# @see

## 基本概念

@see 注解是 JavaDoc 中的一种标签，用于提供与当前文档相关的其他类、方法或字段的引用。这种引用可以帮助开发者更好地理解和使用代码，提供更多的上下文信息。@see 标签通常用在类、方法或字段的文档注释中。它允许你链接到其他 JavaDoc 文档，以便读者能够快速找到相关的信息。

## 语法



你可以使用以下几种方式来引用其他元素：

直接引用类名：@see ExampleClass

引用方法：@see ExampleClass#exampleMethod()

引用字段：@see ExampleClass#exampleField

外部链接：@see <a href="https://example.com">Example Link</a>

## 示例

下面是一个简单的示例，展示了如何在 JavaDoc 中使用 @see 标签：



## 作用

增强可读性：通过添加相关信息，@see 可以使文档更加清晰，帮助开发者理解类或方法的关系。

提高发现性：读者可以通过 @see 快速找到相关的功能或类，减少在文档中查找的时间。

创建文档链接：生成的 JavaDoc 会自动创建超链接，使得浏览文档时更加方便。

## 生成文档

使用 JavaDoc 工具生成文档时，@see 标签会被转换为链接，指向相关的类、方法或字段。这使得用户在阅读文档时可以轻松导航到其他相关部分。

## 总结

@see 是一个非常有用的 JavaDoc 标签，可以帮助开发者在文档中提供额外的上下文信息。通过合理使用 @see，可以显著提升代码的可维护性和可读性，为其他开发者提供更好的帮助。

# @Entity

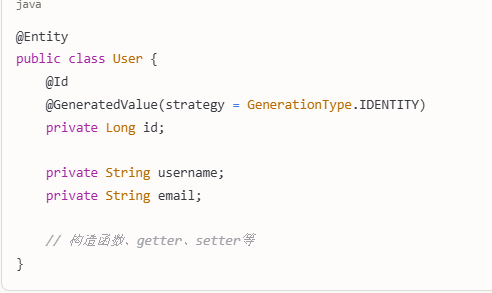
## 基本概念

@Entity注解是JPA（Java Persistence API）中最基础和重要的注解之一，用于标识一个Java类为实体类，使其能够映射到数据库表中。

## 作用

@Entity注解将一个普通的Java类标记为JPA实体，告诉JPA提供者（如Hibernate）这个类需要被持久化到数据库中。被标注的类会自动映射为数据库中的一张表。

## 用法



## 主要特性和要求

### 1、必须条件

（1）类必须有一个无参构造函数（可以是私有的）

（2）类不能是final类型

（3）实体必须有一个主键，使用@Id注解标识

（4）持久化字段不能是final类型

### 2、表名映射

默认情况下，实体类名就是对应的表名。可以通过name属性自定义表名：



### 3、字段映射规则

（1）类的属性默认映射为表的列

（2）属性名默认就是列名

（3）可以使用@Column注解自定义列的属性



## 总结

@Entity注解是JPA ORM框架的核心，它简化了Java对象与数据库表之间的映射关系，使开发者能够以面向对象的方式操作数据库，大大提高了开发效率。

# @Serial

在Java 14中引入的，它的作用是为了控制序列化过程，类似于Java 5中引入的@Deprecated注解和Java 8中引入的@FunctionalInterface注解。具体来说，使用@Serial注解可以为一个类或者接口显式地声明一个默认的序列化方式，以便支持序列化和反序列化。通常情况下，Java对象的序列化方式是由JVM自动生成的，而我们通过@Serial注解可以强制指定一个特定的序列化方式。

# @CallerSensitive

## 基本概念

主要用于指示某个方法是"调用者敏感"的。它的使用场景一般与安全性和访问控制相关。调用者敏感：指的是某个方法的行为可能依赖于调用该方法的类或方法的信息。这通常涉及到访问权限或安全检查。

## 使用场景

通常用于那些在执行时需要依据调用者的上下文（例如其权限或安全级别）来改变行为的方法。这种设计可以让某些方法根据具体的调用者进行不同的操作。

# @Primary

## 作用域

使用在类、方法上

## 作用

@Primary注解用于标记**当存在多个相同类型的Bean时，优先选择哪一个进行注入**。当Spring容器中有多个候选Bean可以满足依赖注入需求时，@Primary标记的Bean会被优先选择。

## 解决的问题

当你有多个实现同一接口的Bean时，Spring不知道该注入哪一个，会抛出NoUniqueBeanDefinitionException异常。@Primary就是用来解决这个歧义性的。

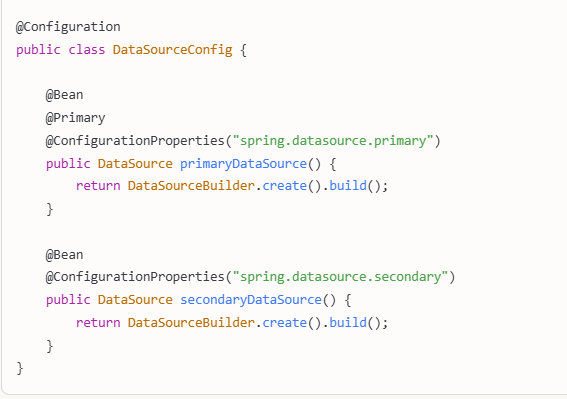
## 使用示例





## 应用场景

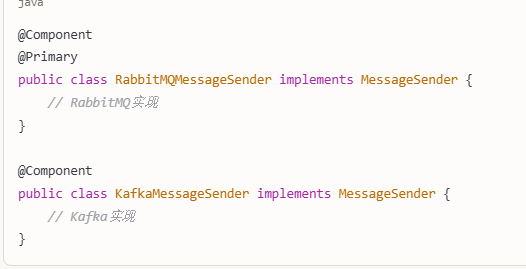
### 1、数据源配置



### 2、缓存策略选择



### 3、消息队列实现



## 与其他注解配合使用

### 1、与@Qualifier结合



### 2、与@ConditionalOnProperty结合

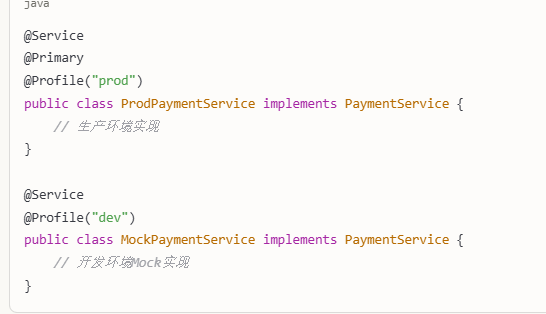


## 配置类中使用

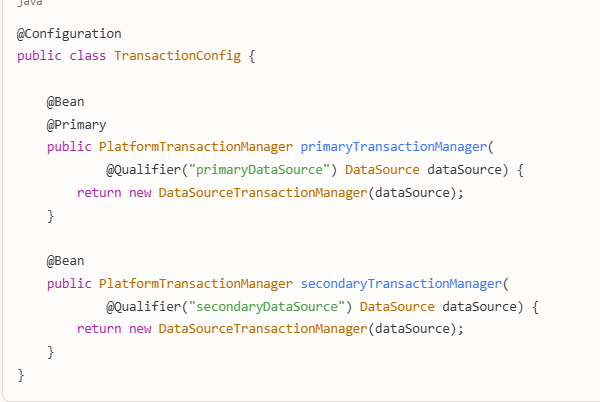


## 实际项目最佳实践

### 1、环境隔离



### 2、多数据源事务管理



## 注意事项

1、优先级关系：

（1）@Qualifier > @Primary > 按名称匹配

（2）如果同时使用@Qualifier，@Primary会被忽略

2、避免滥用

（1）不要在所有Bean上都加@Primary

（2）考虑使用@Qualifier或配置化方式替代

（3）保持代码的可读性和可维护性

## 总结

@Primary注解是Spring依赖注入中解决Bean选择歧义的重要工具，合理使用可以让代码更加清晰和灵活。

# @Autowired

## 基本概念

对成员变量、方法和构造函数进行标注来完成自动注入，默认按类型注入。Spring 容器中匹配的候选 Bean 数目必须有且仅有一个。当找不到一个匹配的 Bean 时，Spring 容器将抛出BeanCreationException 异常，并指出必须至少拥有一个匹配的 Bean。

## 属性

### required

默认值为true。

# @Qualifier

## 基本概念

可以对成员变量、方法以及构造函数进行注释，而 @Qualifier 的标注对象是成员变量、方法入参、构造函数入参。如果写在set方法上，@Qualifier需要写在方法的参数上。@Qualifier("XXX") 中的 XX是 Bean 的名称，所以 @Autowired 和 @Qualifier 结合使用时，自动注入的策略就从 byType 转变byName了。@Qualifier 只能和 @Autowired 结合使用，是对 @Autowired 有益的补充。一般来讲，@Qualifier 对方法签名中入参进行注释会降低代码的可读性，而对成员变量注释则相对好一些。

## 示例

@Autowired

@Qualifier("loginServiceImpl")

private LoginService loginService;

# @Resource

@Resource 的作用相当于 @Autowired，只不过 @Autowired 按 byType 自动注入，而@Resource 默认按 byName 自动注入罢了。不足：如果没有源码，就无法使用注解，只能使用xml文件。 要让resource注解生效，除了在 Bean 类中标注这些注释外，还需要在 Spring 容器中注册一个负责处理这些注释的 BeanPostProcessor：

|  |
| --- |
| <bean  class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor"/> |

CommonAnnotationBeanPostProcessor 实现了 BeanPostProcessor 接口，它负责扫描使用了 JSR-250 注释的 Bean，并对它们进行相应的操作。

# @Autowired和@Resource的区别

## 1、包含的属性不同

## 2、@Autowired默认按byType自动装配，而@Resource默认byName自动装配。

@Autowired如果要使用byName，需要使用@Qualifier一起配合。而@Resource如果指定了name，则用byName自动装配，如果指定了type，则用byType自动装配。

## 3、注解应用的地方不同

@Autowired能够用在：构造器、方法、参数、成员变量和注解上，而@Resource能用在：类、成员变量和方法上。

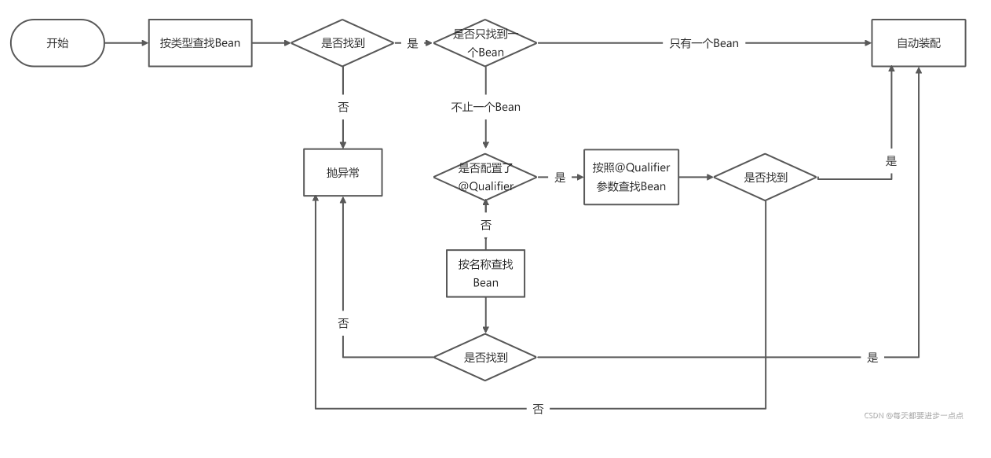
## 4、出处不同

@Autowired是Spring定义的注解，而@Resource是JSR-250定义的注解。所以@Autowired只能在Spring框架下使用，而@Resource则可以与其他框架一起使用。

