setting的plugin里搜索lombok plugin，安装插件。

使用

直接在相应的实体类上加上@Data注解即可；

常用的几个注解：

@Data ： 注在类上，提供类的get、set、equals、hashCode、canEqual、toString方法

@AllArgsConstructor ： 注在类上，提供类的全参构造

@NoArgsConstructor ： 注在类上，提供类的无参构造

@Setter ： 注在属性上，提供 set 方法

@Getter ： 注在属性上，提供 get 方法

@EqualsAndHashCode ： 注在类上，提供对应的 equals 和 hashCode 方法

@Log4j/@Slf4j ： 注在类上，提供对应的 Logger 对象，变量名为 log

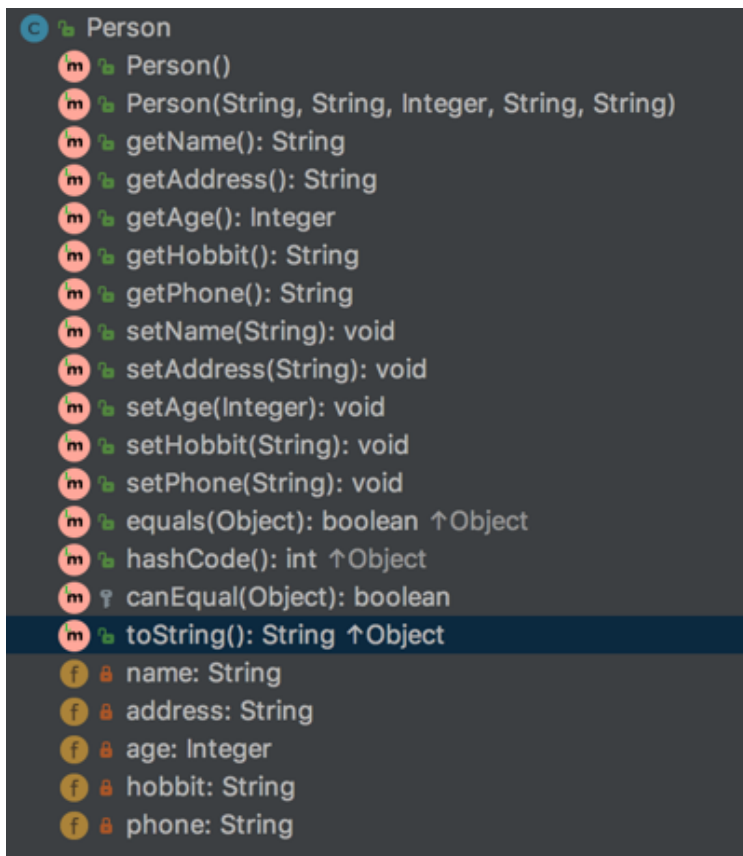
用@Data的写法：

@Data

@AllArgsConstructor

```
@NoArgsConstructor
public class Person {
    private String name;
    private String address;
    private Integer age;
    private String hobbit;
    private String phone;
}
```

自动生成相关的方法：



原理

Lombok本质上就是一个实现了“JSR 269 API”的程序。在使用javac的过程中，它产生作用的具体流程如下：

javac对源代码进行分析，生成了一棵抽象语法树（AST）

运行过程中调用实现了“JSR 269 API”的Lombok程序

此时Lombok就对第一步骤得到的AST进行处理，找到@Data注解所在类对应的语法树（AST），然后修改该语法树（AST），增加getter和setter方法定义的相应树节点

javac使用修改后的抽象语法树（AST）生成字节码文件，即给class增加新的节点（代码块）

优缺点

优点：能通过注解的形式自动生成构造器、getter/setter、equals、hashCode、toString等方法，提高了一定的开发效率

让代码变得简洁，不用过多的去关注相应的方法

属性做修改时，也简化了维护为这些属性所生成的getter/setter方法等

缺点：不支持多种参数构造器的重载

虽然省去了手动创建getter/setter方法的麻烦，但大大降低了源代码的可读性和完整性，降低了阅读源代码的舒适度

像 lombok 这种插件，已经不仅仅是插件了，它在编译器编译时通过操作AST（抽象语法树）改变字节码生成，变相的说它就是在改变java语法，它改变了你编写源码的方式，它不像 spring 的依赖注入一样是运行时的特性，而是编译时的特性。如果一个项目有非常多这样的插件，会极大的降低阅读源代码的舒适度。

lombok 只是省去了一些人工生成代码的麻烦，但是这些getter/setter等等的方法，用IDE的快捷键也可很方便的生成。况且，有时通过给getter/setter加一点点业务代码（但通常不建议这么加），能极大的简化某些业务场景的代码。

用还是不用，这中间如何取舍，自然是要看项目的需要，灵活运用。

E

### @EnableAspectJAutoProxy

如果我们在每一个方法上都加上一套计算时间的逻辑，将会消耗大量的重复工作，并且等到不需要用的时候又需要一个一个删除，这是很恶心人的事情。这时不妨试用下spring的aop。

### 实现步骤

#### 1. 引入依赖

```
```xml
```

```
    org.springframework.boot  
    spring-boot-starter-aop
```

```
```
```

#### 1. 在启动类加上@EnableAspectJAutoProxy注解

```
```java
```

```
@SpringBootApplication
```

```
@EnableAspectJAutoProxy
```

```
public class NoteApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(NoteApplication.class, args);  
    }  
}
```

```
}
```

```
```
```

#### 1. 创建一个需要被添加日志的类，以下以controller为例

```
```java
```



```

@RestController()
@RequestMapping("/testRest" )
public class TestRest {
    @GetMapping("/test")
    public String test(@RequestParam("name") String name) {
        System.out.println("test:" + name);
        return "result:" + name;
    }
}
...

```

## 1. 创建AOP类

```

@Aspect
@Component
public class AopLog {
    @Pointcut(value = "execution(* com.yuyu.learning.note.controller.TestRest.test(..)) && args(name)")
    public void point(String name) {
    }

    /**
     * 方法执行前
     *
     * @param joinPoint
     * @param name      参数
     */
    @Before(value = "point(name)")
    public void beforeMethod(JoinPoint joinPoint, String name) {
        System.out.println("目标方法名为:" + joinPoint.getSignature().getName());
        System.out.println("目标方法所属类的简单类名:" +
joinPoint.getSignature().getDeclaringType().getSimpleName());
        System.out.println("目标方法所属类的类名:" +
joinPoint.getSignature().getDeclaringTypeName());
        System.out.println("目标方法声明类型:" +
Modifier.toString(joinPoint.getSignature().getModifiers()));
        //获取传入目标方法的参数
        Object[] args = joinPoint.getArgs();
        for (int i = 0; i < args.length; i++) {
            System.out.println("第" + (i + 1) + "个参数为:" + args[i]);
        }
        System.out.println("被代理的对象:" + joinPoint.getTarget());
        System.out.println("代理对象自己:" + joinPoint.getThis());
    }
}

```

```

        System.out.println("beforeMethod\t" + name);
    }
}

/**
 * 方法执行后
 *
 * @param joinPoint
 * @param name      参数
 */
@After("point(name)")
public void afterMethod(JoinPoint joinPoint, String name) {
    System.out.println("调用了后置通知\t" + name);
}

/**
 * 返回通知
 *
 * @param joinPoint
 * @param result    返回内容
 * @param name      传入参数
 */
@AfterReturning(value = "point(name)", returning = "result")
public void afterReturningMethod(JoinPoint joinPoint, Object result, String name) {
    System.out.println("调用了返回通知\t" + result);
}

/**
 * 异常通知
 *
 * @param joinPoint
 * @param e
 * @param name
 */
@AfterThrowing(value = "point(name)", throwing = "e")
public void afterReturningMethod(JoinPoint joinPoint, Exception e, String name) {
    System.out.println("调用了异常通知");
}

/**
 * 环绕通知
 *
 * @param pjp
 * @param name
 * @return
 */

```

```

    * @throws Throwable
    */
    @Around("point(name)")
    public Object Around(ProceedingJoinPoint pjp, String name) throws Throwable {
        System.out.println("around执行方法之前");
        //        Object object = pjp.proceed();
        Object object = pjp.proceed(new Object[] {"新参数"});
        System.out.println("around执行方法之后--返回值" + object);
        return object;
    }
}

```

## 详细解释

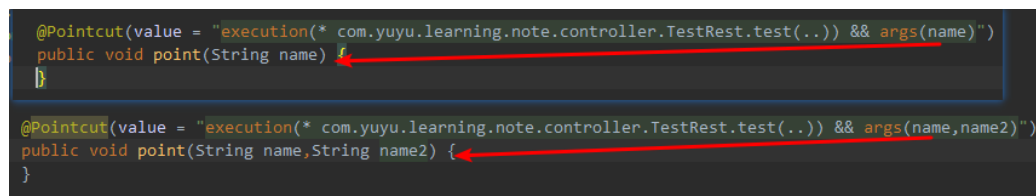
1. @Aspect 声明这个类是切面类

2. @Component 声明为组件，将类交给spring管理

3. @Pointcut 声明切点

3.1 execution(\* com.yuyu.learning.note.controller.TestRest.test(...))这个是声明方法的位置，可以使用正则表达式，我这里精确匹配到了这个test方法

3.2 args(name) 制定参数为name，如果有两个参数就是args(name1,name2)；这时下面方法的参数需要个数/参数名称匹配才可以



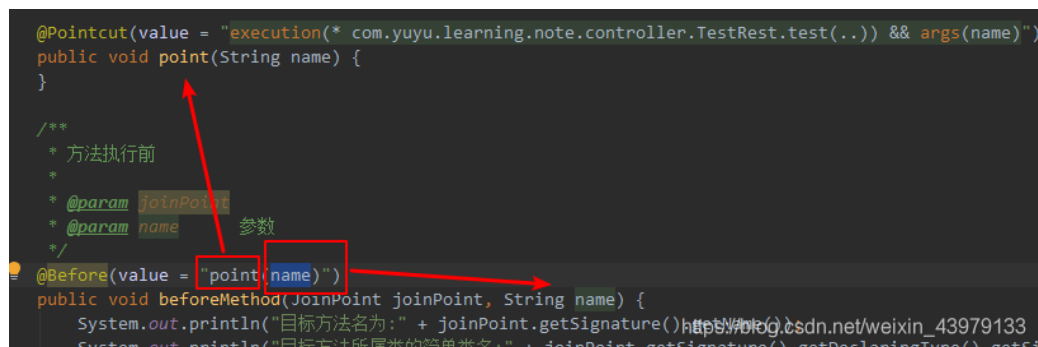
```

@Pointcut(value = "execution(* com.yuyu.learning.note.controller.TestRest.test(..)) && args(name)")
public void point(String name) {
}

@Pointcut(value = "execution(* com.yuyu.learning.note.controller.TestRest.test(..)) && args(name,name2)")
public void point(String name,String name2) {
}

```

4. value = "point(name)" point(arg)指向了定义切点的方法，里面的参数必须和本方法中的参数一致



```

@Pointcut(value = "execution(* com.yuyu.learning.note.controller.TestRest.test(..)) && args(name)")
public void point(String name) {
}

/**
 * 方法执行前
 *
 * @param joinPoint
 * @param name 参数
 */
@Before(value = "point(name)")
public void beforeMethod(JoinPoint joinPoint, String name) {
    System.out.println("目标方法名为:" + joinPoint.getSignature().getName());
    System.out.println("目标方法所属类的简单类名:" + joinPoint.getSignature().getDeclaringType().getSimpleName());
}

```

5. @Around("point(name)") 特别说下around通知，在这个位置是可以修改传入方法的参数值的

@Around("point(name)")

```

public Object Around(ProceedingJoinPoint pjp, String name) throws Throwable {
    System.out.println("around执行方法之前");
    Object object = pjp.proceed(new Object[] {"新参数"});
    System.out.println("around执行方法之后--返回值" + object);
    return object;
}

```

```

}
// 这个位置调用了可以传入参数数组的方法
    public Object proceed(Object[] args) throws Throwable;

```

## @EnableAutoConfiguration

源码