## 5、装配顺序不同

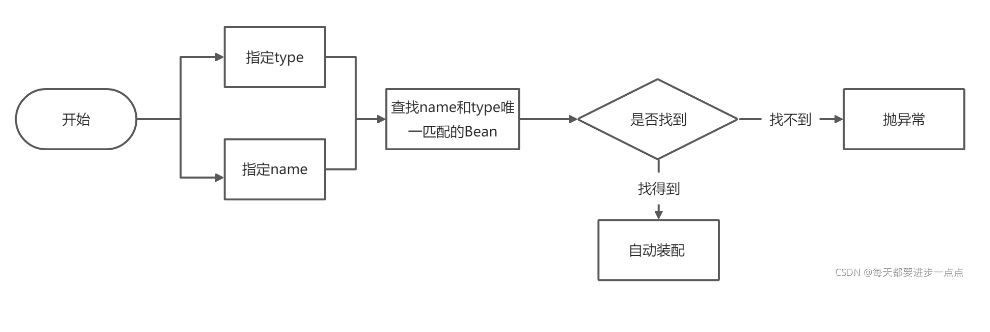
### @Autowired的装配顺序如下

@Autowired默认先按byType进行匹配，如果发现找到多个bean，则又按照byName方式进行匹配，如果还有多个，则报出异常。



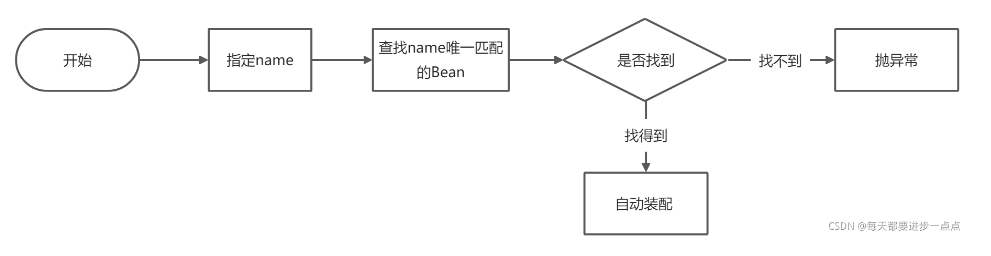
### @Resource的装配顺序如下

如果同时指定了name和type，流程如下：



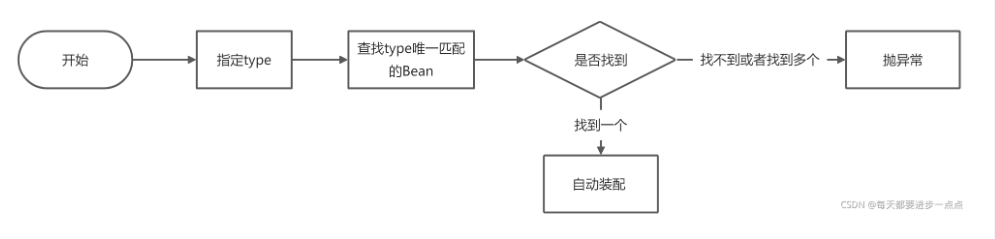
如果指定了name，流程如下：

只是指定了@Resource注解的name，则按name后的名字去bean元素里查找有与之相等的name属性的bean。



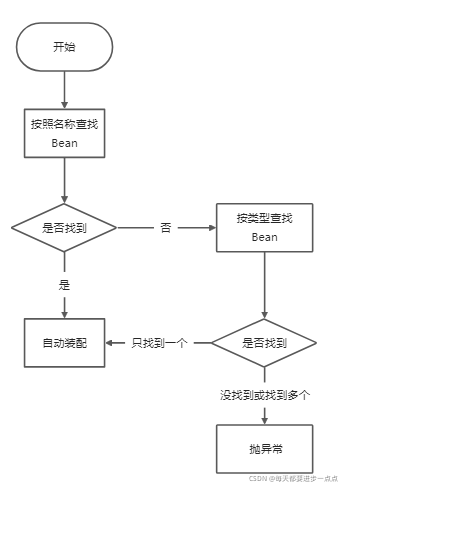
如果指定了type，流程如下：

只指定@Resource注解的type属性，则从上下文中找到类型匹配的唯一bean进行装配，找不到或者找到多个，都会抛出异常。



如果既没有指定name，也没有指定type，流程如下：

既不指定name属性，也不指定type属性，则自动按byName方式进行查找。如果没有找到符合的bean，则回退为一个原始类型进行进行查找，如果找到就注入。



# @RequestBody

## 作用

（1）用于读取request请求的body部分数据，使用系统默认配置的HttpMessageConverter进行解析，然后把响应的数据绑定到要返回的对象上。

（2）再把HttpMessageConverter返回的对象数据绑定到 controller中方法的参数上。

## 使用时机

### （1）GET、POST方式

根据request header Content-Type的值来判断:

application/x-www-form-urlencoded， 可选（即非必须，因为这种情况的数据@RequestParam, @ModelAttribute也可以处理，当然@RequestBody也能处理）

multipart/form-data, 不能处理（即使用@RequestBody不能处理这种格式的数据）

其他格式， 必须（其他格式包括application/json, application/xml等，这些格式的数据，必须使用@RequestBody来处理）

### （2）PUT方式提交时

根据request header Content-Type的值来判断:

application/x-www-form-urlencoded， 必须；

multipart/form-data, 不能处理；

其他格式， 必须；

## form-data与x-www-form-urlencoded的区别

### 基本概念

在 HTTP 请求中，form-data 和 x-www-form-urlencoded 是两种常见的 Content-Type，用于在 POST 请求中传递数据。

### application/x-www-form-urlencoded

这是最常见的表单提交方式，数据会被编码为键值对，格式类似于 URL 查询参数。

#### 特点

1、编码方式：数据会被编码为 key1=value1&key2=value2 的形式。

2、适用场景：适合传递简单的文本数据（如字符串、数字等）。

3、数据格式：所有数据都会被 URL 编码（例如空格会被转换为 %20）。

4、文件上传：不支持文件上传。

### multipart/form-data

这种方式通常用于上传文件或二进制数据，数据会被分成多个部分（part），每个部分都有自己的 Content-Type。

#### 特点

1、编码方式：数据会被分成多个部分，每个部分用分隔符（boundary）分隔。

2、适用场景：适合传递文件、二进制数据或混合类型数据（如文本 + 文件）。

3、数据格式：每个部分可以有自己的 Content-Type，例如 text/plain 或 image/png。

4、文件上传：支持文件上传。

### 区别对比



## 说明

request的body部分的数据编码格式由header部分的Content-Type指定。@RequestBody需要把所有请求参数作为json解析，因此不能包含key=value这样的写法在请求url中，所有的请求参数都是一个json，直接通过浏览器输入url时，@RequestBody获取不到json对象，需要用java编程或者基于ajax的方法请求，将Content-Type设置为application/json。

# @ResponseBody

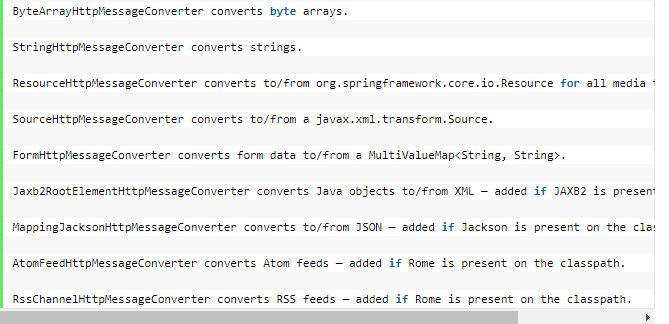
## 作用

  该注解用于将Controller类的方法返回的对象，通过适当的HttpMessageConverter转换为指定格式后，写入到Response对象的body数据区。

## 使用时机

  返回的数据不是html标签的页面，而是其他某种格式的数据时（如json、xml等）使用。HttpMessageConverter这个接口定义了四个方法：分别是读取数据时的 canRead(), read()和写入数据时的canWrite(), write()方法。

在使用 <mvc:annotation-driven />标签配置时，默认配置了RequestMappingHandlerAdapter（注意是RequestMappingHandlerAdapter不是AnnotationMethodHandlerAdapter。



当使用@RequestBody和@ResponseBody注解时，RequestMappingHandlerAdapter就使用它们来进行读取或者写入相应格式的数据。

## 作用范围

这个注解用于方法上。

# @ResponseStatus

## 基本概念

用于指定控制器方法或异常类在发生特定HTTP请求错误时返回的HTTP响应状态码和原因。它可以被用于标记控制器方法或自定义异常类，并根据具体情况返回不同的HTTP状态码。

## 作用范围

用在类和方法上。

## 属性

### value

指定要返回的HTTP状态码。可以使用HttpStatus枚举类中的常量来指定，如HttpStatus.OK、HttpStatus.NOT\_FOUND等。

### reason

指定返回的HTTP响应原因。如果未指定，将使用默认的原因。

## 示例

### 1、控制器方法中的使用

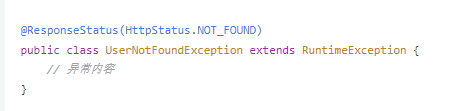
在控制器方法上使用@ResponseStatus注解，可以指定该方法在正常执行时返回的HTTP状态码和原因。如果方法成功执行，将会返回指定的状态码和原因的HTTP响应。



在上述示例中，当执行getUserById方法并成功返回用户信息时，将会返回HTTP状态码200（OK）

### 2、异常类中的使用

在自定义的异常类上使用@ResponseStatus注解，可以指定该异常被抛出时返回的HTTP状态码和原因。当抛出该异常时，Spring框架会自动捕获并返回指定的状态码和原因的HTTP响应。



在上述示例中，当抛出UserNotFoundException异常时，Spring框架会返回HTTP状态码404（NOT\_FOUND）。

## 总结

@ResponseStatus注解用于指定控制器方法或自定义异常类在特定情况下返回的HTTP响应状态码和原因。它提供了一种简便的方式来定义和控制HTTP响应的状态，使得开发人员能够更加灵活地处理和传达与请求相关的错误信息。

# @PathVariable

## 基本说明

将 URL 中占位符参数绑定到控制器处理方法的入参中，URL 中的 {xxx} 占位符可以通过@PathVariable("xxx") 绑定到操作方法的入参中。

## 作用范围

方法的参数上。

## 代码示例



# @RequestParam

## 基本概念

把请求中指定名称的参数传递给控制器中的形参赋值。

## 作用范围

方法的参数上。

## 属性

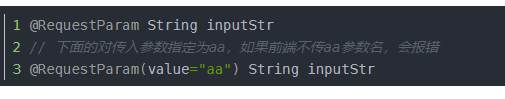
### name

### required

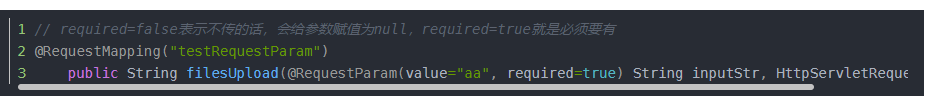
默认值为true。

## 使用方法

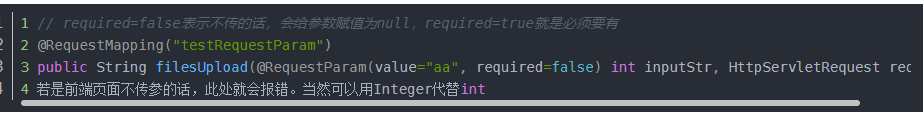
1、可以对传入参数指定参数名



2、可以通过required=false或者true来要求@RequestParam配置的前端参数是否一定要传



3、如果@requestParam注解的参数是int类型，并且required=false，此时如果不传参数的话会报错。原因是required=false时，不传参数的话，会给参数赋值null，这样就会把null赋值给了int，因此会报错。



## 注意事项

默认情况下，@RequestParam注解用于绑定基本类型（如String、Integer等）或数组类型的参数，但是直接使用@RequestParam注解在一个复杂对象上是不支持的，因为Spring无法直接解析请求参数到一个复杂的Java对象中。

# @ModelAttribute

## 基本概念

@ModelAttribute 注解是 Spring 框架中用于数据绑定和预处理的一个重要功能。它主要用于在 Spring MVC 控制器中处理请求时，将请求参数绑定到方法参数或模型属性上。@ModelAttribute 使得在处理 Web 表单数据时更加灵活和高效。

## 用途

### 1、绑定请求参数

@ModelAttribute 用于将 HTTP 请求的参数自动绑定到方法参数或模型属性上。例如，它可以将请求中的表单数据绑定到一个 JavaBean 对象上。

### 2、预处理模型属性

@ModelAttribute 还可以用于在请求处理之前，往模型中添加数据。这些数据可以在视图中进行渲染。

## 用法示例

### 1、绑定请求参数到方法参数



在这个例子中，@ModelAttribute 注解用于将请求中的参数自动绑定到 User 对象上。Spring 会根据请求参数的名称和 User 类的属性进行匹配和绑定。

### 2、预处理模型属性



在这个例子中，@ModelAttribute 注解用于一个方法，这个方法在处理任何请求之前都会被调用。它向模型中添加了 userRoles 属性，可以在视图 userForm 中访问这个属性。

## 作用范围

### 方法参数

当 @ModelAttribute 用于控制器的方法参数时，Spring 会自动将请求参数绑定到该参数指定的对象上。

### 方法

当 @ModelAttribute 用于控制器的方法上时，该方法会在处理请求之前被调用，并且其返回值会添加到模型中。这通常用于提供视图所需的初始数据。

## 属性设置

### value

在 @ModelAttribute 中，value 属性用于指定模型属性的名称。对于方法参数，Spring 会自动使用参数类型的默认名称，但你也可以使用 @ModelAttribute("name") 明确指定。



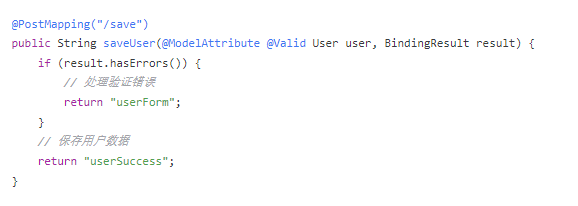
## ****与****@RequestParam****的比较****

@RequestParam 用于绑定单个请求参数到方法参数，而 @ModelAttribute 可以绑定多个请求参数到一个对象上，更适用于复杂的表单数据。

## 注意事项

1、表单数据的绑定: 确保你的 JavaBean 类（如 User）具备合适的 getter 和 setter 方法，以便 Spring 能够进行数据绑定。

2、数据验证: 如果需要对数据进行验证，可以结合使用 @Valid 注解（用于进行数据校验）和 @ModelAttribute 注解。



## 总结

@ModelAttribute 是 Spring MVC 中一个强大的注解，用于简化请求数据的绑定和模型数据的预处理。它使得在处理表单数据和提供视图数据时更加简洁和灵活。通过合理使用 @ModelAttribute，可以提升 Web 应用的开发效率和代码可读性。

# @RequestHeader

## 基本概念

用于从HTTP请求头中获取指定名称的值，然后将其注入到方法参数中。在Spring框架中，@RequestHeader通常与Spring MVC中的Controller方法一起使用，用于获取HTTP请求头中的信息。