```

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";
    Class[] exclude() default {};
    String[] excludeName() default {};
}

```

@EnableAutoConfiguration实现的关键在于引入了AutoConfigurationImportSelector，其核心逻辑为selectImports方法，借助AutoConfigurationImportSelector，它可以帮助SpringBoot应用将所有符合条件的@Configuration配置都加载到当前SpringBoot创建并使用的IoC容器。

原理

当springboot扫描到@EnableAutoConfiguration注解时则会将spring-boot-autoconfigure.jar/META-INF/spring.factories文件中org.springframework.boot.autoconfigure.EnableAutoConfiguration对应的value里的所有xxxConfiguration类加载到IOC容器中。spring.factories文件里每一个xxxAutoConfiguration文件一般都会有下面的条件注解：

```

@ConditionalOnClass : classpath中存在该类时起效
@ConditionalOnMissingClass : classpath中不存在该类时起效
@ConditionalOnBean : DI容器中存在该类型Bean时起效
@ConditionalOnMissingBean : DI容器中不存在该类型Bean时起效
@ConditionalOnSingleCandidate : DI容器中该类型Bean只有一个或@Primary的只有一个时起效
@ConditionalOnExpression : SpEL表达式结果为true时
@ConditionalOnProperty : 参数设置或者值一致时起效
@ConditionalOnResource : 指定的文件存在时起效
@ConditionalOnJndi : 指定的JNDI存在时起效
@ConditionalOnJava : 指定的Java版本存在时起效
@ConditionalOnWebApplication : Web应用环境下起效

```

`@ConditionalOnNotWebApplication` : 非Web应用环境下起效

SpringBoot中`EnableAutoConfiguration`实现的关键在于引入了`AutoConfigurationImportSelector`，其核心逻辑为`selectImports`方法，逻辑大致如下：

1. 从配置文件META-INF/spring.factories加载所有可能用到的自动配置类；
2. 去重，并将`exclude`和`excludeName`属性携带的类排除；
3. 过滤，将满足条件（`@Conditional`）的自动配置类返回；

## `@EnableCaching`

缓存注解使它生效只需要轻松两步：

1. 配置类上开启缓存注解支持：`@EnableCaching`
2. 向容器内至少放置一个`CacheManager`类型的Bean

`@EnableCaching`

`@Configuration`

```
public class CacheConfig {  
    @Bean  
    public ConcurrentMapCacheManager cacheManager() {  
        ConcurrentMapCacheManager cacheManager = new ConcurrentMapCacheManager();  
        //cacheManager.setStoreByValue(true); //true表示缓存一份副本，否则缓存引用  
        return cacheManager;  
    }  
}
```

## `@EnableGlobalMethodSecurity`

开启基于方法的安全认证机制，也就是说在web层的controller启用注解机制的安全确认，

```
@ApiOperation(value = "获取用户列表", httpMethod = "GET")  
@GetMapping  
@PreAuthorize("hasAuthority('admin')")  
//Authentication authentication, 当前用户信息  
public ResponseEntity<PageResult<UserVO>> list(Authentication authentication, UserDTO  
userDTO, @Min(1) @RequestParam(defaultValue = "1") Integer pageNo, @Max(100) @Min(5)  
@RequestParam(defaultValue = "10") Integer pageSize) {  
    System.out.println(authentication);  
    PageInfo<UserVO> listByPage = userService.getListByPage(userDTO, pageNo, pageSize);  
    PageResult<UserVO> result = new PageResult<>();  
    result.setTotal(listByPage.getTotal());  
    result.setData(listByPage.getList());  
}
```

```

        result.setTotalPage(listByPage.getPages());
        result.setPageNO(pageNo);
        result.setPageSize(pageSize);
        return ResponseEntity.ok(result);
    }

```

只有加了@EnableGlobalMethodSecurity(prePostEnabled=true) 那么在上面使用的  
@PreAuthorize(“hasAuthority(‘admin’ )”)才会生效

### @EnableTransactionManagement

所有的数据访问技术都有事务处理机制，这些技术提供了API用来开启事务、提交事务以完成数据操纵，或者在发生错误的时候回滚数据。Spring支持声明式事务，这是基于AOP实现的。

Spring提供了一个\*\*@EnableTransactionManagement\*\* 注解以在配置类上开启声明式事务的支持。添加该注解后，Spring容器会自动扫描被\*\*@Transactional\*\*注解的方法和类。简单开启事务管理：

### @SpringBootApplication

```

@EnableTransactionManagement // 开启注解事务管理，等价于xml配置方式的 <tx:annotation-driven />

public class DemoApplication {
    //业务代码
}

```

### @EnableScheduling

定时任务在配置类上添加@EnableScheduling开启对定时任务的支持，在相应的方法上添加@Scheduled声明需要执行的定时任务。

其中Scheduled注解中有以下几个参数：

1. cron
2. zone
3. fixedDelay和fixedDelayString
4. fixedRate和fixedRateString
5. initialDelay和initialDelayString

1. cron是设置定时执行的表达式，如 0 0/5 \* \* \* ?每隔五分钟执行一次

2. zone表示执行时间的时区

3. fixedDelay 和fixedDelayString 表示一个固定延迟时间执行，上个任务完成后，延迟多长时间执行

4. fixedRate 和fixedRateString表示一个固定频率执行，上个任务开始后，多长时间后开始执行

5. initialDelay 和initialDelayString表示一个初始延迟时间，第一次被调用前延迟的时间

配置类

```
@Configuration
```

```
@ComponentScan({"com.xingguo.logistics.service.aspect"})
```

```
@EnableScheduling
```

```
public class AopConfig{
```

```
}
```

service类

```
@Service
```

```
public class TestService2 {
```

```
    private static final SimpleDateFormat format = new SimpleDateFormat("HH:mm:ss");
```

```
    //初始延迟1秒，每隔2秒
```

```
    @Scheduled(fixedRateString = "2000",initialDelay = 1000)
```

```
    public void testFixedRate() {
```

```
        System.out.println("fixedRateString, 当前时间: " +format.format(new Date()));
```

```
    }
```

```
    //每次执行完延迟2秒
```

```
    @Scheduled(fixedDelayString="2000")
```

```
    public void testFixedDelay() {
```

```
        System.out.println("fixedDelayString, 当前时间: " +format.format(new Date()));
```

```
    }
```

```
    //每隔3秒执行一次
```

```
    @Scheduled(cron="0/3 * * * * ?")
```

```
    public void testCron() {
```

```
        System.out.println("cron, 当前时间: " +format.format(new Date()));
```

```
    }
```

## 测试类

```
public class TestController {  
    public static void main(String[] args) {  
        AnnotationConfigApplicationContext context = new  
AnnotationConfigApplicationContext(AopConfig.class);  
    }  
}
```

测试结果:

```
fixedDelayString,当前时间: 15:30:25  
fixedRateString,当前时间: 15:30:26  
cron,当前时间: 15:30:27  
fixedDelayString,当前时间: 15:30:27  
fixedRateString,当前时间: 15:30:28  
fixedDelayString,当前时间: 15:30:29  
cron,当前时间: 15:30:30  
fixedRateString,当前时间: 15:30:30  
fixedDelayString,当前时间: 15:30:31  
fixedRateString,当前时间: 15:30:32
```

## @EnableSwagger2

1. pom.xml中引入依赖

```
<dependency>  
    <groupId>io.springfox</groupId>  
    <artifactId>springfox-swagger2</artifactId>  
    <version>2.2.2</version>  
</dependency>  
<dependency>  
    <groupId>io.springfox</groupId>  
    <artifactId>springfox-swagger-ui</artifactId>  
    <version>2.2.2</version>  
</dependency>
```

方式一: Application.java中引入 @EnableSwagger2来启动swagger注解

```
@SpringBootApplication // 组件扫描  
@EnableSwagger2  
public class Application {  
  
}
```



方式二：创建Swagger2配置类

```
@Configuration
@EnableSwagger2
@ConditionalOnProperty(prefix = "hr", name = "swagger-open", havingValue = "true")
public class SwaggerConfig {

    @Bean
    public Docket createRestApi() {
        return new Docket(DocumentationType.SWAGGER_2)
            .apiInfo(apiInfo())
            .select()
            .apis(RequestHandlerSelectors.withMethodAnnotation(ApiOperation.class))    //
这里采用包含注解的方式来确定要显示的接口

        // .apis(RequestHandlerSelectors.basePackage("com.fz.hr.modules.system.controller"))    //这
里采用包扫描的方式来确定要显示的接口
            .paths(PathSelectors.any())
            .build();
    }

    private ApiInfo apiInfo() {
        return new ApiInfoBuilder()
            .title("HR Doc")
            .description("HR Api文档")
            .contact("WH")
            .version("2.0")
            .build();
    }
}
```

如上代码所示，通过@Configuration注解，让Spring来加载该类配置。再通过@EnableSwagger2注解来启用Swagger2。再通过createRestApi函数创建Docket的Bean之后，apiInfo()用来创建该Api的基本信息（这些基本信息会展现在文档页面中）。select()函数返回一个ApiSelectorBuilder实例用来控制哪些接口暴露给Swagger来展现，本例采用指定扫描的包路径来定义，Swagger会扫描该包下所有Controller定义的API，并产生文档内容（除了被@ApiIgnore`指定的请求。

接口注解

```

@RestController
@RequestMapping("/user")
@Api("UserController相关api")
public class UserController {

    @Autowired
    private UserService userService;


    @ApiOperation("获取用户信息")
    @ApiImplicitParams({

        @ApiImplicitParam(paramType="header", name="username", dataType="String", required=true, value="用户的姓名", defaultValue="zhaojigang"),

        @ApiImplicitParam(paramType="query", name="password", dataType="String", required=true, value="用户的密码", defaultValue="wangna")
    })
    @ApiResponses({
        @ApiResponse(code=400, message="请求参数没填好"),
        @ApiResponse(code=404, message="请求路径没有或页面跳转路径不对")
    })
    @RequestMapping(value="/getUser", method=RequestMethod.GET)
    public User getUser(@RequestHeader("username") String username,
        @RequestParam("password") String password) {
        return userService.getUser(username, password);
    }
}

```

访问地址： 访问：<http://localhost:8080/swagger-ui.html>

 swagger

default (/v2/api-docs) ▼

api\_key

Explore

### HR Doc

HR Api文档

Created by WH

菜单表相关api : 菜单表相关API

Show/Hide | List Operations | Expand Operations

GET

/system/menu/list

获取所有菜单信息

Response Class (Status 200)

Model | Model Schema

```
[
  {
    "code": "string",
    "icon": "string",
    "id": 0,
    "ismenu": 0,
    "isopen": 0,
  }
]
```

Response Content Type \*/\* ▼

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out!

Hide Response

参考:

<https://swagger.io/>

## @EqualsAndHashCode/@ToString

1. 此注解会生成equals(Object other) 和 hashCode() 方法。
2. 它默认使用非静态，非瞬态的属性
3. 可通过参数exclude排除一些属性
4. 可通过参数of指定仅使用哪些属性
5. 它默认仅使用该类中定义的属性且不调用父类的方法
6. 可通过callSuper=true解决上一点问题。让其生成的方法中调用父类的方法。
7. 另：@Data 相当于 @Getter @Setter @RequiredArgsConstructor @ToString @EqualsAndHashCode这5个注解的合集。

通过官方文档，可以得知，当使用@Data注解时，则有了@EqualsAndHashCode注解，那么就会在此类中存在equals(Object other) 和 hashCode() 方法，且不会使用父类的属性，这就导致了可能的问题。比如，有多个类有相同的部分属性，把它们定义到父类中，恰好id（数据库主键）也在父类中，那么就会存在部分对象在比较时，它们并不相等，却因为lombok自动生成的equals(Object other) 和 hashCode() 方法判定为相等，从而导致出错。修复此问题的方法很简单：

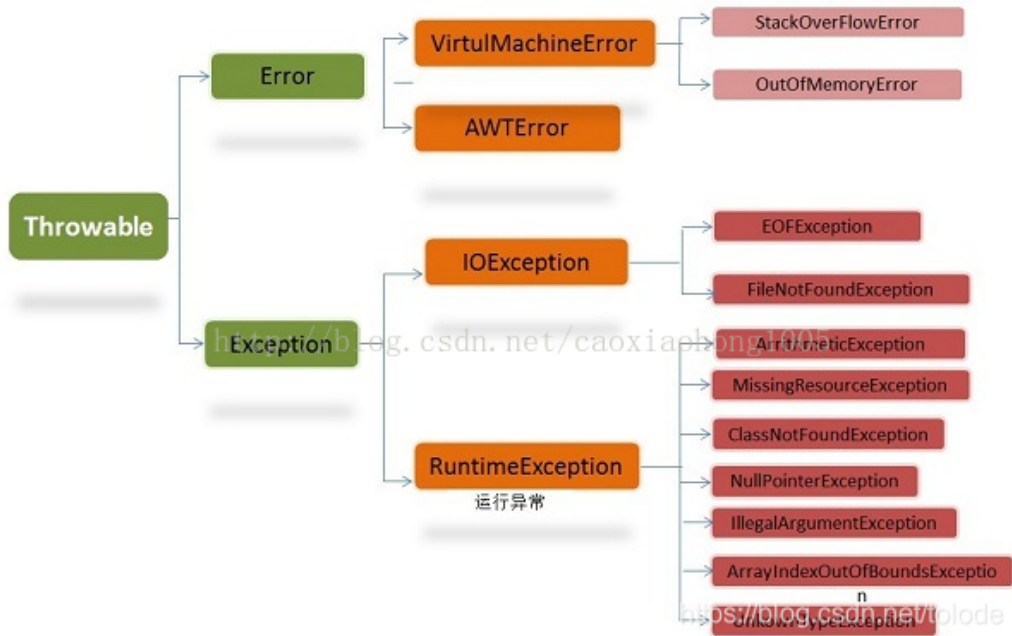
1. 使用@Getter @Setter @ToString代替@Data并且自定义equals(Object other) 和 hashCode() 方法，比如有些类只需要判断主键id是否相等即足矣。
2. 或者使用在使用@Data时同时加上@EqualsAndHashCode(callSuper=true)注解。

@ExceptionHandler

@ExceptionHandler注解我们一般是来自定义异常的。可以认为它是一个异常拦截器（处理器）。

1

异常间的层次关系



一：极简测试，一共4个类：

- 1、一个SpringBoot启动类
- 2、一个控制层
- 3、一个异常处理类
- 4、一个service类

启动类：ExceptionHandlerdemoApplication

@SpringBootApplication