# @RequestMapping

## 基本概念

RequestMapping是一个用来处理请求地址映射的注解，可用于类或方法上。用于类上，表示类中的所有响应请求的方法都是以该地址作为父路径。

## 属性

### value

指定请求的实际地址，指定的地址可以是URI Template 模式。

### method

指定请求的method类型， GET、POST、PUT、DELETE等。

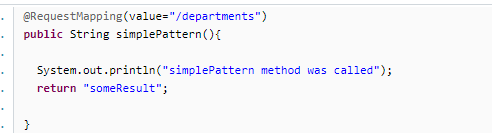
### param

指定request中必须包含某些参数值时才让该方法处理。

## 使用方法

### 1、最基本的方法级别上的应用

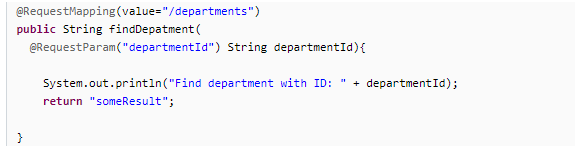
代码



访问http://localhost/xxxx/departments的时候，会调用 simplePattern方法。

### 2、参数绑定

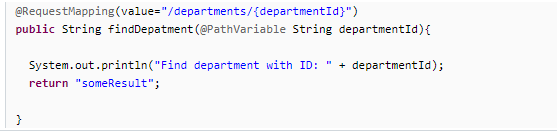
代码



 访问形式 /departments?departmentId=23就可以触发访问findDepatment方法了 。

### 3、REST风格的参数

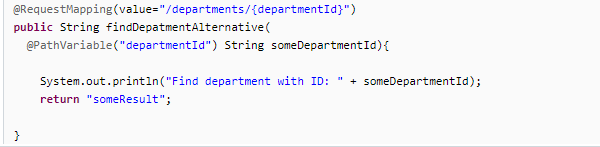
代码



访问形式：/departments/23，其中@PathVariable接收rest风格的参数。

### 4、REST风格的参数绑定

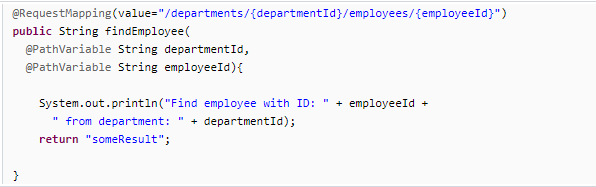
代码



访问形式：/departments/23，把23作为传入的departmetnId,，但是在实际的方法findDepatmentAlternative中，使用 @PathVariable("departmentId") String someDepartmentId将其绑定为 someDepartmentId,所以这里someDepartmentId为23 。

### 5、url中同时绑定多个id

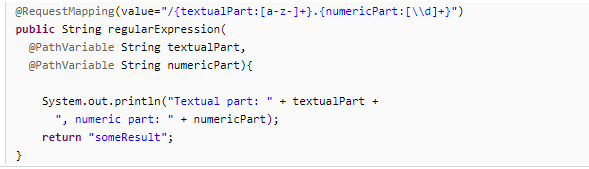
代码



访问形式：/departments/23/employees/21

### 6、支持正则表达式

代码



访问形式：/sometext.123

输出：Textual part: sometext, numeric part: 123

# ****@Component****

## 基本概念

Spring 2.5 在 @Repository 的基础上增加了功能类似的额外三个注解：@Component、@Service、@Controller，它们分别用于软件系统的不同层次。

1、@Component 是一个泛化的概念，仅仅表示一个组件 (Bean) ，可以作用在任何层次。

2、@Service 通常作用在业务层，但是目前该功能与 @Component 相同。

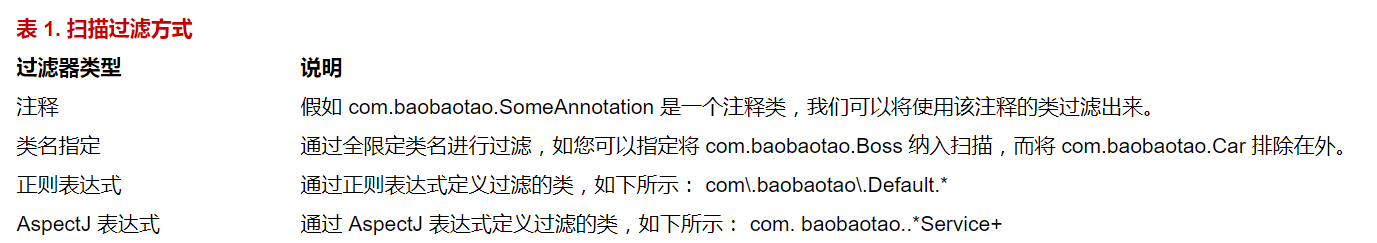
3、@Controller 通常作用在控制层，但是目前该功能与 @Component 相同。

4、通过在类上使用 @Repository、@Component、@Service 和 @Consroller 注解，Spring 会自动创建相应的 Bean 对象，并注册到 ApplicationContext 中，这些类就成了 Spring 受管组件。这三个注解除了作用于不同软件层次的类，其使用方式与 @Repository 是完全相同的。

5、@Component 有一个可选的入参，用于指定 Bean 的名称。一般情况下，Bean 都是 singleton 的，需要注入 Bean 的地方仅需要通过 byType 策略就可以自动注入了，所以大可不必指定 Bean 的名称。在使用 @Component 注释后，Spring 容器必须启用类扫描机制以启用注释驱动 Bean 定义和注释驱动 Bean 自动注入的策略。

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

这里所有通过 <bean> 元素定义 Bean 的配置内容已经被移除，仅需要添加一行 <context:component-scan/> 配置就解决所有问题了，Spring XML 配置文件得到了极致的简化（当然配置元数据还是需要的，只不过以注释形式存在罢了）。<context:component-scan/> 的 base-package 属性指定了需要扫描的类包，类包及其递归子包中所有的类都会被处理。<context:component-scan/> 还允许定义过滤器将基包下的某些类纳入或排除。Spring 支持以下 4 种类型的过滤方式，通过下表说明：



示例：

<context:component-scan base-package="com.baobaotao">

<context:include-filter type="regex"

expression="com\.baobaotao\.service\..\*"/>

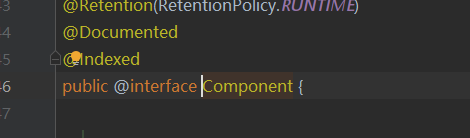
<context:exclude-filter type="aspectj"

expression="com.baobaotao.util..\*"/>

</context:component-scan>

值得注意的是 <context:component-scan/> 配置项不但启用了对类包进行扫描以实施注释驱动 Bean 定义的功能，同时还启用了注释驱动自动注入的功能（即还隐式地在内部注册了AutowiredAnnotationBeanPostProcessor 和 CommonAnnotationBeanPostProcessor），因此当使用 <context:component-scan/> 后，就可以将 <context:annotation-config/> 移除了。默认情况下通过 @Component 定义的 Bean 都是 singleton 的，如果需要使用其它作用范围的 Bean，可以通过@Scope 注释来达到目标。

## 源码



## 属性

### String value()

组件的名称

# @SessionAttributes

# @SessionAttribute

@SessionAttributes注解是用来向HttpSession中存放数据的；

@SessionAttribute注解是用来从HttpSession中读取数据的。

# @Controller

## 基本概念

@Controller 用于标记在一个类上，使用它标记的类就是一个SpringMVC Controller 对象。分发处理器将会扫描使用了该注解类的方法，并检测该方法是否使用了@RequestMapping 注解。

## 属性

# @Service

## 作用

在类上面添加注解可以使用Spring自动向Bean容器中创建该类的Bean，而在我们需要用到该Bean来创建对象时，在变量声明上方添加@AutoWired或者@Qualifier即可。

## 工作原理

在service层类上添加@Service注解，会自动创建Bean，Bean的名字为类的名字首字母小写，也可以自己指定名字@Service(value="XXXX")，这个名字或者说Bean的名字有什么用呢？是为了使用名字进行属性注入时使用的。

## 属性

### value

声明bean的名称

# @Repository

## 基本概念

@Repository是Spring框架中的一个注解，用于标记一个DAO（数据访问对象）组件。

## 作用

将一个DAO组件注册到Spring容器中，并使其能够在需要时被其他组件自动注入，从而实现依赖注入和控制反转。此外，@Repository还可以与其它相关注解配合使用，例如@Transactional。

# @RestController

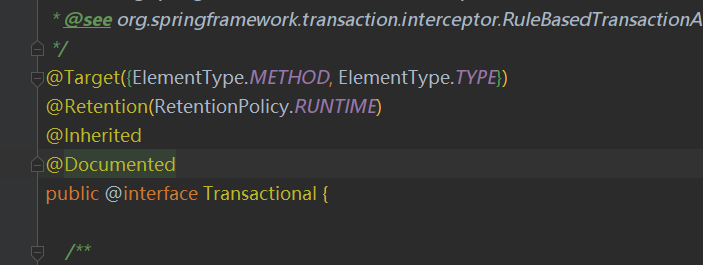
## 基本概念

作用等同于@Controller + @ResponseBody 注解。

# @Transactional

## 基本概念

## 源码



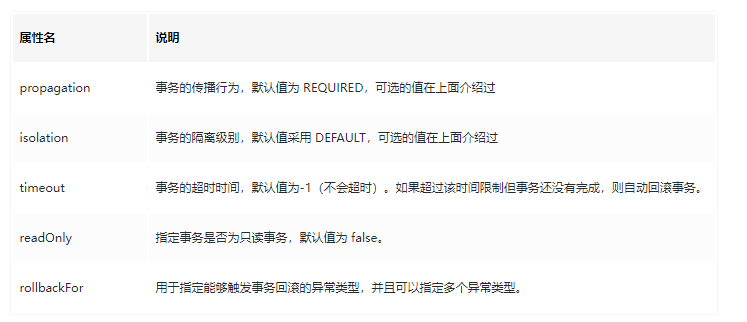
## 作用范围

1、方法：推荐将注解使用于方法上，不过需要注意的是：该注解只能应用到 public方法上，否则不生效。

2、类：如果这个注解使用在类上的话，表明该注解对该类中所有的 public 方法都生效。

3、接口：不推荐在接口上使用。

## 常用的配置参数



### propagation

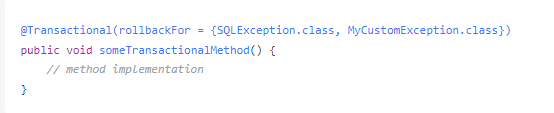
事务的传播行为，默认值是REQUIRED。

### rollbackFor

事务回滚的异常类型。

默认情况下，@Transactional 注解的 rollbackFor 属性是空的，也就是说没有指定任何异常类型。这意味着默认情况下，Spring 在发生未检查异常（继承自 RuntimeException 的异常）或 Error 时会回滚事务，而对于已检查异常（非继承自 RuntimeException 的异常），Spring 不会自动回滚事务。

rollbackFor 属性允许我们指定一个异常类或异常类数组，用于说明哪些异常发生时需要回滚事务。例如：



在上面的例子中，如果方法 someTransactionalMethod 抛出了 SQLException 或 MyCustomException 类型的异常，Spring 会回滚事务。如果抛出的异常不属于指定的回滚异常类型，事务不会回滚（除非是未检查异常或 Error）。

### noRollbackFor

不会发生事务回滚的异常类型。

## 原理

我们知道@Transactional 的工作机制是基于 AOP 实现的，AOP 又是使用动态代理实现的。如果目标对象实现了接口，默认情况下会采用 JDK 的动态代理，如果目标对象没有实现接口,会使用CGLIB动态代理。多提一嘴：createAopProxy() 方法决定了是使用 JDK 还是 Cglib 来做动态代理，源码如下：



如果一个类或者一个类中的 public 方法上被标注@Transactional 注解的话，Spring 容器就会在启动的时候为其创建一个代理类，在调用被@Transactional 注解的 public 方法的时候，实际调用的是TransactionInterceptor类中的 invoke()方法。这个方法的作用就是在目标方法之前开启事务，方法执行过程中如果遇到异常的时候回滚事务，方法调用完成之后提交事务。

TransactionInterceptor 类中的 invoke()方法内部实际调用的是 TransactionAspectSupport 类的invokeWithinTransaction()方法。由于新版本的 Spring 对这部分重写很大，而且用到了很多响应式编程的知识。

## Spring AOP自调用问题

若同一类中的其他没有 @Transactional 注解的方法内部调用有 @Transactional 注解的方法，有@Transactional 注解的方法的事务会失效。这是由于Spring AOP代理的原因造成的，因为只有当 @Transactional 注解的方法在类以外被调用的时候，Spring事务管理才生效。MyService 类中的method1()调用method2()就会导致method2()的事务失效。

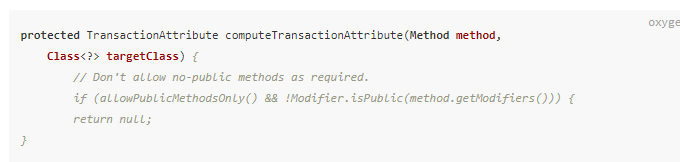


解决办法就是避免在同一类中自调用或者使用AspectJ取代Spring AOP代理。

## @Transactional注解失效的情况

### 1、@Transactional 应用在非 public 修饰的方法上

之所以会失效是因为在Spring AOP 代理时，TransactionInterceptor （事务拦截器）在目标方法执行前后进行拦截，DynamicAdvisedInterceptor（CglibAopProxy 的内部类）的 intercept 方法或 JdkDynamicAopProxy 的 invoke 方法会间接调用 AbstractFallbackTransactionAttributeSource的 computeTransactionAttribute 方法，获取Transactional 注解的事务配置信息。



此方法会检查目标方法的修饰符是否为 public，不是 public则不会获取@Transactional 的属性配置信息。

注意：protected、private 修饰的方法上使用 @Transactional 注解，虽然事务无效，但不会有任何报错，这是我们很容易犯错的一点。

### 2、@Transactional 注解属性propagation设置错误

### 3、@Transactional 注解属性rollbackFor设置错误

### 4、同一个类中方法调用，导致@Transactional失效

### 5、异常被你的 catch“吃了”导致@Transactional失效

### 6、数据库引擎不支持事务

## 使用注意事项总结

1、@Transactional 注解只有作用到 public 方法上事务才会生效，不推荐在接口上使用。

2、避免在同一个类中调用 @Transactional 注解的方法，这样会导致事务失效。

3、正确设置 @Transactional 的 rollbackFor 和 propagation 属性，否则事务可能会回滚失败。

4、被 @Transactional 注解的方法所在的类必须被Spring管理，否则不生效。

5、底层使用的数据库必须支持事务机制，否则不生效。

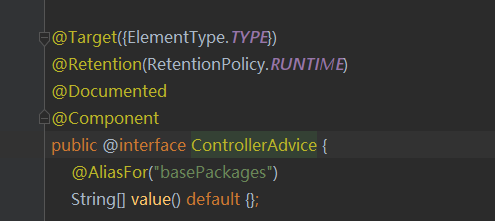
## 示例

# @ControllerAdvice

## 基本概念

@ControllerAdvice 是 Spring 框架中的一个重要注解，用于定义一个全局的控制器增强器（controller advice），可以用于处理跨多个控制器的通用任务。

## 源码



## 主要作用

### 1、全局异常处理

可以捕获应用中的异常，并返回自定义的响应格式。

### 2、全局数据绑定

可以为所有控制器添加通用的模型属性（model attributes）。

### 3、全局响应体处理

可以在响应体写入前，统一修改响应格式。

## 具体功能

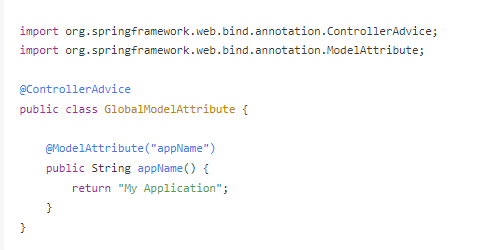
### 1、全局异常处理

通过@ExceptionHandler注解，可以在 @ControllerAdvice 类中定义方法来处理特定异常，提供自定义的错误响应。例如：



### 2、全局数据绑定

通过@ModelAttribute 注解，可以在 @ControllerAdvice 类中定义方法，以添加全局的模型属性。这些属性将自动添加到每个控制器的模型中。例如：



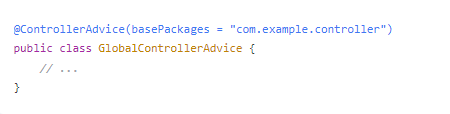
在任何控制器中，都可以使用 ${appName} 获取该属性。

### 3、全局响应体处理

通过@ResponseBody和ResponseBodyAdvice 的结合，可以在所有响应体写入前进行统一处理。实现 ResponseBodyAdvice 接口后，可以对响应体进行修改或包装。

## 作用范围

@ControllerAdvice 可以指定适用的控制器包（或类），以限制该增强器的作用范围。可以通过 basePackages 或 assignableTypes 属性进行指定



## 示例

如何使用@ControllerAdvice 进行全局异常处理和添加模型属性。



## 总结

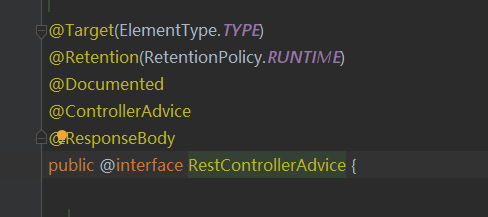
@ControllerAdvice是一个强大的工具，允许开发者在 Spring MVC 应用中集中处理跨控制器的通用逻辑。通过它，可以简化代码，减少重复，增强应用的可维护性。使用时结合异常处理、模型属性添加和响应体处理，可以实现强大的功能，提升用户体验。

# @RestControllerAdvice

## 基本概念

Spring框架中的一个注解，用于实现全局性的异常处理和响应封装。它可以应用于类级别，用于统一处理所有标记了@ResponseBody注解的Controller类的异常和响应。

## 源码



## 属性

### value

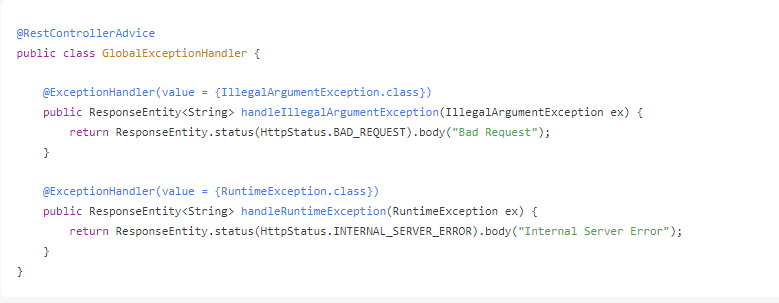
### basePackages

## 作用

使代码更加简洁和易于维护，通过统一处理异常和响应，避免出现重复的异常处理和响应封装代码，提高开发效率和代码质量。

## 使用方法

要使用@RestControllerAdvice注解，只需将其添加到一个类上即可。该类可以包含多个标记了@ExceptionHandler、@InitBinder、@ModelAttribute注解的方法，分别处理异常、绑定器初始化和模型属性的相关操作。



在上述代码中，我们定义了一个名为GlobalExceptionHandler的全局异常处理类，使用@RestControllerAdvice注解进行标记。该类中包含两个标记了@ExceptionHandler注解的方法，分别处理IllegalArgumentException和RuntimeException异常，并封装对应的响应结果。

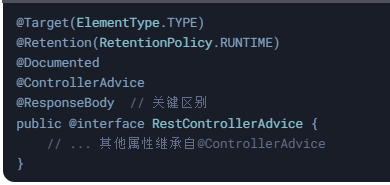
## @RestControllerAdvice和@ControllerAdvice注解的区别

### 1、核心区别总结



### 2、源码级差异

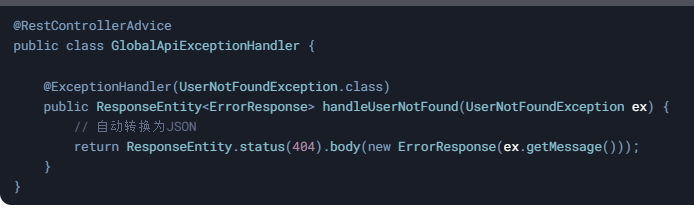
@RestControllerAdvice 定义



它本质上是@ControllerAdvice和@ResponseBody的组合注解。

### 3、使用场景对比

#### 场景1：RESTful API（推荐使用@RestControllerAdvice）



效果：无需额外注解，自动返回JSON结构。

#### 场景2：传统Spring MVC（使用@ControllerAdvice）



### 4、共性功能

二者均支持以下特性：

1、全局异常处理（@ExceptionHandler）

2、全局数据绑定（@ModelAttribute）

3、全局数据预处理（@InitBinder）

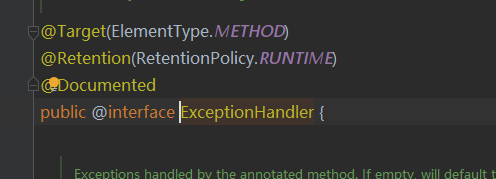
4、指定作用范围（通过basePackages/annotations等属性）

# @ExceptionHandler

## 基本概念

用于定义处理Controller类中抛出的指定类型异常的方法。通过该注解，我们可以将对特定异常的处理逻辑与业务逻辑分离开来，提高应用程序的可维护性和可读性。

## 源码



## 作用

使代码更加简洁和易于维护，通过统一处理Controller类中的异常，避免出现重复的异常处理代码，提高开发效率和代码质量。

## 使用方法

要使用@ExceptionHandler注解，只需在Controller类中定义一个参数类型为指定异常类型，且标记了@ExceptionHandler注解的方法即可。



在上述代码中，我们在UserController类中定义了一个getUserById方法，当传入的参数id无效时，我们会手动抛出一个IllegalArgumentException异常。为了统一处理这类异常，我们在同一个类中定义了一个handleIllegalArgumentException方法，标记了@ExceptionHandler(value={IllegalArgumentException.class})注解，用于处理IllegalArgumentException异常，并封装对应的响应结果。

## 注意事项

1、@ExceptionHandler注解只能应用于Controller类中，用于定义处理特定异常类型的方法。

2、在使用@ExceptionHandler注解时，需要注意处理的异常类型和返回的响应结果类型，避免出现类型不匹配或其它异常的情况。

3、建议每个Controller中单独定义一个异常处理方法，以保证异常处理逻辑的清晰和可读性。

# @Path

## 基本概念

@Path这个是JAVA中JAX-RS (JSR-311) 实现的注解，JAX-RS (JSR-311) 是为Java EE 环境下的RESTful 服务能力提供的一种规范，**@Path**就当@RequestMapping来用。

# @Configuration

## 基本概念

@Configuration注解用于定义配置类，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。

## 作用范围

用在类上。

## 属性

### value

组件的名称

### proxyBeanMethods

用于控制是否启用基于代理的方法调用。默认情况下，proxyBeanMethods 属性设置为 true，表示启用基于代理的方法调用。

当 proxyBeanMethods 设置为 true 时：

在每次调用 @Bean 方法时，Spring 将动态地生成一个 CGLIB 代理对象。该代理对象将负责管理与 @Bean 方法相关联的 bean 实例的创建和生命周期。每次对 @Bean 方法的调用都会返回同一个代理对象的实例，而不是实际的bean实例。

当 proxyBeanMethods 设置为 false 时：

每次调用 @Bean 方法时，Spring 将返回实际的bean实例，而不是代理对象。这意味着即使两次调用相同的 @Bean 方法，也会得到不同的 bean 实例。此选项适合那些对性能敏感，并且依赖 @Bean 方法多次调用返回不同实例的场景。这样做的好处：

1、提高启动性能

2、减少内存占用

3、避免代理相关的问题。

## 与@Component注解的区别

### 基本概念对比

@Component是Spring的基础组件注解，用于标识一个类作为Spring容器管理的Bean。它是一个通用的构造型注解，表示被注解的类是一个"组件"。

@Configuration是一个特殊的@Component，专门用于定义配置类。它告诉Spring这个类包含一个或多个@Bean方法，这些方法将返回由Spring容器管理的Bean实例。

### 核心区别分析

#### 1、代理机制的差异

@Configuration的CGLIB代理



Spring会为@Configuration类创建CGLIB代理，确保多次调用userRepository()方法时返回同一个Bean实例，维护单例特性。

@Component的普通处理



@Component类不会被代理，每次方法调用都是普通的Java方法调用，可能破坏单例模式。

#### 2、Bean定义方式的不同

@Configuration适合集中式配置



@Component适合单一职责组件



### 具体使用场景

#### @Configuration的使用场景

##### 1、第三方库集成配置



##### 2、安全配置



##### 3、多环境配置管理



#### @Component使用场景

##### 1、业务服务层（@Service）



##### 2、数据访问层（@Repository）



##### 3、Web控制层（@Controller）



### 选择原则和最佳实践

#### 使用@Configuration的情况

1、需要定义多个相关联的Bean

2、配置第三方库或框架

3、Bean之间存在依赖关系且需要保证单例

4、需要条件化配置（配合@Conditional注解）

5、需要导入其他配置类或资源

#### 使用@Component及其衍生注解的情况

1、定义业务逻辑组件

2、单一职责的服务类

3、数据访问对象

4、工具类或辅助类

5、不需要复杂Bean创建逻辑的场景

### 性能和内存考虑

@Configuration类由于CGLIB代理会有轻微的性能开销，但换来的是更严格的单例保证。@Component类则更轻量，适合大量的业务组件。

在实际项目中，合理区分和使用这两个注解能够让代码结构更清晰，配置更易维护，同时确保Spring容器的正确行为。配置类专注于基础设施搭建，组件类专注于业务逻辑实现，各司其职，共同构建健壮的应用架构。

# @Bean

## 基本概念

@Bean注解用于告诉方法产生一个Bean对象，然后这个Bean对象交给Spring管理。产生这个Bean对象的方法Spring只会调用一次，随后Spring将这个Bean对象放在自己的IOC容器中。

## 作用范围

方法或者注解类上。

## 基本用法

### 1、简单Bean声明



### 2、指定Bean名称



## 属性

### name

Bean的名称

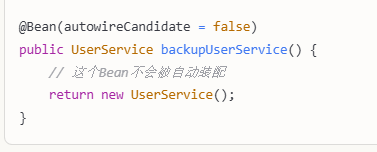


### initMethod和destroyMethod



### autowireCandidate

是否自动装配，默认为true



## 注意事项

1、循环依赖: 避免Bean之间的循环依赖

2、作用域混合: 注意不同作用域Bean之间的引用关系

3、线程安全: 确保单例Bean的线程安全性

4、资源管理: 正确使用initMethod和destroyMethod管理资源

## 总结

@Bean注解提供了灵活而强大的Bean定义方式，是Spring配置的核心工具，特别适合需要复杂初始化逻辑或集成第三方库的场景。

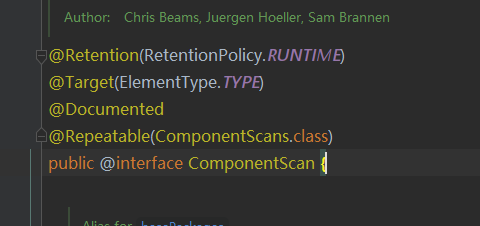
# @ComponentScan

## 基本概念

@ComponentScan注解一般和@Configuration注解一起使用，主要的作用就是定义包扫描的规则，然后根据定义的规则找出哪些类需要自动装配到Spring的bean[容器](https://cloud.tencent.com/product/tke?from=10680)中，然后交由spring进行统一管理。

说明：针对标注了@Controller、@Service、@Repository、@Component 的类都可以被Spring扫描到。

## 源码



## 作用范围

用在类上。

## 属性

### basePackages

用于扫描带注解的基础包

### value

basePackages属性的别名

### basePackageClasses

## 默认扫描的路径

### 1、默认包路径

如果@ComponentScan 注解没有显式设置 basePackages 或 basePackageClasses 属性，那么Spring 默认扫描包含 @ComponentScan 注解的配置类所在的包及其子包。换句话说默认情况下，Spring 会扫描配置类的包以及配置类包中的所有子包。



在上面的例子中，Spring 会扫描 AppConfig 所在的包及其子包。例如，如果 AppConfig 位于 com.example 包中，那么默认扫描路径将包括 com.example 包以及 com.example 包的所有子包（如 com.example.service、com.example.repository 等）。