```
public class ExceptionhandlerdemoApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.run(ExceptionhandlerdemoApplication.class, args);  
    }  
}
```

异常处理类

@RestControllerAdvice

```

public class GlobalExceptionHandler {

    private final Logger logger = LogManager.getLogger(GlobalExceptionHandler.class);

    @ExceptionHandler({Exception.class})    //申明捕获那个异常类
    public String ExceptionDemo(Exception e) {
        logger.error(e.getMessage(), e);
        return "自定义异常返回";
    }

}

```

控制层TestControll

```

@Controller
@RequestMapping("/fu")
public class TestControll {
    private Logger logger = LoggerFactory.getLogger(getClass());

    @Autowired
    private UserInfoSerimpl userInfoSerimpl;

    @ResponseBody
    @RequestMapping("/test")
    public String test() {
        logger.info("11111111111");
        userInfoSerimpl.saveUserInfo();

        logger.info("2222222222");
        return "sdfsfs";
    }
}

```

业务层：UserInfoSerimpl

```

@Service("userInfoService")
public class UserInfoSerimpl {
    private Logger logger = LoggerFactory.getLogger(UserInfoSerimpl.class);

    public void saveUserInfo() {

```

```

        logger.error("获取用户信息失败");
        test1();
        logger.info("ddddddddd");
    }

    private void test1() {
        logger.error("test1 失败");
        throw new RuntimeException();
    }
}

```

测试: <http://localhost:8080/fu/test>

输出: 自定义异常返回

二、关于ExceptionHandler定义的拦截器之间的优先级. 在GlobalExceptionHandler类中定义两个拦截器

```

@ExceptionHandler({RuntimeException.class})    //申明捕获那个异常类
public String RuntimeExceptionDemo(Exception e) {
    logger.error(e.getMessage(), e);
    return "运行时异常返回";
}

@ExceptionHandler({NumberFormatException.class})    //申明捕获那个异常类
public String NumberFormatExceptionDemo(Exception e) {
    logger.error(e.getMessage(), e);
    return "数字转换异常返回";
}

```

在UserInfoSerimpl的test1方法中定义一个数字转换异常,  
这个异常在运行时异常之前出现。

```

private void test1() {
    logger.error("test1 失败");
    String a = "123a";
    Integer b = Integer.valueOf(a);
    throw new RuntimeException();
}

```

测试: <http://localhost:8080/fu/test>

输出: 自定义异常返回

结论: 自定义的异常越详细, 得到的异常结果就越详细。

三: 为什么我们不直接使用一个Exception

1:Exception什么的异常太过广泛, 我们直接抛出所有异常信息, 对用户而言是非常不友好的。

2:在事务管理里, 如果我们自定义的异常继承的是Exception, 则事务无效。如果我们是继承RuntimeException, 则不会出现这个问题。

G

**@Getter/@Setter**

这一对注解从名字上就很好理解, 用在成员变量前面, 相当于为成员变量生成对应的get和set方法, 同时还可以为生成的方法指定访问修饰符, 当然, 默认为public, 直接来看下面的简单的例子:

```
public class Programmer{  
    @Getter  
    @Setter  
    private String name;  
  
    @Setter(AccessLevel.PROTECTED)  
    private int age;  
  
    @Getter(AccessLevel.PUBLIC)  
    private String language;  
}
```

实际效果相当于:

```
public class Programmer{  
    private String name;  
    private int age;  
    private String language;  
  
    public void setName(String name){  
        this.name = name;  
    }  
  
    public String getName(){  
        return name;  
    }  
}
```

```

    protected void setAge(int age) {
        this.age = age;
    }

    public String getLanguage() {
        return language;
    }
}

```

这两个注解还可以直接用在类上，可以为此类里的所有非静态成员变量生成对应的get和set方法。

J

### @JsonInclude

比如说我有个场景，返回前端的实体类中如果某个字段为空的话那么就不返回这个字段了，如果我们平时遇到这个问题，那么真的该脑壳疼了。幸亏有我们今天的主角，这个注解就是用来在实体类序列化成json的时候在某些策略下，加了该注解的字段不去序列化该字段。

策略

介绍

JsonJsonInclude.Include.ALWAYS	这个是默认策略，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	这个最常用，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	这个包含NON_NULL，即为null的时候不序列化，第二种情况是下面的英文
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。这个也比较常用
JsonJsonInclude.Include.NON_DEFAULT	如果字段是默认值的话就不序列化。
JsonJsonInclude.Include.CUSTOM	见官网
JsonJsonInclude.Include.USE_DEFAULTS	见官网

这里我们以如果为null则不序列化举例说明

```

public class User {
    @JsonInclude(JsonInclude.Include.NON_NULL)
    private String username;
    private String password;
    private Integer age;
}

```



## 测试代码

```
public static void main(String[] args) throws IOException {  
    User user = new User();  
    ObjectMapper objectMapper = new ObjectMapper();  
    String value = objectMapper.writeValueAsString(user);  
    System.out.println(value);  
}
```

## 结果

```
{ "password":null, "age":null}
```

可以看出我们在username上面加了这儿注解并且指定为null的时候不序列化，结果里面没有序列化username这个属性。password和age字段没有加属性，正常序列化成功。

## @JsonIgnore

1. 作用：在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。
2. 使用方法：一般标记在属性或者方法上，返回的json数据即不包含该属性。
3. 场景模拟：

需要把一个List转换成json格式的数据传递给前台。但实体类中基本属性字段的值都存储在快照属性字段中。此时我可以在业务层中做处理，把快照属性字段的值赋给实体类中对应的基本属性字段。最后，我希望返回的json数据中不包含这两个快照字段，那么在实体类中快照属性上加注解@JsonIgnore，那么最后返回的json数据，将不会包含goodsInfo和extendsInfo两个属性值。

```
public class HistoryOrderBean {  
  
    //基本属性字段  
    private String insurantName;  
    private String insuranceName;  
    private String insurancePrice;  
    private String insurancePicture;  
    private String insuranceLimit;  
  
    //快照属性字段  
    @JsonIgnore  
    private String goodsInfo;        //快照基本信息
```

```

@JsonIgnore
private String extendsInfo;    //快照扩展属性信息
}

```

#### 4. 注解失效:

如果注解失效，可能是因为你使用的是fastJson，尝试使用对应的注解来忽略字段，注解为：  
@JSONField(serialize = false)，使用方法一样。

## L

### @Log

这个注解用在类上，可以省去从日志工厂生成日志对象这一步，直接进行日志记录，具体注解根据日志工具的不同而不同，同时，可以在注解中使用topic来指定生成log对象时的类名。不同的日志注解总结如下（上面是注解，下面是实际作用）：

```

@CommonsLog
private static final org.apache.commons.logging.Log log =
org.apache.commons.logging.LogFactory.getLog(LogExample.class);

@JBossLog
private static final org.jboss.logging.Logger log =
org.jboss.logging.Logger.getLogger(LogExample.class);

@Log
private static final java.util.logging.Logger log =
java.util.logging.Logger.getLogger(LogExample.class.getName());

@Log4j
private static final org.apache.log4j.Logger log =
org.apache.log4j.Logger.getLogger(LogExample.class);

@Log4j2
private static final org.apache.logging.log4j.Logger log =
org.apache.logging.log4j.LogManager.getLogger(LogExample.class);

@Slf4j
private static final org.slf4j.Logger log =
org.slf4j.LoggerFactory.getLogger(LogExample.class);

@XSlf4j
private static final org.slf4j.ext.XLogger log =
org.slf4j.ext.XLoggerFactory.getXLogger(LogExample.class);

```

## lombok相关注解

### @ToString/@EqualsAndHashCode

这两个注解也比较好理解，就是生成toString，equals和hashCode方法，同时后者还会生成一个canEqual方法，用于判断某个对象是否是当前类的实例，生成方法时只会使用类中的非静态和非transient成员变

量，这些都比较好理解，就不举例子了。

当然，这两个注解也可以添加限制条件，例如用@ToString(exclude={ “param1” , “param2” })来排除param1和param2两个成员变量，或者用@ToString(of={ “param1” , “param2” })来指定使用param1和param2两个成员变量，@EqualsAndHashCode注解也有同样的用法。

#### @NoArgsConstructor/@RequiredArgsConstructor /@AllArgsConstructor

这三个注解都是用在类上的，第一个和第三个都很好理解，就是为该类产生无参的构造方法和包含所有参数的构造方法，第二个注解则使用类中所有带有@NonNull注解的或者带有final修饰的成员变量生成对应的构造方法，当然，和前面几个注解一样，成员变量都是非静态的，另外，如果类中含有final修饰的成员变量，是无法使用@NoArgsConstructor注解的。

三个注解都可以指定生成的构造方法的访问权限，同时，第二个注解还可以用

@RequiredArgsConstructor(staticName=“methodName”)的形式生成一个指定名称的静态方法，返回一个调用相应的构造方法产生的对象，下面来看一个生动鲜活的例子：

```
@RequiredArgsConstructor(staticName = "sunsfan")
@AllArgsConstructor(access = AccessLevel.PROTECTED)
@NoArgsConstructor
public class Shape {
    private int x;
    @NonNull
    private double y;
    @NonNull
    private String name;
}
```

实际效果相当于：

```
public class Shape {
    private int x;
    private double y;
    private String name;

    public Shape() {
    }

    protected Shape(int x, double y, String name) {
        this.x = x;
        this.y = y;
        this.name = name;
    }
}
```

```

    public Shape(double y,String name) {
        this.y = y;
        this.name = name;
    }

    public static Shape sunsfan(double y,String name) {
        return new Shape(y,name);
    }
}

```

### @Data/@Value

呃!!

@Data注解综合了3,4,5和6里面的@RequiredArgsConstructor注解，其中@RequiredArgsConstructor使用了类中的带有@NonNull注解的或者final修饰的成员变量，它可以使用

@Data(staticConstructor="methodName")来生成一个静态方法，返回一个调用相应的构造方法产生的对象。这个例子就也省略了吧...

@Value注解和@Data类似，区别在于它会把所有成员变量默认定义为private final修饰，并且不会生成set方法。

### @SneakyThrows

这个注解用在方法上，可以将方法中的代码用try-catch语句包裹起来，捕获异常并在catch中用Lombok.sneakyThrow(e)把异常抛出，可以使用@SneakyThrows(Exception.class)的形式指定抛出哪种异常，很简单的注解，直接看个例子：

```

public class SneakyThrows implements Runnable {
    @SneakyThrows(UnsupportedEncodingException.class)
    public String utf8ToString(byte[] bytes) {
        return new String(bytes, "UTF-8");
    }