### 2、基于配置类位置

默认的扫描路径是相对于配置类所在的包。配置类的包决定了扫描的起点。如果你的配置类放在不同的包中，可以通过设置 basePackages 或 basePackageClasses 属性来改变默认扫描路径。

## 示例

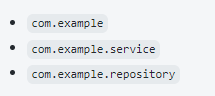
假设你的项目目录结构如下：



并且你的 AppConfig 类如下：



在这种情况下，AppConfig 类位于 com.example 包中，因此 Spring 会扫描以下包：



# @ComponentScans

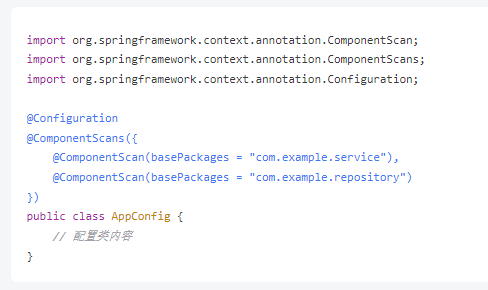
## 基本概念

@ComponentScans 是一个 Spring 框架中的注解，用于配置多个 @ComponentScan 注解，以便在 Spring 应用上下文中扫描多个包以查找 Spring 组件。它通常在 Spring Boot 或 Spring Framework 的配置类中使用。

@ComponentScans 注解用于在一个配置类中配置多个 @ComponentScan 注解。它的目的是在同一个配置类中提供灵活的包扫描配置，以便支持扫描多个不同的包。

@ComponentScans 注解通常用于需要配置多个 @ComponentScan 注解的场景，这在实际项目中可能会出现，例如，当应用程序结构复杂时，需要从多个不同的包中扫描组件。

## 示例代码

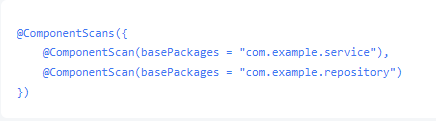


在这个例子中，@ComponentScans 注解包含了两个 @ComponentScan 注解，分别扫描 com.example.service 和 com.example.repository 包。这允许 Spring 在这两个包及其子包中扫描并注册组件。

## 属性

### value

它用于指定一个或多个 @ComponentScan 注解。一个 @ComponentScan 注解数组，用于指定多个 @ComponentScan 配置。



## 使用场景

### 1、复杂的包结构

当应用程序有复杂的包结构，需要从多个包中扫描组件时，可以使用 @ComponentScans 来灵活配置。

### 2、模块化项目

在模块化的项目中，可能需要分别配置不同的包进行扫描，以确保每个模块的组件都被正确加载。

### 3、多配置类

如果在同一个配置类中需要配置不同的扫描策略，可以使用 @ComponentScans 注解来简化配置。

## ****与**** @ComponentScan ****区别****

1、@ComponentScan 注解用于扫描一个包或多个包，而 @ComponentScans 则是一个复合注解，可以同时配置多个 @ComponentScan 注解。

2、自定义扫描: 除了基本的包扫描配置，@ComponentScan 还支持更多的自定义配置选项，如 includeFilters 和 excludeFilters，以控制扫描的组件类型和范围。

## 总结

@ComponentScans 注解在 Spring 应用中用于配置多个 @ComponentScan 注解，从而支持扫描多个包。它帮助开发者在复杂项目中灵活配置组件扫描，简化了 Spring 应用上下文的组件注册过程。

# @EnableScheduling

## 基本概念

开启定时任务的自动配置。

# @Scheduled

## 基本概念

这个注解的作用是进行定时任务的调度。

# @Import

用来导入配置类或者一些需要前置加载的类。

# @ContextConfiguration

## 基本概念

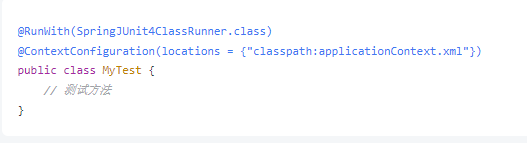
用于指定Spring测试类（如JUnit测试类）的上下文配置信息，以便在测试过程中加载Spring容器并进行相应的依赖注入。

@ContextConfiguration这个注解通常与@RunWith(SpringJUnit4ClassRunner.class)联合使用用来测试。

## 作用范围

用在类上

## 代码示例



# @Indexed

可以为Spring 的模式注解添加索引，以提升应用启动性能。

# @ActiveProfiles

一般声明在UT测试类上，用于指定这个测试类里的测试方式运行时的profiles。

# @EnableMBeanExport

## 基本概念

为了 **配置 Spring 的 JMX（Java Management Extensions）MBean 的注册行为。**

## 作用

启用Spring 对JMX MBean 的自动导出，将 Spring 管理的 Bean 暴露为 JMX 管理接口。

## 属性

### registration

Spring 提供了三种 MBean 注册冲突解决策略



## 应用场景

监控应用运行时状态（如线程池、数据源）、动态修改配置等。

## ****为什么需要****IGNORE\_EXISTING

1、Spring Boot 应用部署在 Servlet 容器（如 Tomcat）中：  
容器可能已注册了某些 MBean（如线程池），与应用的 MBean 名称冲突。

2、多应用共享同一个 MBeanServer：  
多个 Spring 上下文尝试注册同名 MBean 时引发冲突。

解决方案

通过 IGNORE\_EXISTING 策略：

1、避免因冲突导致应用启动失败。

2、允许应用与其他组件（如容器）共享 MBean。

## 注意事项

1、性能影响：JMX 会引入少量性能开销，生产环境需权衡是否需要启用。

2、安全性：暴露 MBean 可能带来安全风险，应限制访问（如配置 JMX 密码或网络隔离）

3、容器兼容性：在 Tomcat/JBoss 等容器中运行时，需测试与容器自带 MBean 的兼容性。

# @SpringBootApplication

## 基本概念

组合注解，由@Configuration，@EnableAutoConfiguration，@ComponentScan这三个注解组成的。

## 属性

### scanBasePackages

# @SpringBootConfiguration

## 基本概念

@SpringBootConfiguration 注解是 Spring Boot 中的一个重要注解，实际上是 @Configuration 的一种特殊形式，用于标记一个类为 Spring Boot 应用程序的配置类。

## 作用

### 1、标记配置类

@SpringBootConfiguration 用于表明该类包含 Spring Boot 应用程序的配置。它意味着这个类可以用来定义 Bean、配置属性等。这样，Spring Boot 可以自动扫描并加载这些配置。

### 2、结合@EnableAutoConfiguration

@SpringBootConfiguration 通常与 @EnableAutoConfiguration 和 @ComponentScan 一起使用，以实现自动配置和组件扫描。@EnableAutoConfiguration 让 Spring Boot 能够根据依赖的库自动配置应用程序的各种功能，而 @ComponentScan 则指示 Spring 需要扫描的包，以便发现其他组件（如服务、控制器等）。

### 3、继承@Configuration

@SpringBootConfiguration 是 @Configuration 的一个专门化版本，所有在 @Configuration 上的特性也适用于 @SpringBootConfiguration。这意味着可以使用 @Bean 注解在此类中定义 Bean，Spring 将自动管理它们的生命周期。

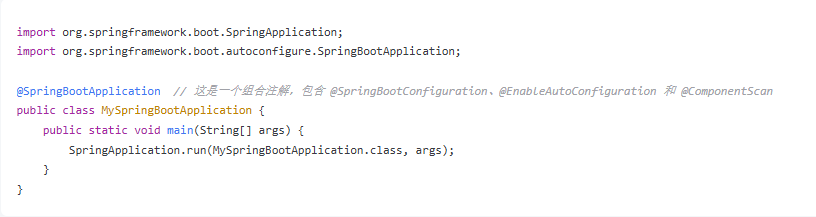
### 4、使配置更清晰

使用 @SpringBootConfiguration 可以帮助开发者更明确地了解这个配置类是为 Spring Boot 应用程序而设计的，与普通的 Spring 配置类有区别，从而提高代码的可读性和维护性。

### 5、默认包扫描

@SpringBootConfiguration 类通常是 Spring Boot 应用的启动类，因此 Spring Boot 默认会从这个类所在的包开始进行组件扫描。这确保了在启动时能够自动检测到该类所在包中的所有组件。

## 示例



## 总结

@SpringBootConfiguration 是用于定义 Spring Boot 应用程序的配置类的注解，它结合了 Spring Boot 的自动配置特性，使得开发者可以更轻松地构建和配置应用程序。通过使用它，开发者可以有效地管理应用程序的上下文和 Bean 的创建，提升开发效率。

# @EnableCaching

@EnableCaching是由Spring框架提供的，Spring Boot框架对该注解进行了继承，该注解需要配置在类上（在Spring Boot中，通常配置在项目启动类上），用于开启基于注解的缓存支持。

# @CacheConfig

## 基本概念

@CacheConfig是Spring框架中的一个注解，用于配置缓存通用属性。它可以应用于类级别和方法级别，用于配置缓存的名称、key生成规则、缓存的生命周期等通用属性。

## 作用

使代码更加简洁和易于维护，通过统一管理缓存的通用属性，避免重复的缓存配置代码出现在每个方法中。

# @Cacheable

## 基本概念

@Cacheable注解也是由Spring框架提供的，可以作用于类或方法（通常用在数据查询方法上），用于对方法结果进行缓存存储。@Cacheable注解的执行顺序是先进行缓存查询，如果为空则进行方法查询，并将结果进行缓存；如果缓存中有数据，不进行方法查询，而是直接使用缓存数据。@Cacheable注解提供了多个属性，用于对缓存存储进行相关配置，具体属性及说明如下表：



## 属性

### value

指定缓存空间的名称

### key

指定缓存数据的key

### cacheManager

# @CachePut

## 基本概念

它的主要作用是**将方法的返回值存入缓存**，同时**不影响方法的执行**。与 @Cacheable 不同，@CachePut 不会跳过方法的执行，而是每次都会执行方法，并将结果更新到缓存中。

## 作用

### 1、更新缓存

将方法的返回值存储到缓存中，如果缓存中已存在相同的键，则会覆盖旧值。

### 2、不影响方法执行

无论缓存中是否存在对应的键，@CachePut 都会执行方法体，并将结果存入缓存。

## 属性

### value

### cacheNames

指定缓存的名称（必填）

### key

指定缓存的键，支持SpEL表达式

### condition

指定条件，满足条件时才缓存

### unless

指定条件，满足条件时不缓存

# @CacheEvict

## 基本概念

在 Spring 框架中，@CacheEvict 注解用于清除缓存中的数据。当使用该注解标记在方法上时，每次调用该方法后，缓存中的相关数据将被移除。@CacheEvict 注解可以用于任何带有缓存注解（如 @Cacheable、@CachePut）的方法上，以便在方法执行后从缓存中移除指定的数据。

## 作用范围

用在类或者方法上

## 属性

### String[] value

cacheNames属性的别名。

### String[] cacheNames

指定要清除的缓存名称，可以是一个或多个缓存的名称。多个名称之间用逗号隔开。如果要清除多个缓存中的数据，可以使用数组指定多个名称。

### key

指定要清除的缓存项的键值。可以使用 SpEL 表达式，通过方法参数或其他方式动态指定。如果不指定，则默认使用方法的所有参数作为缓存的 key。

### allEntries

默认为 false。如果设置为 true，则会清除整个缓存中的所有数据，无视 key 参数。

### beforeInvocation

默认为 false。如果设置为 true，则在方法执行之前触发缓存清除操作。

## 注意

@CacheEvict 注解只会触发缓存的清除操作，不会返回已删除的数据。如果需要在方法执行后返回更新后的数据并放入缓存，可以考虑使用 @CachePut 注解。实际使用时，需正确配置缓存管理器，并在 Spring 应用上下文中启用缓存，以确保缓存的正常使用。

## 说明

这个注解在spring-context-5.2.10.RELEASE.jar包中。

# @Mapper

## 作用范围

一般在接口类上

## 作用

在接口类上添加了@Mapper注解，在运气时期会生成相应的接口实现类，如果想要每个接口都要变成实现类，那么需要在每个接口类上加上@Mapper注解，比较麻烦，解决这个问题用@MapperScan。

# @MapperScan

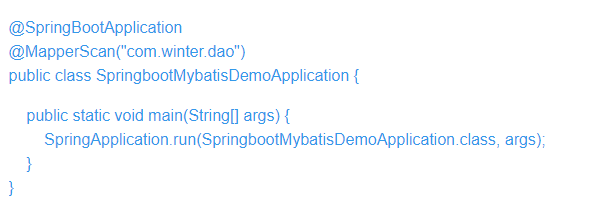
## 作用范围

写在类上

## 作用

指定要变成实现类的接口所在的包，然后包下面的所有接口在运行时期会生成相应的实现类，在Springboot启动类上面添加。

## 代码示例



添加@MapperScan(“com.winter.dao”)注解以后，com.winter.dao包下面的接口类，在运行时期都会生成相应的实现类。

## 说明

配置了@MapperScan注解后，@Mapper注解并不会失效，它们可以一起使用。@MapperScan用于指定Mapper接口的扫描路径并注册为Spring的Bean，而@Mapper注解则是标记Mapper接口并定义具体的SQL操作。这样可以更灵活地配置和管理MyBatis的Mapper接口。

# @EnableAutoConfigration

自动配置注解，在启动Spring应用程序上下文时进行自动配置，自动配置通常是基于项目classpath中引入的类和已定义的bean来实现的。

# @AutoConfigureBefore

# @AutoConfigureAfter

# @ServletComponentScan

## 基本概念

@ServletComponentScan是Spring Boot中的一个注解，用于扫描和注册使用了@WebServlet、@WebFilter和@WebListener注解的组件，以便它们能够被容器正确地加载和使用。