    @SneakyThrows
    public void run() {
        throw new Throwable();
    }
}

```

实际效果相当于：

```

public class SneakyThrows implements Runnable {
    @SneakyThrows(UnsupportedEncodingException.class)
    public String utf8ToString(byte[] bytes) {
        try{
            return new String(bytes, "UTF-8");
        }catch(UnsupportedEncodingException uee){
            throw Lombok.sneakyThrow(uee);
        }
    }
    @SneakyThrows
    public void run() {
        try{
            throw new Throwable();
        }catch(Throwable t){
            throw Lombok.sneakyThrow(t);
        }
    }
}

```

### @Synchronized

这个注解用在类方法或者实例方法上，效果和synchronized关键字相同，区别在于锁对象不同，对于类方法和实例方法，synchronized关键字的锁对象分别是类的class对象和this对象，而@synchronized得锁对象分别是私有静态final对象LOCK和私有final对象LOCK和私有final对象lock，当然，也可以自己指定锁对象，例子也很简单，往下看：

```

public class Synchronized {
    private final Object readLock = new Object();
    @Synchronized
    public static void hello() {
        System.out.println("world");
    }
    @Synchronized
    public int answerToLife() {
        return 42;
    }
    @Synchronized("readLock")
    public void foo() {
        System.out.println("bar");
    }
}

```

实际效果相当于：

```
public class Synchronized {
    private static final Object $LOCK = new Object[0];
    private final Object $lock = new Object[0];
    private final Object readLock = new Object();
    public static void hello() {
        synchronized($LOCK) {
            System.out.println("world");
        }
    }
    public int answerToLife() {
        synchronized($lock) {
            return 42;
        }
    }
    public void foo() {
        synchronized(readLock) {
            System.out.println("bar");
        }
    }
}
```

N

@NotBlank/@NotEmpty/@NotNull

在前段向后端提交较多数据时，我们一般都会遇到字段校验的问题，使用Spring的字段验证很省事，一般会使用@NotNull、@NotEmpty、@NotBlank这三个东西，但使用的时候后端接收参数一定要注意接收参数的数据类型。

注解	限制	描述	null	""	""	"hello"
@NotNull	不能为null，但可为empty("",";")	一般用在基本数据类型的非空校验上，而且被其标注的字段可以使用 @size/@Max/@Min 对字段数值进行大小的控制	false	true	true	true
@NotEmpty	不能为null，而且长度必须大于0("",";")	一般用在集合类上面	false	false	true	true
@NotBlank	不能为null	这只能作用在接收的String类型上，注意是只能，而且调用trim()后，长度必须大于0	false	false	false	true

注意在使用@NotBlank等注解时，一定要和@valid一起使用，不然@NotBlank不起作用

```
public class User {


    public interface UserSimpleView {}

    public interface UserDetailsView extends UserSimpleView {}

    private String id;

    @MyConstraint(message = "这是一个测试")
    @ApiModelProperty(value = "用户名")
    private String username;

    @NotBlank(message = "密码不能为空")
    private String password;
}
```



```
@PostMapping
@ApiOperation(value = "创建用户")
public User create(@Valid @RequestBody User user) {
}
```



实际开发中，这三个东西一定要分的清楚，乱用或者没注意容易吃亏。一次开发中就是由于开发人员的疏忽，一个BigDecimal的字段使用成了@NotBlank（还是@NotEmpty来着，记不清了，不重要\_），然后导致服务器报错，后来将字段校验标签改成@NotNull后问题得到解决。

## @NotNull

这个注解可以用在成员方法或者构造方法的参数前面，会自动产生一个关于此参数的非空检查，如果参数为空，则抛出一个空指针异常，举个例子来看看：

//成员方法参数加上@NotNull注解

```
public String getName(@NotNull Person p) {
    return p.getName();
}
```

实际效果相当于：

```
public String getName(@NotNull Person p) {
    if(p==null){
        throw new NullPointerException("person");
    }
    return p.getName();
}
```

用在构造方法的参数上效果类似，就不再举例子了。

0

## @Override

在java中如果方法上加@Override的注解的话，表示子类重写了父类的方法。当然也可以不写，写的好处是：

可读性提高

编译器会校验写的方法在父类中是否存在

```
public class Father {  
    public void test() {  
        System.out.println("test");  
    }  
    class child extends Father {  
        @Override  
        public void test() {  
        }  
    }  
}
```

如果将test写成test1的话，编译器在父类中未找到此方法，将会报错

```
3 public class Father {  
4  
5     public void test(){  
6         System.out.println("test");  
7     }  
8  
9     class child extends Father {  
10  
11         @Override  
12         public void test1(){  
13  
14         }  
15     }  
16 }  
17  
18
```

如果将@Override注释去掉的话，那么编译器则会认为创建了新的方法

```
3 public class Father {  
4  
5     public void test(){  
6         System.out.println("test");  
7     }  
8  
9     class child extends Father {  
10  
11 //     @Override  
12     public void test1(){  
13  
14     }  
15 }  
16 }  
17  
18
```



P

### **@PathParam**

这个注解是和spring的pathVariable是一样的，也是基于模板的，但是这个是jboss包下面的一个实现，上面的是spring的一个实现，都要导包

### **@PathVariable**

@RequestParam 和 @PathVariable 注解是用于从request中接收请求的，两个都可以接收参数，关键点不同的是@RequestParam 是从request里面拿取值，而 @PathVariable 是从一个URI模板里面来填充。

这个注解能够识别URL里面的一个模板，我们看下面的一个URL

http://localhost:8080/springmvc/hello/101?param1=10&param2=20

上面的一个url你可以这样写：

```
@RequestMapping("/hello/{id}")
    public String getDetails(@PathVariable(value="id") String id,
        @RequestParam(value="param1", required=true) String param1,
        @RequestParam(value="param2", required=false) String param2) {
    .....
}
```

区别很明显了

### **@PreAuthorize**

spring security中可以通过表达式控制方法权限：

@PreAuthorize

@PostAuthorize

@PreFilter

@PostFilter

其中前两者可以用来在方法调用前或者调用后进行权限检查，后两者可以用来对集合类型的参数或者返回值进行过滤。@PreAuthorize可以用来控制一个方法是否能够被调用

Controller层

/\*\*

```

    * 根据用户编号获取详细信息
    */
    @PreAuthorize("@ss.hasPermi('system:user:query')")
    @GetMapping(value = { "/",("/{userId}" })
    public AjaxResult getInfo(@PathVariable(value = "userId", required = false) Long userId)
    {
        ...
    }
}

```

对应service层

```

@Service("ss")
public class PermissionService
{
    /**
    * 验证用户是否具备某权限
    *
    * @param permission 权限字符串
    * @return 用户是否具备某权限
    */
    public boolean hasPermi(String permission)
    {
    }
}

```

@PreAuthorize("@ss.hasPermi('system:user:query')")表示:

@ss标签对应的PermissionService

hasPermi方法

传入参数为system:user:query, 表示用户的查询权

@PostAuthorize目前并没有碰到;使用@PreFilter和@PostFilter进行过滤;使用@PreFilter和@PostFilter可以对集合类型的参数或返回值进行过滤。使用@PreFilter和@PostFilter时, Spring Security将移除使对应表达式的结果为false的元素。目前碰到的不多。

## @PreDestroy

Java EE5 引入了@PostConstruct和@PreDestroy这两个作用于Servlet生命周期的注解, 实现Bean初始化之前和销毁之前的自定义操作。

## 使用说明

PostConstruct 注释用于在依赖关系注入完成之后需要执行的方法上，以执行任何初始化。此方法必须在将类放入服务之前调用。支持依赖关系注入的所有类都必须支持此注释。即使类没有请求注入任何资源，用 PostConstruct 注释的方法也必须被调用。只有一个方法可以用此注释进行注释。

PreDestroy 用与在依赖注入完成之前的方法前面执行，

遵守准则：

该方法不得有任何参数

该方法的返回类型必须为 void；

该方法不得抛出已检查异常；

应用 PostConstruct 的方法可以是 public、protected、package private 或 private；

该方法不能是 static；该方法可以是 final；

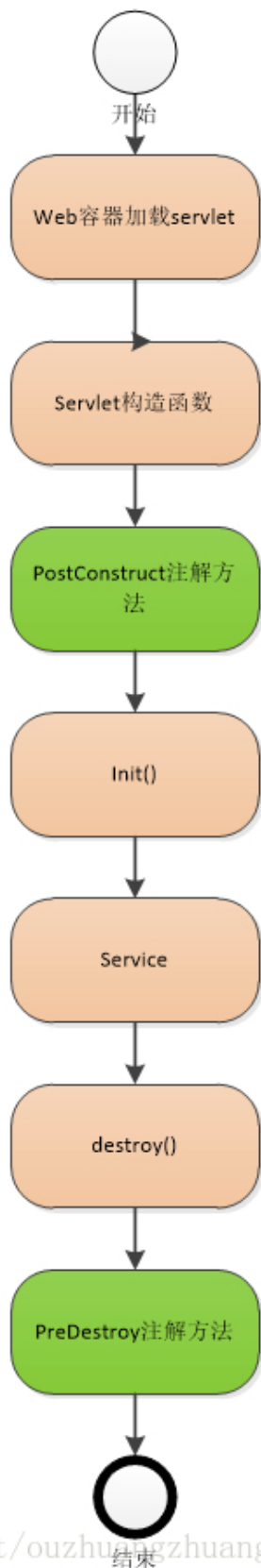
该方法只会被执行一次

例如：

```
@PreDestroy
public void delLock() {
    RedisShardedPoolUtil.del(Const.REDIS_LOCK.CLOSE_ORDER_TASK_LOCK);
}

@PostConstruct
private void init() {
    try{
        config.useSingleServer().setAddress(new
StringBuilder().append(redisIp).append(":").append(redisPort).toString());
        redisson = (Redisson) Redisson.create(config);
        log.info("初始化Redisson结束");
    }catch (Exception e){
        log.info("redisson init error",e);
    }
}
```

如果感觉还是很模糊，那么看下流程图估计你就好明白了，从网上找到的一张图，很详细：



R

@RabbitListener

RabbitMQ是目前非常热门的一款消息中间件，不管是互联网大厂还是中小企业都在大量使用。Spring Boot的兴起，极大地简化了Spring的开发。使用Spring Boot与RabbitMQ进行简单整合，实现生产和消费消息。

第一步配置。RabbitMQ实现了AMQP协议，引入依赖spring-boot-starter-amqp。

```
<!-- rabbitmq依赖 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

第二步要配置RabbitMQ连接信息，包括主机、端口号、用户名和密码。RabbitMQ配置信息：

```
spring.rabbitmq.host=192.168.16.128
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

如果没有安装RabbitMQ，我推荐使用Docker快速安装和启动，启动命令：

```
docker run -d --hostname my-rabbit -p 5672:5672 -p 15672:15672 rabbitmq:3.8.0-beta.4-
management
```

第三步实现生产和消费者。

生产者

生产者用来生产消息并进行发送。需要用到RabbitTemplate。RabbitTemplate是发送消息的关键类，convertAndSend方法以指定消息发送的交换器、路由键、消息内容等。

```
@Component
public class Producer {
    @Autowired
    RabbitTemplate rabbitTemplate;

    public void produce() {
        String message = new Date() + "Beijing";
        System.out.println("生产者生产消息=====" + message);
        rabbitTemplate.convertAndSend("rabbitmq_queue", message);
    }
}
```

消费者

消费者消费生产者发送的消息。实现消费者主要用到注解@RabbitListener。@RabbitListener是一个功能强大的注解。这个注解里面可以注解配置@QueueBinding、@Queue、@Exchange直接通过这个组合注解一次

性搞定多个交换机、绑定、路由、并且配置监听功能等。

在RabbitMQ控制面板创建好队列，使用@RabbitListener监听队列。

```
@RabbitListener(queues = "rabbitmq_queue")
```

使用@RabbitListener自动创建队列。

```
@RabbitListener(queuesToDeclare = @Queue("myQueue"))
```

使用@RabbitListener自动创建队列，并对Exchange和Queue进行绑定。

```
@RabbitListener(bindings = @QueueBinding(value = @Queue("myQueue"), key = "mobi", exchange  
= @Exchange("myExchange")))
```

使用@RabbitListener自动创建一个队列。

```
@Component  
public class Consumer {  
    @RabbitHandler  
    @RabbitListener(queuesToDeclare = @Queue("rabbitmq_queue"))  
    public void process(String message) {  
        System.out.println("消费者消费消息=====" + message);  
    }  
}
```

第四步测试。为了方便，写一个测试类生产消息。然后启动工程，运行测试类，使生产者发送消息，不出意外消费者将会消费消息，在控制台输出信息。

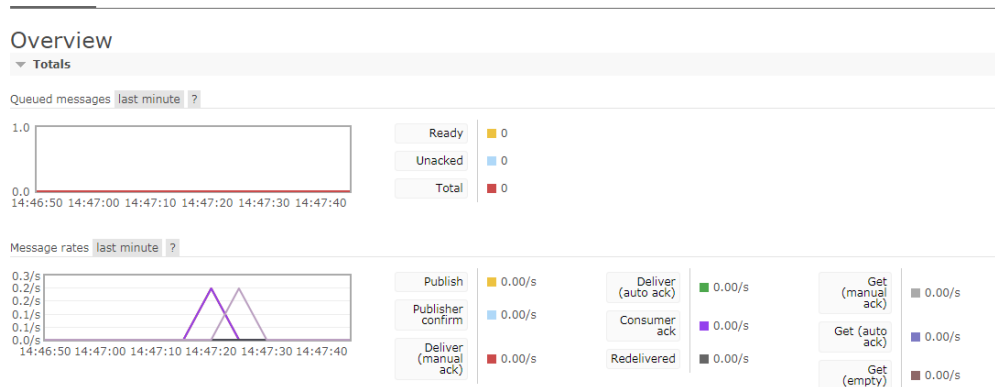
```
@RunWith(SpringRunner.class)
```

```
@SpringBootTest
```

```
public class ApplicationTests {  
    @Autowired  
    Producer producer;  
    @Test  
    public void contextLoads() {  
        producer.produce();  
    }  
}
```

操作之后可以在测试控制台看到生产者消息发送成功，在该工程的控制台看到消息消费成功。

访问RabbitMQ控制面板也会看到有消息。



## 总结

Spring Boot极大的简化各种组件的使用，在实际应用中，当一个服务处理完成之后可以将消息放进RabbitMQ消息队列，另外一个服务从队列中取消息进行消费，这样可以降低服务之间的耦合，实现一些异步的操作。

## @RepeatSubmit

表单重复提交在web应用中是比较常见的问题，重复提交的动作容易造成脏数据，为了避免这重复提交的操作简便的方便是采用拦截器+注解的方式。

基本的原理：url请求时，用拦截器拦截，生成一个唯一的标识符（token），在新建页面中Session保存token随机码，当保存时验证，通过后删除，当再次点击保存时由于服务器端的Session中已经不存在了，所有无法验证通过。

## 第一步：自定义注解

@Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME)

```
public @interface AvoidDuplicateSubmission {  
    boolean needSaveToken() default false;  
  
    boolean needRemoveToken() default false;  
  
    String tokenName() default "token";  
}
```

元注解的作用就是负责注解其他注解。Java5.0定义了4个标准的meta-annotation类型，它们被用来提供对其它 annotation类型作说明。Java5.0定义的元注解：

	表示该注解可以用于什么地方。可能的ElementType参数包括： CONSTRUCTOR：构造器的声明 FIELD：域声明（包括enum实例）
@Target	LOCAL_VARIABLE：局部变量声明 METHOD：方法声明 PACKAGE：包声明 PARAMETER：参数声明 TYPE：类、接口（包括注解类型）和enum声明
@Retention	表示需要在什么级别保存该注解信息。可选的RetentionPolicy参数包括： SOURCE：注解将在编译器丢弃 CLASS：注解在class文件中可用，但会被VM丢弃 RUNTIME：VM将在运行期也保留注解，因此可以通过反射机制读取注解的信息
@Documented	将此注解包含在Javadoc中
@Inherited	允许子类继承父类中的注解

## 元注解介绍

### @Target:

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）

取值(ElementType)有：

1. CONSTRUCTOR:用于描述构造器
2. FIELD:用于描述域
3. LOCAL\_VARIABLE:用于描述局部变量
4. METHOD:用于描述方法
5. PACKAGE:用于描述包
6. PARAMETER:用于描述参数
7. TYPE:用于描述类、接口(包括注解类型) 或enum声明

### @Retention:

作用：表示需要在什么级别保存该注释信息，用于描述注解的生命周期（即：被描述的注解在什么范围内有效）

取值(RetentionPoicy)有：

1. SOURCE:在源文件中有效（即源文件保留）
2. CLASS:在class文件中有效（即class保留）
3. RUNTIME:在运行时有效（即运行时保留）

### @Inherited:

@interface自定义注解时，自动继承了java.lang.annotation.Annotation接口，由编译程序自动完成其他细节。在定义注解时，不能继承其他的注解或接口。@interface用来声明一个注解，其中的每一个方法实际上是声明了一个配置参数。方法的名称就是参数的名称，返回值类型就是参数的类型（返回值类型只能是基本类型、Class、String、enum）。可以通过default来声明参数的默认值。

定义注解格式：

```
public @interface 注解名 {定义体}
```



Annotation类型里面的参数该怎么设定:

第一, 只能用public或默认(default)这两个访问权修饰. 例如, String value();这里把方法设为default默认类型;

第二, 参数成员只能用基本类型byte, short, char, int, long, float, double, boolean八种基本数据类型和 String, Enum, Class, annotations等数据类型, 以及这一些类型的数组. 例如, String value();这里的参数成员就为String;

第三, 如果只有一个参数成员, 最好把参数名称设为" value", 后加小括号. 例: 下面的例子FruitName注解就只有一个参数成员。

## 第二步: 实现拦截器

```
public class AvoidDuplicateSubmissionInterceptor extends HandlerInterceptorAdapter {

    private static final Logger logger =
        LogManager.getLogger(AvoidDuplicateSubmissionInterceptor.class);

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
        Object handler) throws Exception {

        HttpSession session = request.getSession(false);
        Object userId = session == null ? null : session.getAttribute("userId");
        HandlerMethod handlerMethod = (HandlerMethod) handler;
        Method method = handlerMethod.getMethod();

        AvoidDuplicateSubmission annotation =
            method.getAnnotation(AvoidDuplicateSubmission.class);
        if (annotation != null) {
            boolean needSaveSession = annotation.needSaveToken();
            String tokenName = annotation.tokenName();
            if (needSaveSession) {
                request.getSession().setAttribute(tokenName,
                    TokenProcessor.getInstance().makeToken(userId == null ? "" : userId.toString()));
            }

            boolean needRemoveSession = annotation.needRemoveToken();
            if (needRemoveSession) {
                if (isRepeatSubmit(request, tokenName)) {
```

```

        logger.warn("please don't repeat submit,[user:" + userId + ",url:" +
request.getServletPath() + "]");
        ResponseBody responseBody = method.getAnnotation(ResponseBody.class);
        if (responseBody == null) {
            response.sendRedirect("repeatsubmit.htm");
        }
        return false;
    }
    request.getSession(false).removeAttribute(tokenName);
}
}

return true;
}

private boolean isRepeatSubmit(HttpServletRequest request, String tokenName) {
    String serverToken = (String) request.getSession(false).getAttribute(tokenName);
    if (serverToken == null) {
        return true;
    }
    String clientToken = request.getParameter(tokenName);
    if (clientToken == null) {
        return true;
    }
    if (!serverToken.equals(clientToken)) {
        return true;
    }
    return false;
}
}

```

**第三步：** 在springBoot的配置类中加入拦截器

```

@Configuration
public class StartupConfig extends WebMvcConfigurerAdapter {
    private static final Logger logger = LogManager.getLogger(StartupConfig.class);

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new
AvoidDuplicateSubmissionInterceptor()).addPathPatterns("/*.htm");
    }
}

```

```

    }
}

```

第四步：在相应的controller方法加上注解

```

@AvoidDuplicateSubmission(needSaveToken = true)
@RequestMapping(value = "/insertXXX.htm")
@ResponseBody
public String insertXXX(Grade grade, HttpServletRequest req, HttpServletResponse resp) {
    JSONObject obj = new JSONObject();
    obj.put("success", true);
    try {

        String message = validateInsertForm(membershipGrade);
        if (StringUtils.isBlank(message)) {
            String userId = req.getSession().getAttribute("userId").toString();
            String userName = req.getSession().getAttribute("realName").toString();
            grade.setId(String.valueOf(Snowflake.getId()));
            grade.setCreateUser(userId);
            grade.setCreateUserName(userName);
            grade.setCreateTime(DateUtil.formatDatetime(new Date(), "yyyyMMddHHmmss"));
            gradeService.insert(membershipGrade);
            obj.put("message", "添加成功");
        } else {
            obj.put("success", false);
            obj.put("message", message);
        }
    } catch (Exception e) {
        obj.put("success", false);
        obj.put("message", "系统错误");
        logger.error(e.getMessage(), e);
    }
    return obj.toString();
}

```

最后在页面的form中加入下面的，就防止重复提交的问题

```

<input type="hidden" name="token" value="${token}">

```

下面是生成token的方法

```
public class TokenProcessor {
    private static final Logger logger = LogManager.getLogger(TokenProcessor.class);

    private TokenProcessor() {
    }

    private static final TokenProcessor instance = new TokenProcessor();

    public static TokenProcessor getInstance() {
        return instance;
    }

    public String makeToken(String userId) {
        String token = (System.currentTimeMillis() + new Random().nextInt(999999999)) +
userId;
        try {
            return DigestUtils.md5Hex(token);
        } catch (Exception e) {
            logger.error(e.getMessage(), e);
        }
        return System.currentTimeMillis() + "";
    }
}
```

### @RequestBody

@RequestBody这个一般处理的是content-type不是默认的application/x-www-form-urlencoded编码的内容，比如说：application/json、application/xml等。一般情况来说常用其来处理application/json类型，它是通过使用HandlerAdapter 配置的HttpMessageConverters来解析post data body，然后绑定到相应的bean上的，因为配置有FormHttpMessageConverter，所以也可以用来处理 application/x-www-form-urlencoded的内容，处理完的结果放在一个MultiValueMap<String, String>里，这种情况在某些特殊需求下使用，例如jQuery easyUI的datagrid请求数据的时候需要使用到这种方式、小型项目只创建一个POJO类的话也可以使用这种接受方式，详情查看FormHttpMessageConverter api。

通过@requestBody可以将请求体中的JSON字符串绑定到相应的bean上，当然，也可以将其分别绑定到对应的字符串上。

```
$.ajax({
    url: "/login",
```

```

        type: "POST",
        data: ' { "userName": "admin", "pwd", "admin123" }' ,
        content-type: "application/json charset=utf-8",
        success: function(data) {
            alert( "request success ! " );
        }
    });

```

```

@RequestMapping( "/login" )
public void login(@RequestBody String userName,@RequestBody String pwd){
    System.out.println(userName+" : "+pwd);
}

```

这种情况是将JSON字符串中的两个变量的值分别赋予了两个字符串. 假如我有一个User类, 拥有如下字段: String userName; String pwd; 那么上述参数可以改为以下形式: @RequestBody User user 这种形式会将JSON字符串中的值赋予user中对应的属性上, 需要注意的是, JSON字符串中的key必须对应user中的属性名, 否则是请求不过去的。

## @RequestMapping

### @RequestMapping 参数说明

value: 定义处理方法的请求的 URL 地址。(重点)

method: 定义处理方法的 http method 类型, 如 GET、POST 等。(重点)

params: 定义请求的 URL 中必须包含的参数。或者不包含某些参数。(了解)

headers: 定义请求中 Request Headers 必须包含的参数。或者不包含某些参数。(了解)

### @RequestMapping 的用法

@RequestMapping 有两种标注方式, 一种是标注在类级别上, 一种是标注在方法级别上。标注在方法上时, value 表示访问该方法的 URL 地址。标注在类上时, value 相当于一个命名空间, 即访问该 Controller 下的任意方法都需要带上这个命名空间。例如:

```

1 @Controller
2 @RequestMapping("/example")
3 public class ExampleController {
4
5     @RequestMapping
6     public String execute() {
7         return "example_page";
8     }
9 }

```