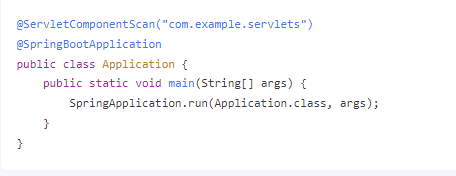
在传统的Java Web开发中，我们通常使用web.xml文件来配置和注册Servlet、Filter和Listener等组件。但是在Spring Boot中，为了简化配置和提高开发效率，可以使用基于注解的方式来实现相同的功能。@ServletComponentScan注解的作用是扫描指定包及其子包下的组件，并自动将带有@WebServlet、@WebFilter和@WebListener注解的组件注册到Servlet容器中。

## 属性

### value

### basePackages

## 示例



在上述示例中，@ServletComponentScan注解指定了要扫描的包路径为"com.example.servlets"，该包下的带有@WebServlet、@WebFilter和@WebListener注解的组件将会被自动扫描并注册到Servlet容器中。通过使用@ServletComponentScan注解，我们不再需要手动编写web.xml文件来配置组件，而是可以直接使用注解进行配置。这样可以减少配置的工作量，提高开发效率，并且更符合Spring Boot的习惯。需要注意的是，@ServletComponentScan注解通常与@SpringBootApplication注解一起使用，后者是Spring Boot的核心注解，用于启用Spring Boot的自动配置和组件扫描等功能。总结来说，@ServletComponentScan注解用于在Spring Boot中扫描、注册带有@WebServlet、@WebFilter和@WebListener注解的组件。通过使用该注解，可以简化Java Web应用程序的配置过程，提高开发效率，并符合Spring Boot的开发习惯。

# 条件注解

## @Conditional

@Conditional是Spring4版本新提供的一种注解，它的作用是按照设定的条件进行判断，把满足判断条件的bean注册到Spring容器。

## @ConditionalOnClass

### 基本概念

@ConditionalOnClass 是 Spring Framework 提供的一个条件注解，用于在某个类存在于 Classpath 中时才会生效。它可以应用于类级别或方法级别的注解上。@ConditionalOnClass 注解的作用是根据指定的类是否存在于 Classpath 中来决定是否启用或禁用配置类、组件、自动配置等。当被注解的类存在于 Classpath 中时，该注解修饰的配置类或组件被激活，并参与 Spring 容器的加载和初始化过程。如果指定的类不存在于 Classpath 中，则该注解修饰的配置类或组件将被忽略，不会起作用。

### 作用范围

类或者方法上。

### 使用场景

1、自动配置类的条件加载：使用 @ConditionalOnClass 注解，可以根据某个特定的类是否存在来决定是否启用自动配置类，这样可以确保只有在依赖的类存在时才会自动配置相关的功能。

2、Bean条件加载：通过在方法上添加 @ConditionalOnClass 注解，可以根据某个类是否存在来决定是否创建特定的 Bean，这样可以根据依赖类的存在与否来动态地加载不同的 Bean。

3、条件化的模块集成：通过使用 @ConditionalOnClass 注解，可以根据第三方库是否存在来选择性地加载模块。这样可以避免在缺少依赖类的情况下导致应用程序异常或失败。

### 使用示例



在上面的示例中，MyConfiguration 是一个配置类，它被 @ConditionalOnClass(SomeClass.class) 注解修饰。这意味着只有当 SomeClass 类存在于 Classpath 中时，该配置类才会被加载和应用。

### 总结

@ConditionalOnClass 注解使得我们能够根据类的存在与否来动态地选择是否加载配置类、组件或 Bean，从而实现更加灵活的条件化编程。

## @ConditionalOnProperty

### 基本概念

用于根据配置属性的值来决定是否生效。通过该注解，可以根据应用配置文件中的属性值来动态地控制某个Bean或配置类是否被加载到容器中。

### 属性

#### prefix

属性的前缀

#### name

指定配置属性的名称。

#### havingValue

指定配置属性的期望值。如果未设置该属性，则默认为任意非空值都满足条件。

#### matchIfMissing

指定当配置属性未定义时，是否满足条件。默认为false，即未定义时不满足条件。

### 使用示例



在上面的代码中，@ConditionalOnProperty注解表示：当名为myapp.feature.enabled的配置属性的值为true时，该配置类才会生效，对应的Bean将被加载到容器中。如果配置属性的值不为true，则该配置类及其Bean将被忽略。

### 总结

@ConditionalOnProperty是Spring Boot提供的一个条件注解，用于根据配置属性的值来决定是否加载某个Bean或配置类到容器中。通过配置属性的名称、期望值和其他属性的组合，可以灵活地控制应用在不同环境下的行为。

## @ConditionalOnMissingBean

### 基本概念

用于在容器中注册一个Bean时，仅在该Bean不存在时生效。如果容器中已经存在了该Bean，则该注解对应的配置类中的Bean方法将被忽略，不会加载到容器中。

### 作用范围

类或者方法上

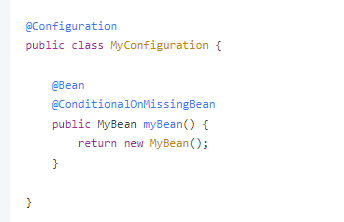
### 属性

#### name

#### value

#### type

### 使用示例



在上面的代码中，@ConditionalOnMissingBean注解表示：如果容器中不存在名为myBean的Bean，则注册MyBean类型的Bean到容器中。如果容器中已经存在了名为myBean的Bean，则该方法将不会被调用，对应的Bean也不会被注册到容器中。需要注意的是@ConditionalOnMissingBean注解只有在当前配置类中才会生效，而不会影响其他配置类中同名的Bean定义。如果不希望重复注册同名的Bean，可以使用@Primary注解来标记某个Bean作为首选实现。

### 总结

@ConditionalOnMissingBean是Spring Boot提供的一个条件注解，用于在容器中注册一个Bean时，仅在该Bean不存在时生效。它可以通过指定参数类型或者使用@Primary注解来避免重复注册同名的Bean。

### @ConditionalOnWebApplication

# @Filter

## 基本概念

@Filter是Spring框架中的一个注解，用于筛选出指定类型的Bean并进行处理。它可以应用于类级别和方法级别，用于实现定制化的Bean过滤和处理逻辑。

## 作用

使代码更加灵活和可扩展，通过自定义实现特定接口的Bean或对标记有指定注解的Bean进行过滤和处理操作，从而实现对应的业务逻辑。

# Validation校验参数涉及的注解

## 说明

## 引入的依赖

## @AssertFalse

## @AssertTrue

## @DecimalMax

## @Valid

### 基本概念

@Valid是Java Bean Validation规范中的一个注解，用于触发对嵌套对象的数据校验。通过该注解，我们可以将数据校验扩展到嵌套对象上，实现更加全面、灵活的数据校验。

### 作用范围

用在方法、字段、参数上

### 作用

触发对嵌套对象的数据校验，通过在需要进行数据校验的类或属性中添加该注解，可以对其内部的嵌套对象进行数据校验。

### 使用方法

要使用@Valid注解，只需在进行数据校验的类或属性上添加该注解即可。当需要对嵌套对象进行数据校验时，需要在嵌套对象类型的属性上同时添加@Valid注解。



在上述代码中，我们定义了一个名为Order的类和一个名为User的类，并在Order类中嵌套了一个User对象。在Order类中，需要对User对象进行数据校验，因此在user属性上添加了@Valid注解。在User类中，我们定义了两个属性username和age，并对其进行了数据校验。在username属性上添加了@NotNull注解，表示该属性不能为空，在age属性上添加了@Min注解并设置了其value参数为18，表示该属性必须大于等于18岁。当我们执行数据校验时，会自动触发对Order对象以及其嵌套的User对象进行数据校验。

## @NotBlank

## @NotNull

## @Size

## @Pattern

## @Constraint

### 基本概念

@Constraint是Java Bean Validation规范中的一个注解，用于实现自定义的校验约束。通过该注解，我们可以在Bean Validation中定义自己的校验注解，从而实现更加灵活、精确的数据校验。

### 作用

实现自定义的数据校验，通过该注解，我们可以通过编写校验类和约束注解来实现对特定数据类型的校验。

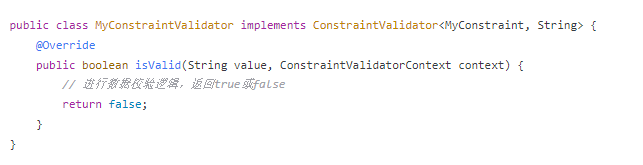
### 使用方法

要使用@Constraint注解，需要定义两个部分：自定义校验类和注解。自定义校验类实现了ConstraintValidator接口，并重写了其isValid方法，其中isValid方法的参数即为需要校验的对象。约束注解则应用在需要进行数据校验的属性上，其value参数指定了校验类的类型。



在上述代码中，我们定义了一个名为MyConstraint的约束注解，并设置了默认的校验失败提示信息。该注解被应用在方法或字段上，并通过@Constraint注解指定了对应的校验类为MyConstraintValidator.class。

自定义的校验类：



在上述代码中，我们定义了一个名为MyConstraintValidator的校验类，实现了ConstraintValidator接口，并重写了其isValid方法。该方法中实现了自定义的校验逻辑，并返回校验结果。

# @Validated

## 基本概念

用于实现对Controller类和其方法参数的数据校验。通过该注解，我们可以在参数绑定之后对数据进行校验，从而保证数据的正确性和安全性。

## 作用范围

类、方法、参数上。

## 作用

### 1、****方法参数校验****

在方法调用时，校验参数是否符合约束（如 @NotNull, @Size 等）。

### ****2、分组校验****

根据不同的业务场景，启用不同的校验规则组。

### **3、**返回值校验****

校验方法的返回值是否符合约束（需结合 @Validated 在类或方法级别）。

## 属性

### value

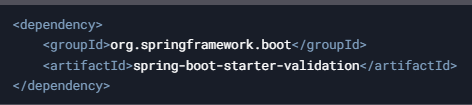
指定一个或多个验证组以应用于由此注释启动的验证步骤。

## 与 @Valid 的区别



## 使用方法

### 1、添加依赖



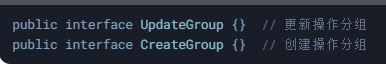
### 2、配置方法校验支持

在Spring Boot中，默认已通过 MethodValidationPostProcessor 支持方法校验。若非Boot项目，需手动配置：

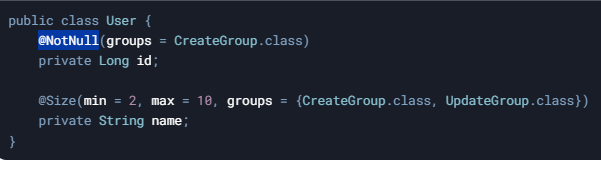


### 3、定义校验分组（可选）

通过接口定义分组



### 4、在实体类中添加校验注解



### 5、在Service层使用 @Validated

类级别：校验类中所有方法的参数和返回值。

方法级别：仅校验当前方法。



## 最佳实践

## 说明

这个注解在spring-context-5.3.31.jar包中

# @Nullable

## 基本概念

@Nullable 注解是 Java 编程中用来标识一个变量、方法参数、返回值等可能为空（null）的注解。它通常用于提高代码的可读性、减少潜在的空指针异常（NullPointerException），并且在代码分析和静态检查时起到提示作用。

## 作用范围

类字段（成员变量）、方法参数、方法返回值

## 作用

@Nullable 注解的作用是告知开发者或静态分析工具，某个元素的值可能是 null。它帮助开发者理解代码的行为，减少空指针异常的发生，同时也是一些现代开发工具和框架的静态分析工具（例如 IDE 或自动化工具）用来执行空值检查的依据。

## 场景

### 1、方法参数

## 说明

这个注解在spring-core-5.3.31.jar包中

# @Value

## 基本概念

用于将外部配置文件中的值注入到Spring管理的Bean中。它可以被用于字段、方法或构造函数上，实现对配置值的注入。

## 代码示例

"${aliyun.sms.expire-minute:15}"：这是一个 SpEL (Spring Expression Language) 表达式，SpEL 是 Spring 框架中用于解析表达式的语言。表达式以 ${...} 的形式包围，并使用冒号分隔默认值。在这个例子中，${aliyun.sms.expire-minute:15} 表示获取名为 aliyun.sms.expire-minute 的属性值，如果该属性不存在，则使用默认值 15。

# @PropertySource

## 基本概念

用于指定外部属性文件的位置，并将属性文件中的属性值加载到Spring的环境中。

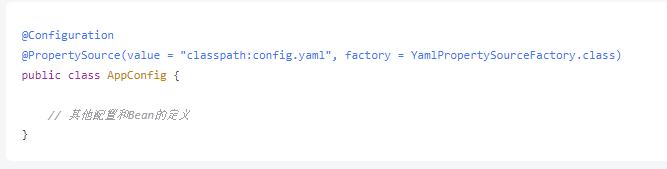
## 属性

### value

### factory

factory属性用于指定属性源的工厂类。属性源工厂类是一个实现了PropertySourceFactory接口的类，用于创建属性源对象。使用@PropertySource注解时，如果属性文件不是标准的.properties文件，而是使用其他格式（如.yaml或.yml）编写的，那么可以通过指定factory属性来告诉Spring如何解析该属性文件。

示例：



在上述代码中，我们使用了@PropertySource注解加载了一个名为config.yaml的属性文件，并通过factory属性指定了YamlPropertySourceFactory作为属性源的工厂类。

factory属性用于指定属性源的工厂类。属性源工厂类是一个实现了PropertySourceFactory接口的类，用于根据不同的属性文件格式创建属性源对象。通过使用自定义的属性源工厂类，我们可以加载和解析非标准的属性文件格式，例如.yaml或.yml文件。

## 使用方法

使用@PropertySource注解很简单，只需在配置类上添加该注解，并指定要加载的属性文件的路径。

代码如下：



在上述代码中，@PropertySource注解的value属性指定了要加载的属性文件的路径为"classpath:config.properties"，这表示属性文件config.properties位于类路径下。接下来我们可以通过@Value注解将属性文件中的属性值注入到相应的字段或方法参数中。