```

8      }
9
10     @RequestMapping("/todo")
11     public String doSomething() {
12         return "example_todo_page";
13     }
15 }

```

1: example.action: 执行的是 execute() 方法。execute() 方法的 @RequestMapping 注解缺省 value 值，在这种情况下，当访问命名空间时默认执行的是这个方法。方法级别上的 @RequestMapping 标注是必须的，否则方法无法被正确访问。

2: example/todo.action 执行的是 doSomething() 方法。类级别上的 @RequestMapping 标注不是必须的，在不写的情况下，方法上定义的 URL 都是绝对地址，否则，方法上定义的 URL 都是相对于它所在的 Controller 的。

#### **@RequestMapping(method):**

指定页面请求方式

```

1 @RequestMapping(value = "/register", method = RequestMethod.GET)
2 public String register() {
3     return "example_register_page";
4 }

```

method 的值一旦指定，那么，处理方法就只对指定的 http method 类型的请求进行处理。这里方法/register只能使用get请求，使用post请求无法访问

```

1 @RequestMapping(value = "/register", method = RequestMethod.GET)
2 public String register1() {
3     return "example_register_get_page";
4 }
5
6 @RequestMapping(value = "/register", method = RequestMethod.POST)
7 public String register2() {
8     return "example_register_post_page";
9 }

```

可以为多个方法映射相同的 URI，不同的 http method 类型，Spring MVC 根据请求的 method 类型是可以区分开这些方法的。当 /example/register.action 是以 GET 的方式提交的时候，Spring MVC 调用

register1() 来处理请求；若是以 POST 的方式提交，则调 register2() 来处理提交的请求。

method 若是缺省没指定，并不是说它默认只处理 GET 方式的请求，而是它可以处理任何方式的 http method 类型的请求。指定 method 是为了细化映射（缩小处理方法的映射范围），在 method 没有指定的情况下，它的映射范围是最大的。

#### @RequestMapping(params)

与 method 相类似，作用是为了细化映射。只有当 URL 中包含与 params 值相匹配的参数的请求，处理方法才会被调用。

```
1 @RequestMapping(value = "/find", params = "target")
2 public String find1() {
3     return "example_find1_page";
4 }
5
6 @RequestMapping(value = "/find", params = "!target")
7 public String find2() {
8     return "example_find2_page";
9 }
10
11 @RequestMapping(value = "/search", params = "target=product")
12 public String search1() {
13     return "example_search1_page";
14 }
15
16 @RequestMapping(value = "/search", params = "target!=product")
17 public String search2() {
18     return "example_search2_page";
19 }
```

find1(): 请求的 URL 中必须要有 target 参数，才能够到达此方法。如 /example/find.action?target 或 /example/find.action?target=x 等

find2(): 请求的 URL 中必须不能有 target 参数，才能够到达此方法。如 /example/find.action 或 /example/find.action?q=x 等

search1(): 请求的 URL 中必须要有 target=product 参数，才能够到达此方法。如 /example/search.action?target=product 等

search2(): 请求的 URL 中必须不能有 target=product 参数, 才能够到达此方法。如 /example/search.action?target=article 等

@RequestMapping(headers)

headers 的作用也是用于细化映射。只有当请求的 Request Headers 中包含与 headers 值相匹配的参数, 处理方法才会被调用。

```
1 @RequestMapping(value = "/specify", headers = "accept=text/*")
2 public String specify() {
3     return "example_specify_page";
4 }
```

请求的 Request Headers 中 Accept 的值必须匹配 text/\* ( 如 text/html ), 方法才会被调用。

支持Ant风格的通配符

通配符	说明	示例
?	匹配一个任意字符	/a/?b 可以匹配/a/ab;/a/cb。但不能匹配/a/acb之类
*	匹配任意长度的字符	/a/*b可以匹配/a/cb;/a/acb。但不能匹配/a/cb/vb
**	匹配多层路径	可以匹配/a/ab;/a/acb;/a/ab/abc/.../...

@RequestParam

1. 常用来处理简单类型的绑定, 通过Request.getParameter() 获取的String可直接转换为简单类型的情况 ( String -> 简单类型的转换操作由ConversionService配置的转换器来完成); 因为使用 request.getParameter() 方式获取参数, 所以可以处理get 方式中queryString的值, 也可以处理post方式中 body data的值;

2. 用来处理Content-Type: 为 application/x-www-form-urlencoded编码 (默认编码类型) 的内容, 提交方式可为GET、POST;

3. 该注解有两个属性: value、required; value用来指定要传入值的id名称, required用来指示参数是否必须绑定; 如下所示

http://localhost:8080/springmvc/hello/101?param1=10&param2=20

1

根据上面的这个URL, 你可以用这样的方式来进行获取

```
public String getDetails(
    @RequestParam(value="param1", required=true) String param1,
    @RequestParam(value="param2", required=false) String param2){
```



```
...  
}
```

**@RequestParam** 支持下面四种参数

defaultValue 如果本次请求没有携带这个参数，或者参数为空，那么就会启用默认值

name 绑定本次参数的名称，要跟URL上面的一样

required 这个参数是不是必须的

value 跟name一样的作用，是name属性的一个别名

**@Resource**

@Resource注解作用与@Autowired非常相似。先看一下@Resource，直接写School.java了：

```
public class School{  
    @Resource(name = "teacher")  
    private Teacher teacher;  
    @Resource(type = Student.class)  
    private Student student;  
    public String toString(){  
        return teacher + "\n" + student;  
    }  
}
```

这是详细一些的用法，说一下@Resource的装配顺序：

1. @Resource后面没有任何内容，默认通过name属性去匹配bean，找不到再按type去匹配。
2. 指定了name或者type则根据指定的类型去匹配bean。
3. 指定了name和type则根据指定的name和type去匹配bean，任何一个不匹配都会报错。

@Autowired和@Resource两个注解的区别：

@Autowired默认按照byType方式进行bean匹配，@Resource默认按照byName方式进行bean匹配 -

@Autowired是Spring的注解，@Resource是J2EE的注解，根据导入注解的包名就可以知道。

Spring属于第三方的，J2EE是Java自己的东西。因此，建议使用@Resource注解，以减少代码和Spring之间的耦合。

**@ResponseBody**

@RequestParam和@RequestBody均是处理request body部分的注解，都用于获取请求部分的参数。

@ResponseBody是用于响应部分的注解。@ResponseBody注解的作用是将controller的方法返回的对象通过适当的HttpMessageConverter转换器转换为指定的格式之后，写入到response对象的body区，通常用来返回JSON数据或者是XML数据，需要注意的呢，在使用此注解之后不会再走视图处理器，而是直接将数据写入到输入流中，他的效果等同于通过response对象输出指定格式的数据

```
@RequestMapping(“/login”)
```

```
    @ResponseBody
```

```
    public User login(User user) {
```

```
        return user;
```

```
    }
```

User字段: userName pwd

那么在前台接收到的数据为: ' { “userName” :” xxx” ,” pwd” :” xxx” } '

效果等同于如下代码:

```
@RequestMapping(“/login”)
```

```
public void login(User user, HttpServletResponse response) {
```

```
    response.getWriter().write(JSONObject.fromObject(user).toString());
```

```
}
```

@RestController

相当于@Controller+@ResponseBody两个注解的结合，返回json数据不需要在方法前面加@ResponseBody注解了，但使用@RestController这个注解，就不能返回jsp,html页面，视图解析器无法解析jsp,html页面

@CrossOrigin

```
@RestController /* @Controller + @ResponseBody*/
```

```
public class HospitalController {
```

```
    //注入Service服务对象
```

```
    @Autowired
```

```
    private HospitalService hospitalService;
```

```
    /**
```

```
    * 查询所有医院信息（未分页）
```

```
    */
```

```
@RequestMapping(value = “findAllHospital”,method = RequestMethod.GET)
```

```
public List<Hospital> findAllHospital() {
```

```
    List<Hospital> hospitalList= hospitalService.findAllHospital();
```

```
    return hospitalList;
```

```
}
```

## @RestControllerAdvice

在spring 3.2中, 新增了@ControllerAdvice, @RestControllerAdvice 注解, 可以用于定义 @ExceptionHandler、@InitBinder、@ModelAttribute, 并应用到所有@RequestMapping中。参考\*\*帮助文档。@RestControllerAdvice 是组件注解, 他使得其实现类能够被classpath扫描自动发现, 如果应用是通过MVC命令空间或MVC Java编程方式配置, 那么该特性默认是自动开启的。\*\*

主要配合@ExceptionHandler使用, 统一处理异常情况。下面的ResponseEntity、ResponseData 都是项目自定义的返回对象。

@Slf4j

@RestControllerAdvice

```
public class GlobalExceptionHandler {

    /**
     * 处理运行异常
     */
    @ExceptionHandler(RuntimeException.class)
    public ResponseEntity<String> handleRuntimeException(HttpServletRequest request,
        RuntimeException ex) {
        log.error("", ex);          log.error("请求地址: " + request.getRequestURL());
        log.error("请求参数: " + JSONUtil.toJsonStr(request.getParameterMap()));
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.BAD_REQUEST);
    }

    /**
     * 用来捕获404, 400这种无法到达controller的错误
     *
     * @param ex
     * @return
     * @throws Exception
     */
    @ExceptionHandler(value = Exception.class)
    public ResponseData defaultErrorHandler(Exception ex) throws Exception {
        log.error("", ex);
        ResponseData<Object> result = new ResponseData<Object>();
        result.setMessage(ex.getMessage());
        if (ex instanceof NoHandlerFoundException) {
            result.setCode("404");
        } else {
```

```

        result.setCode("500");
    }
    result.setData(null);
    result.setSuccess(false);
    return result;
}
}

```

补充：同时定义2套ExceptionHandler

**\*\*需求：\*\***原项目是根据一个SpringBoot开源项目改造来的，返回对象只满足前端使用，后来改成SpringCloud项目后，项目使用Feign相互调用时，再用原来那套返回对象就不好用了，只能接收到一个Http状态码，描述信息都收不到。因为之前已经有一个全局的GlobalExceptionHandler了，所以要在做一个套给Feign使用的ExceptionHandler，换个更适合的返回对象.代码如下：

```

@RestControllerAdvice(basePackageClasses = {com.ucap.zh.controller.ApiController.class})
@Order(Ordered.HIGHEST_PRECEDENCE)
public class ApiExceptionHandler {

}

```

@RestControllerAdvice注解使用了basePackageClasses指定了为Feign提供接口的ApiController类，说明此ExceptionHandler只作用到这个Controller，对其他无效。

@Order(Ordered.HIGHEST\_PRECEDENCE) 顺序注解，要提高此ExceptionHandler的执行顺序，必须在全局的GlobalExceptionHandler之前执行，如果此ExceptionHandler定义的异常未拦截成功，在走GlobalExceptionHandler

S

@Schedule

Cron表达式是一个字符串，字符串以5或6个空格隔开，分为6或7个域，每一个域代表一个含义，Cron有如下两种语法格式：

## cron表达式语法

1 | [秒] [分] [小时] [日] [月] [周] [年]

注: [年]不是必须的域, 可以省略[年], 则一共6个域

序号	说明	必填	允许填写的值	允许的通配符
1	秒	是	0-59	, - * /
2	分	是	0-59	, - * /
3	时	是	0-23	, - * /
4	日	是	1-31	, - * ? / L W
5	月	是	1-12 / JAN-DEC	, - * /
6	周	是	1-7 or SUN-SAT	, - * ? / L #
7	年	否	1970-2099	, - * /

### Cron语法

每一个域可出现的字符如下:

Seconds: 可出现", - \* / "四个字符, 有效范围为0-59的整数

Minutes: 可出现", - \* / "四个字符, 有效范围为0-59的整数

Hours: 可出现", - \* / "四个字符, 有效范围为0-23的整数

DayofMonth :可出现", - \* / ? L W C"八个字符, 有效范围为0-31的整数

Month: 可出现", - \* / "四个字符, 有效范围为1-12的整数或JAN-DEC

DayofWeek: 可出现", - \* / ? L C # "四个字符, 有效范围为1-7的整数或SUN-SAT两个范围。1表示星期天, 2表示星期一, 依次类推

Year: 可出现", - \* / "四个字符, 有效范围为1970-2099年

每一个域都使用数字, 但还可以出现如下特殊字符, 它们的含义是:

- (1) \*: 表示匹配该域的任意值, 假如在Minutes域使用\*, 即表示每分钟都会触发事件。
- (2) ?: 只能用在DayofMonth和DayofWeek两个域。它也匹配域的任意值, 但实际不会。因为DayofMonth和DayofWeek会相互影响。例如想在每月的20日触发调度, 不管20日到底是星期几, 则只能使用如下写法: 13 13 15 20 \* ?, 其中最后一位只能用?, 而不能使用\*, 如果使用\*表示不管星期几都会触发, 实际上并不是这样。
- (3) -: 表示范围, 例如在Minutes域使用5-20, 表示从5分到20分钟每分钟触发一次
- (4) /: 表示起始时间开始触发, 然后每隔固定时间触发一次, 例如在Minutes域使用5/20, 则意味着5分钟触发一次, 而25, 45等分别触发一次。
- (5) ,: 表示列出枚举值。例如: 在Minutes域使用5, 20, 则意味着在5和20分每分钟触发一次。
- (6) L: 表示最后, 只能出现在DayofWeek和DayofMonth域, 如果在DayofWeek域使用5L, 意味着在最后的一个星期四触发。

(7) W: 表示有效工作日(周一到周五), 只能出现在DayofMonth域, 系统将在离指定日期的最近的有效工作日触发事件。例如: 在 DayofMonth使用5W, 如果5日是星期六, 则将在最近的工作日: 星期五, 即4日触发。如果5日是星期天, 则在6日(周一)触发; 如果5日在星期一到星期五中的一天, 则就在5日触发。另外一点, W的最近寻找不会跨过月份。

(8) LW: 这两个字符可以连用, 表示在某个月最后一个工作日, 即最后一个星期五。

(9) #: 用于确定每个月第几个星期几, 只能出现在DayofMonth域。例如在4#2, 表示某月的第二个星期三。

举几个例子:

每隔5秒执行一次: `"*/5 * * * * ?"`

每隔1分钟执行一次: `"0 */1 * * * ?"`

每天23点执行一次: `"0 0 23 * * ?"`

每天凌晨1点执行一次: `"0 0 1 * * ?"`

每月1号凌晨1点执行一次: `"0 0 1 1 * ?"`

每月最后一天23点执行一次: `"0 0 23 L * ?"`

每周星期天凌晨1点实行一次: `"0 0 1 ? * L"`

在26分、29分、33分执行一次: `"0 26,29,33 * * * ?"`

每天的0点、13点、18点、21点都执行一次: `"0 0 0,13,18,21 * * ?"`

表示在每月的1日的凌晨2点调度任务: `"0 0 2 1 * ? *"`

表示周一到周五每天上午10:15执行作业: `"0 15 10 ? * MON-FRI"`

表示2002-2006年的每个月的最后一个星期五上午10:15执行: `"0 15 10 ? 6L 2002-2006"`

```
/**
```

```
*      定时任务
```

```
*                               作用: 同步数据库和redis
```

```
*/
```

```
@Scheduled(cron = "0 0/30 * * * ?")
```

```
    public void importItemDataToRedis() {
```

```
        System.err.println("同步开始—————"+new Date());
```

```
        //根据RedisKeyEnum.SEARCH_ALL_ITEMINFO_LIST.getKey()先删除redis中的数据
```

```
        redisUtils.del(RedisKeyEnum.SEARCH_ALL_ITEMINFO_LIST.getKey());
```

```
        List<MemberMoredayItemDetailsExtend> allItemList =
```

```
memberMoredayItemDetailsService.getAllItemList();
```

```
        //遍历
```

```
        for (MemberMoredayItemDetailsExtend itemInfo : allItemList) {
```

```
            redisUtils.hset(RedisKeyEnum.SEARCH_ALL_ITEMINFO_LIST.getKey(),
```

```
itemInfo.getItemId()+"", JSONObject.toJSONString(itemInfo));
```

```
        }
```

```
    }
```

```
@Service
```

使用@Service, 可以更加简化.xml文件配置. 因为spring的配置文件里面还有12行~14行三个bean, 应用spring配置文件里面一个自动扫描的标签, 可以把这三个bean也给去掉, 增强Java代码的内聚性并进一步减少配置文件。先看一下配置文件:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xmlns="http://www.springframework.org/schema/beans"
4     xmlns:context="http://www.springframework.org/schema/context"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6         http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
7         http://www.springframework.org/schema/context
8         http://www.springframework.org/schema/context/spring-context-4.2.xsd">
9
10     <context:component-scan base-package="com.zxt" />
11 </beans>
```

配置文件看起来特别清爽。School.java, Teacher.java和Student.java分别做如下修改:

@Service

```
public class School{
    @Autowired
    private Teacher teacher;
    @Autowired
    private Student student;
    public String toString(){
        return teacher + "\n" + student;
    }
}
```

@Service

```
public class Teacher{
    private String teacherName = "TW";
    public String toString() {
        return "TeacherName:" + teacherName;
    }
}
```

@Service

```
public class Student{
    private String studentName = "SL";
    public String toString() {
        return "StudentName:" + studentName;
    }
}
```

```
}  
}
```

这样，School.java在Spring容器中存在的形式就是“school”，即可以通过ApplicationContext的getBean(“school”)方法来得到School.java。

@Service注解，其实做了两件事情：

声明School.java是一个bean。这点很重要，因为School.java是一个bean，其他的类才可以使用@Autowired将School作为一个成员变量自动注入。

School.java在bean中的id是“school”，即类名且首字母小写。

### @Size

约束注解名称	约束注解说明
@Null	验证对象是否为空
@NotNull	验证对象是否为非空
@AssertTrue	验证 Boolean 对象是否为 true
@AssertFalse	验证 Boolean 对象是否为 false
@Min	验证 Number 和 String 对象是否大等于指定的值
@Max	验证 Number 和 String 对象是否小等于指定的值
@DecimalMin	验证 Number 和 String 对象是否大等于指定的值，小数存在精度
@DecimalMax	验证 Number 和 String 对象是否小等于指定的值，小数存在精度
@Size	验证对象（Array,Collection,Map,String）长度是否在给定的范围之内
@Digits	验证 Number 和 String 的构成是否合法
@Past	验证 Date 和 Calendar 对象是否在当前时间之前
@Future	验证 Date 和 Calendar 对象是否在当前时间之后
@Pattern	验证 String 对象是否符合正则表达式的规则

### ## @Slf4j

slf4j是一个日志标准，使用它可以完美的桥接到具体的日志框架，必要时可以简便的更换底层的日志框架，而不需要关心具体的日志框架的实现（slf4j-simple、logback等）。

slf4j提供了日志接口、获取具体日志对象的方法，常见用法：

```
private static final Logger logger = LoggerFactory.getLogger(LoggerTest.class);  
logger.debug("debug");
```



```
logger.info("info");
logger.error("error");
```

### @SpringBootApplication

SpringBoot程序启动入口一个是SpringApplication.run, 一个是@SpringBootApplication注解, 这个注解是由三部分组成:

1. @ComponentScan注解, 主要用于组件扫描和自动装配。
2. @SpringBootConfiguration注解, 这个注解主要是继承@Configuration注解, 主要用于加载配置文件。
3. @EnableAutoConfiguration注解, 这个注释启用了Spring Boot的自动配置功能, 可以自动为您配置很多东西。

@SpringBootApplication = @Configuration + @ComponentScan + @EnableAutoConfiguration, 是这三个Spring注释的组合, 只需一行代码即可提供所有三个注释的功能。

### 源码

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
    @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
    @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
    @AliasFor(annotation = EnableAutoConfiguration.class)
    Class[] exclude() default {};
    @AliasFor(annotation = EnableAutoConfiguration.class)
    String[] excludeName() default {};
    @AliasFor(annotation = ComponentScan.class, attribute = "basePackages")
    String[] scanBasePackages() default {};
    @AliasFor(annotation = ComponentScan.class, attribute = "basePackageClasses")
    Class[] scanBasePackageClasses() default {};
}
```

## 实例

```
@SpringBootApplication
public class HelloDemo implements CommandLineRunner {
    private static final Logger log = LoggerFactory.getLogger(HelloDemo.class);
    public static void main(String args[]) {
        SpringApplication.run(HelloDemo.class);
    }
    @Override
    public void run(String...args) throws Exception {
        log.info("hello 51gjie");
    }
}
```

## @SpringBootConfiguration

SpringBootConfiguration是SpringBoot项目的配置注解，这也是一个组合注解，

SpringBootConfiguration注解可以用java代码的形式实现spring中xml配置文件配置的效果，并会将当前类内声明的一个或多个以@Bean注解标记的方法的实例纳入到spring容器中，并且实例名就是方法名。

## 实例

## @SpringBootConfiguration

```
public class Config {
    @Bean
    public Map MyMap() {
        Map map = new HashMap();
        map.put("website", "51gjie");
        map.put("type", "javaschool");
        map.put("age", 5);
        return map;
    }
}

@RestController
@SpringBootApplication
public class App
{
    public static void main( String[] args )
    {
        Map map = (Map) context.getBean("MyMap");    //注意这里直接获取到这个方法bean
        int age = (int) map.get("age");
    }
}
```

```
        System.out.println("age==" + age);
    }
}
```

SpringBoot项目中推荐使用@SpringBootConfiguration替代@Configuration。

T

**@TableField**

该注解用于标识非主键的字段。将数据库列与 JavaBean 中的属性进行映射，例如：

```
@TableName(value = "user")
public class AnnotationUser4Bean {
    @TableId(value = "user_id", type = IdType.AUTO)
    private String userId;
    @TableField("name")
    private String name;
    @TableField("sex")
    private String sex;
    @TableField("age")
    private Integer age;
    @TableLogic(value = "0", delval = "1")
    private String deleted;
}
```

属性名称	类型	默认值	描述
value	String	""	数据库字段名
el	String	""	映射为原生 <code>#{ ... }</code> 逻辑,相当于写在 xml 里的 <code>#{ ... }</code> 部分
exist	boolean	true	是否为数据库表字段
condition	String	""	字段 <code>where</code> 实体查询比较条件,有值设置则按设置的值为准,没有则为默认全局的 <code>%s=#{%s}</code> ,
update	String	""	字段 <code>update set</code> 部分注入,例如: <code>update="%s+1"</code> : 表示更新时会set <code>version=version+1</code> (该属性优先级高于 <code>el</code> 属性)
insertStrategy	Enum	DEFAULT	见官网
updateStrategy	Enum	DEFAULT	见官网
whereStrategy	Enum	DEFAULT	见官网
fill	Enum	FieldFill.DEFAULT	字段自动填充策略
select	boolean	true	是否进行 select 查询
keepGlobalFormat	boolean	false	是否保持使用全局的 format 进行处理
jdbcType	JdbcType	JdbcType.UNDEFINED	JDBC类型 (该默认值不代表会按照该值生效)
typeHandler	Class<? extends TypeHandler>	UnknownTypeHandler.class	类型处理器 (该默认值不代表会按照该值生效)
numericScale	String	""	指定小数点后保留的位数

@TableId

描述：主键注解

属性	类型	必须指定	默认值	描述
value	String	否	""	主键字段名
type	Enum	否	IdType.NONE	主键类型

idType

值	描述
AUTO	数据库ID自增
NONE	无状态,该类型为未设置主键类型(注解里等于跟随全局,全局里约等于 INPUT)
INPUT	insert前自行set主键值
ASSIGN_ID	分配ID(主键类型为Number(Long和Integer)或String)(since 3.3.0),使用接口 <code>IdentifierGenerator</code> 的方法 <code>nextId</code> (默认实现类为 <code>DefaultIdentifierGenerator</code> 雪花算法)
ASSIGN_UUID	分配UUID,主键类型为String(since 3.3.0),使用接口 <code>IdentifierGenerator</code> 的方法 <code>nextUUID</code> (默认default方法)
ID_WORKER	分布式全局唯一ID 长整型类型(please use <code>ASSIGN_ID</code> )
UUID	32位UUID字符串(please use <code>ASSIGN_UUID</code> )
ID_WORKER_STR	分布式全局唯一ID 字符串类型(please use <code>ASSIGN_ID</code> )

@TableLogic

效果：在属性字段上加@TableLogic注解，使用MyBatis-Plus自带方法删除（在执行BaseMapper的删除方法时，删除方法会变成修改）和查找都会附带逻辑删除功能（自己写的xml不会）。

@TableLogic(value= “原值”,delval= “改值” ).

逻辑删除是为了方便数据恢复和保护数据本身价值等等的一种方案，但实际就是删除。如果你需要再查出来就不应使用逻辑删除，而是以一个状态去表示。如：员工离职，账号被锁定等都应该是一个状态字段，此种场景不应使用逻辑删除。若确需查找删除数据，如老板需要查看历史所有数据的统计汇总信息，请单独手写sql。

@TableName

描述：表名注解

属性	类型	默认值	描述
value	String	""	表名
schema	String	""	schema
keepGlobalPrefix	boolean	false	是否保持使用全局的 tablePrefix 的值(如果设置了全局 tablePrefix 且自行设置了 value 的值)
resultMap	String	""	xml 中 resultMap 的 id
autoResultMap	boolean	false	是否自动构建 resultMap 并使用(如果设置 resultMap 则不会进行 resultMap 的自动构建并

V

@Validated

@Valid是使用hibernate validation的时候使用；而@Validated 是只用spring Validator 校验机制使用。

b/s系统中对http请求数据的校验多数在客户端进行，这也是出于简单及用户体验性上考虑，但是在一些安全性要求高的系统中服务端校验是不可缺少的。

Spring3支持JSR-303验证框架，JSR-303 是Java EE 6 中的一项子规范，叫做BeanValidation，官方参考实现是hibernate Validator（与Hibernate ORM 没有关系），JSR 303 用于对Java Bean 中的字段的值进行验证。

引入依赖