代码如下：



在上述代码中，我们使用@Value注解将属性文件中的db.url属性的值注入到dbUrl字段中。另外还可以将@PropertySource注解与@ConfigurationProperties注解一起使用，实现更灵活的属性配置。

## 总结

@PropertySource注解用于指定外部属性文件的位置，并将属性文件中的属性值加载到Spring的环境中。通过这种方式我们可以方便地在应用程序中使用外部属性文件中的配置信息。

# @ConfigurationProperties

## 基本概念

用于实现对配置文件的自动绑定。通过该注解，我们可以将配置文件中的属性值映射到Java类的属性上，从而方便地读取和管理配置信息。

## 作用范围

类和方法上

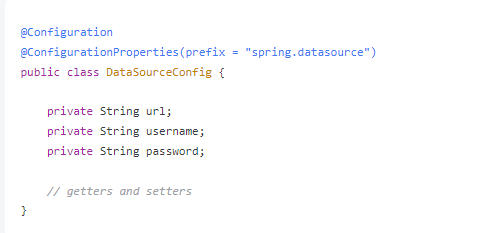
## 作用

使代码更加健壮和易于维护，通过自动绑定配置文件，避免手动读取和解析配置文件的麻烦和错误，提高开发效率和代码质量。

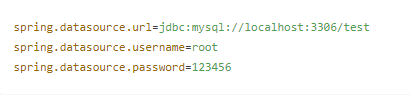
## 使用方法

要使用@ConfigurationProperties注解，只需在Java类上

添加该注解，并指定对应的配置文件前缀即可。



在上述代码中，我们在DataSourceConfig类中定义了三个私有属性url、username、password，并在类上标记了@ConfigurationProperties注解，并指定了前缀为"spring.datasource"，表示将这三个属性与配置文件中前缀为"spring.datasource"的属性进行自动绑定。当配置文件中有如下定义时，Spring会自动将其映射到DataSourceConfig类的属性中：



## 注意事项

1、@ConfigurationProperties注解只能应用于Java类上，并必须与@Configuration注解一起使用。

2、在使用@ConfigurationProperties注解时，需要注意配置文件前缀和Java类中的属性名称要对应，否则会出现自动绑定失败的问题

3、可以将@ConfigurationProperties注解作为属性标注在类上，也可以作为方法标注在FactoryBean实现类方法上，从而实现复杂加工。

# @EnableConfigurationProperties

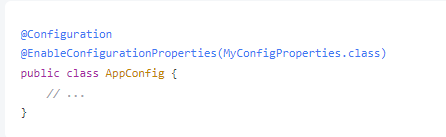
## 基本概念

用于启用对@ConfigurationProperties注解类的支持。它可以被用于@Configuration类上，用于开启对特定配置属性类的支持。

1、启用@ConfigurationProperties支持

通过在@Configuration类上使用@EnableConfigurationProperties注解，可以启用对特定配置属性类的支持，使得这些配置属性类可以被自动注入到Spring管理的Bean中。

示例：



## 作用范围

类上

## 作用

1、让使用了@ConfigurationProperties注解的类生效。

2、将该类注入到 IOC 容器中,交由 IOC 容器进行管理。

# @RefreshScope

## 基本概念

@RefreshScope 是 Spring Cloud 中的一个注解，用于标记需要在运行时刷新的 Bean。它的主要作用是使被注解的 Bean 可以在配置信息发生变化时重新初始化，以便及时获取最新的配置值。

## 作用

### 1、实现动态配置更新

当应用程序中的配置信息（如属性文件、配置中心的配置）发生变化时，使用 @RefreshScope 注解标记的 Bean 可以自动刷新，重新加载最新的配置值，而不需要重启应用程序。

### 2、提供配置信息的实时变更

通过结合 Spring Cloud Config Server 或其他分布式配置中心，可以将配置信息集中管理，并在配置发生变化时自动通知应用程序。被 @RefreshScope 注解标记的 Bean 可以在配置更新后立即获取到最新的配置值。

### 3、动态更新应用程序状态

某些情况下，配置的变化可能会引起应用程序的状态或行为发生变化。使用 @RefreshScope 注解可以实现对应用程序状态的动态更新，以适应配置变化带来的需求调整。

## 总结

@RefreshScope注解是Spring Cloud 中用于标记需要在运行时刷新的 Bean 的注解。它实现了动态配置更新和实时配置变更的功能，使得应用程序可以根据最新的配置信息进行动态调整。通过添加该注解，在配置信息发生变化时，被标记的 Bean 可以自动刷新，并重新初始化以获取最新的配置值。这对于使用分布式配置中心管理配置、实现应用程序状态的动态更新非常有用。

# @EnableDiscoveryClient

能够让注册中心发现，扫描到该服务。从Spring Cloud Edgware开始，@EnableDiscoveryClient 或@EnableEurekaClient **可省略**，只需加上相关依赖，并进行相应配置，即可将微服务注册到服务发现组件上。

# @EnableFeignClients

## 基本概念

@EnableFeignClients 是 Spring Cloud Feign 中的一个注解，它的作用是启用 Feign 客户端功能。

## 作用

1、启用 Feign 客户端：通过在 Spring Boot 应用程序的配置类上添加 @EnableFeignClients 注解，可以启用 Feign 客户端功能。Feign 客户端可以简化服务之间的 HTTP 调用，使得开发者可以像调用本地方法一样方便地调用其他微服务的接口。

2、扫描 Feign 客户端接口：@EnableFeignClients 注解会扫描指定的包路径下的 Feign 客户端接口，并生成相应的代理对象。这些代理对象通过 Feign 的机制来处理 HTTP 请求，并将其转化为对目标服务的调用。

3、配置 Feign 客户端：通过 @EnableFeignClients 注解，可以配置 Feign 客户端的相关属性。例如，可以指定 Feign 客户端的请求超时时间、连接超时时间等参数。

## 总结

@EnableFeignClients 注解的作用是启用 Spring Cloud Feign 客户端功能，并在指定的包路径下扫描 Feign 客户端接口，生成相应的代理对象。这样可以方便地进行服务之间的 HTTP 调用，并通过配置来设置 Feign 客户端的参数。

# @EnableAspectJAutoProxy

## 基本概念

@EnableAspectJAutoProxy 是一个在 Spring 框架中开启 AOP （面向切面编程）的注解。使用该注解可以开启自动代理功能，使得 Spring 框架能够自动将带有 @Aspect 注解的类作为切面，并生成代理对象。通常情况下，我们在使用 Spring AOP 的时候，需要在 ApplicationContext 中配置一个 AspectJAutoProxyCreator 的 bean，以便在运行时创建通知和切面等对象。但是使用@EnableAspectJAutoProxy 注解后，就不需要手动配置这个bean 了。

除了简化配置之外，@EnableAspectJAutoProxy 注解还可以指定其他参数，例如：

proxyTargetClass：是否使用 CGLIB 代理，默认为 false，表示使用 JDK 动态代理。

exposeProxy：是否公开当前代理对象，默认为 false，表示不公开。

## 代码示例

例如以下代码展示了如何使用 @EnableAspectJAutoProxy 注解来开启 Spring AOP 的自动代理功能。



使用该注解后，Spring 框架就可以自动将带有 @Aspect 注解的切面类作为代理对象，并通过代理对象对应用程序进行通知和增强。

# @EnableWebSecurity

## 基本概念

用于启用 Spring Security 的功能

## 作用

### 1、启用 Spring Security 配置

@EnableWebSecurity 注解告诉 Spring 框架启用 Web 安全性，并加载 Spring Security 的配置。具体来说，它会引导 Spring Security 去寻找并应用 Web 安全相关的配置类。

如果你没有显式定义任何 WebSecurityConfigurerAdapter 或类似的安全配置，Spring Security 会提供一个默认的配置，这通常包括：

（1）对所有 URL 的访问都需要身份验证。

（2）默认提供一个内存中的用户（用户名为 user，密码在启动时打印在控制台中）。

（3）提供一个基本的登录页面。

### 2、与 @Configuration 一起使用

@EnableWebSecurity 本质上是一个组合注解，包含了 @Configuration 注解，这意味着它也是一个配置类的标识。通常，带有 @EnableWebSecurity 的类会定义 Spring Security 的相关配置。



### 3、引入 Spring Security 的核心组件

@EnableWebSecurity 注解会自动引入与 Spring Security 相关的一些核心组件，常见的包括：

**WebSecurityConfigurer** 接口及其实现类，用于自定义安全配置。

**DelegatingFilterProxy**，它是一个用于集成 Spring Security 的核心过滤器。

**AuthenticationManager 和 SecurityContext**，用于身份验证和上下文管理。

### 4、替代 SpringSecurityFilterChain 的 XML 配置

在 Spring Security 早期版本中，安全配置通常通过 XML 文件完成，例如：



@EnableWebSecurity 的引入使得开发者能够通过 Java 配置替代 XML 配置，以实现更灵活、更强大的功能。

### 5、与 @SpringBootApplication 的默认安全配置整合

在 Spring Boot 中，如果你不定义任何自定义的安全配置类，Spring Boot 会自动为你启用一个默认的安全配置。但一旦你添加了带有 @EnableWebSecurity 的配置类，这些默认配置会被覆盖。

## 底层实现原理

1、注解驱动 @EnableWebSecurity 会引入一个名为 SpringWebSecurityConfiguration 的配置类，这是 Spring Security 配置的入口。

2、自动配置 在 Spring 的启动过程中，@EnableWebSecurity 会触发 WebSecurityConfiguration 的加载，这个类会解析所有实现了 WebSecurityConfigurer 接口的 Bean（包括 WebSecurityConfigurerAdapter 的子类），并将它们合并到一个全局的安全过滤器链中。

3、生成 Filter Chain 它最终会生成一个 FilterChainProxy，这是 Spring Security 的核心过滤器链，负责拦截所有 HTTP 请求并执行相应的安全逻辑。

## 示例



## 注意事项

1、自定义配置优先 如果存在多个实现了 WebSecurityConfigurer 的类，Spring Security 会根据配置的顺序来决定优先级。

2、版本差异 在 Spring Security 5.7 开始，WebSecurityConfigurerAdapter 已被标记为过时，推荐使用 SecurityFilterChain 来定义安全配置。

示例：



# @EnableGlobalMethodSecurity

# @RequestPart

## 基本概念

## 作用

Spring中用来处理multipart/form-data 类型的请求的注解，它用于将请求中的文件和表单数据封装成对象。

# @PostConstruct

## 基本概念

@PostConstruct 是一个Java注解，它用于标记一个方法在构造函数执行之后、依赖注入完成之后调用。被@PostConstruct注解标记的方法将在对象创建和初始化阶段的最后被自动调用。

## 作用

## 使用场景

## 依赖

# @PreDestroy

基本概念

# @IntegrationComponentScan

## 基本概念

@IntegrationComponentScan注解是Spring Integration框架提供的一个注解，用于扫描和注册Spring Integration组件。

## 作用

### 1、自动扫描

@IntegrationComponentScan注解用于自动扫描指定包及其子包中的类，查找并注册Spring Integration组件。它可以扫描到具有特定注解的类，如@MessagingGateway、@ServiceActivator、@Transformer等，或者实现了特定接口的类。

### 2、注册组件

扫描到的类会被自动注册为Spring容器中的组件，可以通过类名或自定义名称进行引用。这些组件包括消息处理器、消息网关、通道适配器、转换器等，用于构建消息驱动的应用程序。

### 3、简化配置

使用@IntegrationComponentScan注解可以简化Spring Integration配置的过程。传统的配置方式需要显式地在XML或Java配置类中声明每个组件的配置信息，而使用该注解后，只需将相关类放置在指定的包路径中，即可自动注册为组件，减少了繁琐的配置工作。

### 4、组件之间的关联

@IntegrationComponentScan注解可以自动识别并构建组件之间的关联关系。例如，在扫描到的类中，如果存在消息处理器与消息通道的关联，那么自动创建这些消息通道并将相应的消息处理器连接到对应的通道上，简化了组件之间的协作配置。

## 总结

@IntegrationComponentScan注解的作用是在Spring Integration中实现自动扫描和注册组件，简化配置过程，并能够自动构建组件之间的关联关系。通过使用该注解，可以快速搭建消息驱动的应用程序。

# @ServiceActivator

# @Profile

# @CrossOrigin

## 基本概念

@CrossOrigin 注解用于处理 **跨域请求**（Cross-Origin Resource Sharing, CORS）。跨域请求是指浏览器从一个域名向另一个域名发起请求，通常会被浏览器的同源策略（Same-Origin Policy）限制。@CrossOrigin 注解可以轻松配置允许跨域请求的规则。

## 作用

@CrossOrigin 注解用于在 Spring 中配置 CORS 规则，允许指定的跨域请求。它可以应用于：

1、类级别：对整个控制器生效。

2、方法级别：对单个方法生效。

## 属性

### origins

允许的源（域名），支持多个域名，用逗号分隔。

### allowedHeaders

允许的请求头，支持多个请求头，用逗号分隔。

### methods

允许的 HTTP 方法（如 GET、POST），支持多个方法，用逗号分隔。

### exposedHeaders

允许暴露给客户端的响应头，支持多个响应头，用逗号分隔。

### allowCredentials

是否允许携带凭证（如 Cookie、Authorization 头）。

### maxAge

预检请求（OPTIONS）的缓存时间（秒）。

## 说明

这个注解在spring-web-5.3.31.jar包中

# 异步的注解

## @EnableAsync

### 基本概念

用于启用异步方法调用的支持。

### 作用

#### 1、异步方法调用

通常情况下，Spring 中的方法调用是同步的，即方法的调用会阻塞当前线程直到方法执行完成。而异步方法调用则是指在方法调用后立即返回，并在另外的线程中进行方法的执行，这样可以提高系统的性能和响应性，特别适用于处理耗时的操作。

这个注解用于在 Spring 应用程序中启用异步方法调用的支持。当我们在一个配置类或者在 XML 配置中使用 @EnableAsync 注解时，Spring 将会创建一个专用的线程池来处理异步方法，从而实现异步调用的功能。