```
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>2.0.1.Final</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.16.Final</version>
</dependency>
```

## 1. bean 中添加标签

标签需要加在属性上，@NotBlank 标签含义文章末尾有解释

```
public class User {
    private Integer id;
    @NotBlank(message = "{user.name.notBlank}")
    private String name;
    private String username;
}
```

## 2. Controller中开启验证

在Controller 中 请求参数上添加@Validated 标签开启验证

```
@RequestMapping(method = RequestMethod.POST)
public User create(@RequestBody @Validated User user) {
    return userService.create(user);
}
```

```
@RequestMapping(method = RequestMethod.GET)
public User getUserById(@NotNull(message = "id不能为空") int userId) {
```

```

        return userService.getUserById(userId);
    }
}

```

### 3. resource 下新建错误信息配置文件

当然 message 信息也可以配置在标签后面例如

```

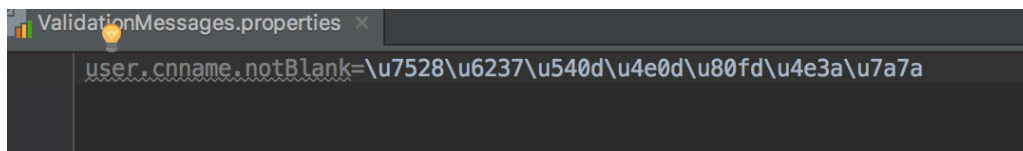
public class User {
    private Integer id;
    @NotBlank(message = "名字不能为空")
    private String name;
    private String username;
}

```

也可以在resource 目录下新建提示信息配置文件 “ValidationMessages.properties “ 这样可以全局统一管理错误消息

注意：名字必须为 “ValidationMessages.properties “ 因为SpringBoot自动读取classpath中的 ValidationMessages.properties里的错误信息

ValidationMessages.properties 文件的编码为ASCII。数据类型为 key value 。  
key “user.name.notBlank “为第一步 bean的标签 大括号里面对应message的值  
value 为提示信息 ， 但是是ASCII 。（内容为 “名字不能为空 “）



### 4. 自定义异常处理器，捕获错误信息

当验证不通过时会抛异常出来，异常的message 就是 ValidationMessages.properties 中配置的提示信息。此处定义异常处理器。捕获异常信息（因为验证不通过的项可能是多个所以统一捕获处理），并抛给前端。（此处是前后端分离开发）

```

@ControllerAdvice(MethodArgumentNotValidException.class)
public class MethodArgumentNotValidExceptionHandler {
    private static final Logger logger = LoggerFactory.getLogger(MethodArgumentNotValidExceptionHandler.class);

    public void handleMethodArgumentNotValidException(MethodArgumentNotValidException ex, HttpServletRequest request,
        HttpServletResponse response) {
        logger.error(":" + CommonUtil.getHttpClientInfo(request), ex);
        MethodArgumentNotValidException c = (MethodArgumentNotValidException) ex;
        List<ObjectError> errors = c.getBindingResult().getAllErrors();
        StringBuffer errorMsg = new StringBuffer();
        errors.stream().forEach(x -> errorMsg.append(x.getDefaultMessage()).append(";"));
    }
}

```

```

        populateExceptionResponse(response, HttpStatus.INTERNAL_SERVER_ERROR,
errorMsg.toString());
    }

```

```

private void populateExceptionResponse(HttpServletResponse response, HttpStatus errorCode,
String errorMessage) {
    try {
        response.sendError(errorCode.value(), errorMessage);
    } catch (IOException e) {
        logger.error("failed to populate response error", e);
    }
}

```

## 5. 附上部分标签含义

限制	说明
@Null	限制只能为null
@NotNull	限制必须不为null
@AssertFalse	限制必须为false
@AssertTrue	限制必须为true
@DecimalMax(value)	限制必须为一个不大于指定值的数字
@DecimalMin(value)	限制必须为一个不小于指定值的数字
@Digits(integer,fraction)	限制必须为一个小数，且整数部分的位数不能超过integer，小数部分的位数不能超过fraction
@Future	限制必须是一个将来的日期
@Max(value)	限制必须为一个不大于指定值的数字
@Min(value)	限制必须为一个不小于指定值的数字
@Past	限制必须是一个过去的日期
@Pattern(value)	限制必须符合指定的正则表达式
@Size(max,min)	限制字符长度必须在min到max之间
@Past	验证注解的元素值（日期类型）比当前时间早
@NotEmpty	验证注解的元素值不为null且不为空（字符串长度不为0、集合大小不为0）
@NotBlank	验证注解的元素值不为空（不为null、去除首位空格后长度为0），不同于@NotEmpty，@NotBlank只应用于字符串且在比较时会去除字符串的空格
@Email	验证注解的元素值是Email，也可以通过正则表达式和flag指定自定义的email格式



示例

```
@Pattern(regexp="^[a-zA-Z0-9]+$", message="{account.username.space}")
@Size(min=3, max=20, message="{account.username.size}")
}
```

如果上述的参数校验不满足要求可以 考虑自定义注解

自定义注解校验

步骤：1、定义注解，2、实现校验逻辑。用法

```
public class MySaveArgs {
    @NotEmpty
    @MustBeMyCode
    private String code;
}
```

定义注解

```
@Constraint(
    validatedBy = {MyCodeConstraintValidator.class}
)
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface MustBeMyCode {

    String message() default "编码校验不通过";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}
```

实现ConstraintValidator 接口，编写自己的校验逻辑，

```
public class MyCodeConstraintValidator implements ConstraintValidator<MustBeMyCode, String>
{

    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
```

```

        //此处编写自己的校验逻辑，并返回
        return value != null;
    }
}

```

注意：ConstraintValidator<MustBeMyCode, String> 此处应填写你自己的校验注解名 和 需校验参数类型

## @Value

Spring开发过程中经常遇到需要把特殊的值注入到成员变量里，比如普通值、文件、网址、配置信息、系统 变量等等。Spring主要使用注解@Value把对应的值注入到变量中。

常用的注入类型有以下几种：

1. 注入普通字符串。
2. 注入操作系统属性。
3. 注入表达式运算结果。
4. 注入其他bean的属性。
5. 注入文件内容。
6. 注入网址信息。
7. 注入属性文件。

准备. 由于例子需要读取文件和网页内容，为了方便读取，我们引入一个IO包：

```

<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.4</version>
</dependency>12345

```

在resources下面建立一个文件夹，名称为ch2.value。

在文件夹下面建立一个test.text，内容随意，我们的内容是”测试文件”。

在文件夹下面再建立一个test.properties，内容为：

```

book.author = feige
book.name = spring12
1
2
测试bean

```

新建一个用来测试的类，声明成一个bean。

```
@Service
public class DemoService {
    @Value("我是其他属性")
    private String anotherValue;

    public String getAnotherValue() {
        return anotherValue;
    }

    public void setAnotherValue(String anotherValue) {
        this.anotherValue = anotherValue;
    }
}
```

## 配置类

```
@Configuration
@ComponentScan("ch2.value")
@PropertySource("classpath:ch2/value/test.properties")
public class Config {
    @Value("我是个普通字符串")
    private String normal;

    @Value("#{systemEnvironment['os.name']}")
    private String osName;

    @Value("#{T(java.lang.Math).random()*1000.0}")
    private double randomNumber;

    @Value("#{demoService.anotherValue}")
    private String anotherValue;

    @Value("classpath:ch2/value/test.txt")
    private Resource testFile;

    @Value("http://www.baidu.com")
    private Resource testUrl;

    @Value("${book.name}")
```

```
private String bookName;
```

```
@Autowired
```

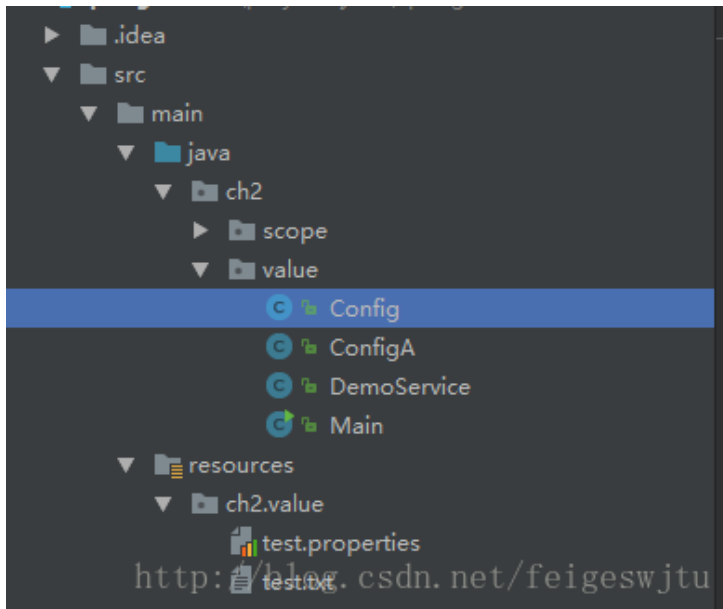
```
private Environment environment;
```

```
public void outSource() {  
    System.out.println(normal);  
    System.out.println(osName);  
    System.out.println(randomNumber);  
    System.out.println(anotherValue);  
    try {  
        System.out.println(IOUtils.toString(testFile.getInputStream()));  
        System.out.println(IOUtils.toString(testUrl.getInputStream()));  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
    System.out.println(bookName);  
    System.out.println(environment.getProperty("book.author"));  
}
```

## 运行示例

```
public class Main {  
    public static void main(String []args) {  
        AnnotationConfigApplicationContext context = new  
AnnotationConfigApplicationContext(Config.class);  
        Config config = context.getBean(Config.class);  
        config.outSource();  
    }  
}
```

## 目录结构



运行结果

我是个普通字符串

null

47.47599424058235

我是其他属性

测试文件

spring

```
14:11:10.719 [main] DEBUG org.springframework.core.env.PropertySourcesPropertyResolver -  
Found key 'book.author' in [class path resource [ch2/value/test.properties]] with type  
[String]
```

feige

@Value

普通字符串

```
@Value("我是个普通字符串")  
private String normal;12
```

操作系统属性

```
@Value("#{systemEnvironment['os.name']}")  
private String osName;12
```

操作系统的属性是静态全局变量systemEnvironment存入，可通过它获取到操作系统的属性。

表达式值

```
@Value("#{T(java.lang.Math).random()*1000.0}")  
private double randomNumber;12
```

表达式的对象必须是通过T()包起来，才能执行。

其他Bean的属性

```
@Value("#{demoService.anotherValue}")  
private String anotherValue;12
```

demoService是一个Bean对象，anotherValue是它的一个属性，可以通过@Value("#{demoService.anotherValue}")将这个bean的属性注入@Value声明的属性里。

注入文件资源

```
java  
@Value("classpath:ch2/value/test.txt")  
private Resource testFile;12
```

通过Resource接收这个文件。

注入网页资源

```
``` java  
@Value("http://www.baidu.com")  
private Resource testUrl;12
```

通过Resource接收这个资源。

注入配置属性

```
@Value("${book.name}")  
private String bookName;12
```

通过\${}注入配置属性，注意不是#号，这个是和其他的不一样，另外在Spring 4中需要用property-placeholder标签把当前要注入的配置注册一下才可以使用