#### 2、配置异步方法

在使用 @EnableAsync 注解后，我们还需要为希望异步执行的方法进行配置。可以通过在方法上添加 @Async 注解来标识该方法为异步方法。在调用这些方法时，Spring 会将其放入线程池进行异步执行，而不会阻塞当前线程。

#### 3、线程池配置

使用 @EnableAsync 注解后，Spring 默认会创建一个简单的线程池来处理异步方法调用。但是我们也可以通过配置文件或者自定义 TaskExecutor Bean 来自定义线程池的行为，包括线程数量、队列大小、拒绝策略等。

#### 4、返回结果和异常处理

异步方法可以返回 Future 或 CompletableFuture 对象，用于获取最终的执行结果。另外异步方法中抛出的异常可以通过 Future 或 CompletableFuture 进行异常处理。

### 属性

#### proxyTargetClass

用于指定是否使用基于类的代理（CGLIB）来创建异步方法的代理对象。在 Spring 中，默认情况下，使用基于接口的代理（JDK 动态代理）来创建异步方法的代理对象。这意味着被 @Async 注解修饰的方法必须定义在接口中，并且通过接口进行调用。但是，如果目标对象没有实现任何接口，或者你希望使用基于类的代理来创建异步方法的代理对象，就可以使用 proxyTargetClass 属性。

1、默认值：proxyTargetClass 属性的默认值为 false，即默认使用基于接口的代理（JDK 动态代理）。这是因为 JDK 动态代理是基于接口的，能够灵活地创建代理对象并实现方法调用。

2、使用基于类的代理：当将 proxyTargetClass 属性设置为 true 时，Spring 将使用基于类的代理（CGLIB）来创建异步方法的代理对象。CGLIB 是一个强大的代码生成库，允许对目标类生成子类来实现代理。这样，就可以在目标类没有实现接口的情况下，仍然能够创建异步方法的代理对象。

### 注意事项

1、使用基于类的代理时，目标类必须是非 final 的，因为 CGLIB 无法动态生成 final 类的子类。

2、基于类的代理相比基于接口的代理，通常会带来一些性能损耗，并且生成的代理类可能会更复杂。因此，在没有必要使用基于类的代理时，推荐使用默认的基于接口的代理方式。

总结一下，proxyTargetClass 属性用于指定 @EnableAsync 注解在创建异步方法的代理对象时是否使用基于类的代理（CGLIB）。默认情况下使用基于接口的代理，但在没有实现接口的场景下或有特定需求时，可以将 proxyTargetClass 设置为 true，以使用基于类的代理。

### 总结

总结一下，@EnableAsync 注解的作用是启用 Spring 应用程序中的异步方法调用支持。通过在方法上添加 @Async 注解，我们可以将方法调用转变为异步执行，提高系统的性能和响应性。同时，我们还可以通过配置文件或自定义 TaskExecutor Bean 来自定义线程池的行为。

## @Async

# @Order

# @SpringBootTest

## 基本概念

@SpringBootTest 是 Spring Boot 提供的一个注解，用于在测试类上标记以启动整个 Spring 上下文。它通常用于集成测试，确保在测试中可以使用 Spring 的所有功能。

## 作用

### 1、加载 Spring 上下文

@SpringBootTest 会加载完整的 Spring 上下文，包括所有的 Spring Bean。这使得测试可以与 Spring 的依赖注入、配置等功能进行交互。

### 2、支持 Web 环境

可以通过 webEnvironment 属性指定测试的 Web 环境类型，如 WebEnvironment.MOCK 或 WebEnvironment.RANDOM\_PORT。这对于测试 Web 应用程序时非常有用。

### 3、自动配置

在测试中，它会启用 Spring Boot 的自动配置功能，确保必要的组件（如数据源、Web 服务器等）能被正确配置。

### 4、集成其他测试框架

可以与 JUnit 和其他测试框架无缝集成，支持编写单元测试和集成测试。

## 属性

### String[] args()

要传递给待测试应用程序的应用程序参数。

### Class<?>[] classes()

用于加载应用程序上下文的组件类

### String[] properties()

添加的属性

### String[] value()

添加的属性

### WebEnvironment webEnvironment()

web环境的类型

## 基本用法



## 示例代码

以下是一个带有 Web 环境和自定义属性的示例：



## 注意事项

针对SpringBoot的测试类，2.2版本之前和2.2版本之后是不一样的，在2.2版本之前需要贴注解@SpringBootTest和@RunWith(SpringRunner.class)需要在Spring容器环境下进行测试，因为@Test导包的是org.junit.Test,而 在2.2版本之后只需要贴注解@SpringBootTest,@Test导包为org.junit.jupiter.api.Test。

## 总结

1、@SpringBootTest 是一个强大的注解，适用于需要加载整个 Spring 上下文的测试场景。

2、它提供了多种配置选项，以适应不同的测试需求，尤其是在进行集成测试时。

3、使用 @SpringBootTest 可以帮助开发者确保应用程序的各个部分在一起正常工作，并验证整体的功能性。

# @Test（Junit4）

## 基本概念

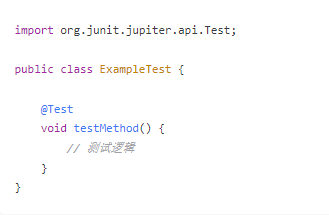
# @Test（JUnit5）

## 基本概念

@Test 注解是来自 JUnit 5（即 JUnit Jupiter）的一部分，用于标识一个方法是一个测试用例。它在单元测试中扮演着重要的角色。

## 标识测试方法

功能: 使用 @Test 注解的方法会被 JUnit 框架识别为测试方法。在执行测试时，JUnit 会找到所有带有 @Test 注解的方法并运行它们。



## 运行测试

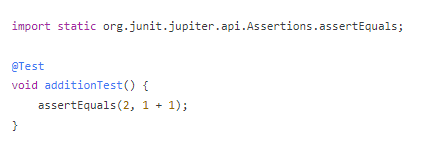
运行机制: 当你运行包含 @Test 注解的方法的测试类时，JUnit 会自动调用这些方法。JUnit 会创建一个新的测试实例（通常是一个新的对象），然后执行每个测试方法。

## 无参数和无返回值

**方法签名**: 被 @Test 注解的方法不应该有任何参数，也不应该返回任何值（通常是 void）。JUnit 只关心方法的执行结果。

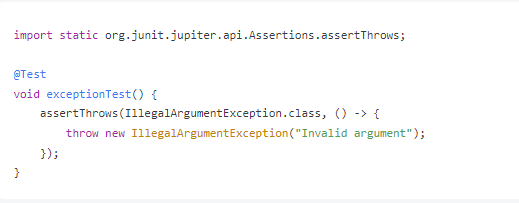
## 断言

验证结果: 在测试方法中，你通常会使用断言（Assertions）来验证代码的行为是否符合预期。例如：



## 异常测试

预期异常: 如果你希望测试一个方法抛出特定的异常，可以结合 assertThrows 方法：

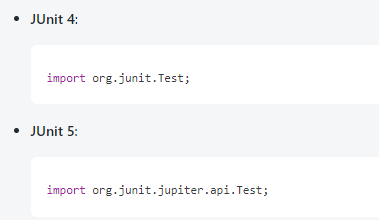


## 总结

@Test 注解是 JUnit 5 中用于标识测试方法的核心元素，它使得编写和运行单元测试变得简单和高效。通过与其他功能（如断言、异常处理和生命周期管理）结合使用，可以构建全面的测试用例，从而确保代码的正确性和健壮性。

## @Test注解在Junit4和Junit5中的区别

### 1、包和导入



### 2、结构和模块化

1、JUnit 4: 在 JUnit 4 中，所有的测试都是在同一个包中定义的，包含了核心功能。

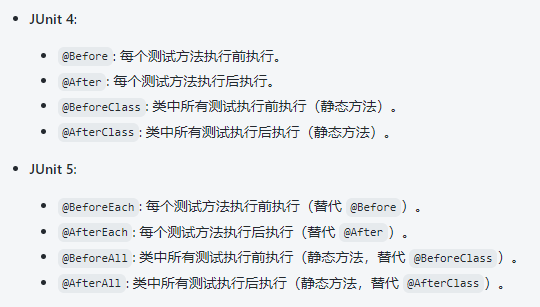
2、JUnit 5: JUnit 5 引入了模块化的设计，主要分为三个部分：

JUnit Platform: 提供运行测试的基础。

JUnit Jupiter: 提供新的编程模型和扩展模型，包括 @Test 注解。

JUnit Vintage: 允许运行 JUnit 4 和之前版本的测试。

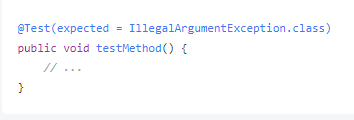
### 3、测试生命周期的注解



### 4、异常处理

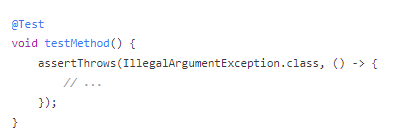
#### 1、Junit4

通过 expected 属性来预期异常



#### 2、JUnit 5

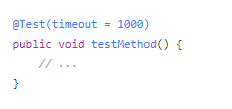
使用 assertThrows 方法来捕获和验证异常



### 5、超时限制

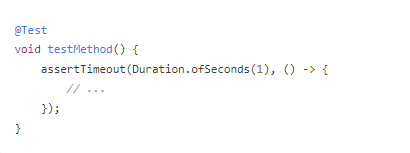
#### 1、Junit4

使用 timeout 属性



#### 2、Junit5

使用 assertTimeout 或 assertTimeoutPreemptively 来设定超时：



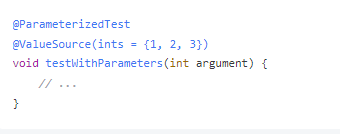
### 6、参数化测试

1、Junit4

使用 @RunWith(Parameterized.class) 来实现参数化测试，配置相对复杂。

2、Junit5

使用 @ParameterizedTest 注解，配置更简洁且功能丰富



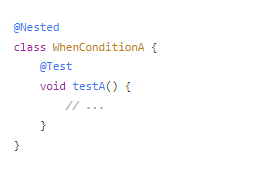
### 7、嵌套测试

1、Junit4

不支持嵌套测试类。

2、Junit5

支持嵌套测试类，通过 @Nested 注解可以实现更清晰的结构



### 8、扩展模型

1、Junit4

使用 @Rule 和 @ClassRule 机制来实现扩展。

2、Junit5

引入了更强大的扩展机制，可以通过实现 Extension 接口来创建自定义扩展。

### 总结

虽然 @Test 注解的基本功能在 JUnit 4 和 JUnit 5 中是类似的，但 JUnit 5 提供了更灵活、更现代的编程模型，增强了对测试的支持，提升了可读性和可维护性。

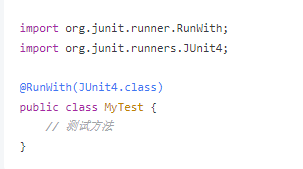
# @RunWith

## 基本概念

@RunWith 注解是 JUnit 4 中的重要特性，用于指定一个自定义的测试运行器（Test Runner）。测试运行器负责控制测试用例的执行流程、收集结果并生成报告。

在 JUnit 中，测试运行器是用来执行测试的核心组件。它决定了如何发现和运行测试，如何处理测试结果等。

## 基本语法

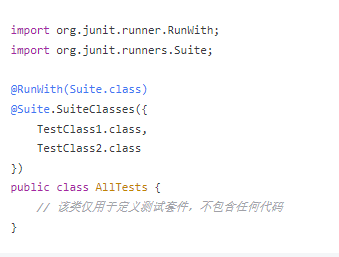


## 常见的测试运行器

JUnit4.class: 默认的 JUnit 4 运行器，通常不需要显式声明。

Parameterized.class: 用于参数化测试，让你可以为同一个测试方法提供多组输入数据。

Suite.class: 用于将多个测试类组合成一个测试套件，便于一起运行。



## 配合其他注解使用

@RunWith 通常与其他 JUnit 注解一起使用，例如 @Test、@Before、@After 等，来控制测试的行为和生命周期。

## 注意事项

JUnit 5 与 JUnit 4: JUnit 5 不再使用 @RunWith，取而代之的是 @ExtendWith。因此如果你使用的是 JUnit 5，请参考其相应的扩展机制。

## 总结

@RunWith 注解是 JUnit 4 中用于指定自定义测试运行器的重要工具。它使得用户能够灵活地控制测试的执行方式，支持参数化测试和测试套件的创建，同时也允许开发者实现更复杂的测试逻辑。

# 多数据源

## @DS

### 基本概念

这个注解在dynamic-datasource-spring-boot-starter-3.5.0.jar包中，@DS 注解的作用是指示应用程序在特定方法或类中使用哪个数据源。通过在方法或类级别上添加 @DS 注解，可以动态切换应用程序的数据源，以便在运行时决定使用哪个数据源。例如假设我们有两个数据源：datasource1 和 datasource2。通过在方法上添加 @DS("datasource1") 注解，我们可以告诉应用程序在调用带有该注解的方法时使用 datasource1 数据源。同样，通过在另一个方法上添加 @DS("datasource2") 注解，我们可以告诉应用程序在调用该方法时使用 datasource2 数据源。通过使用 @DS 注解，我们可以根据不同的业务需求和条件动态地选择使用不同的数据源，从而实现灵活的数据源管理和路由。

### 作用范围

类或者方法上

### 属性

#### value

数据源名称

# Jackson注解

## @JsonProperty

指定属性在序列化和反序列化过程中的名称。

## @JsonIgnore

### 基本概念

@JsonIgnore 注解是 Jackson 序列化和反序列化库中的一个注解，用于指定在 JSON 序列化或反序列化过程中忽略某个字段或方法。

### 作用

#### 1、****序列化时忽略字段****

当对象被转换为 JSON 时，标记为 @JsonIgnore 的字段将不会被包含在生成的 JSON 中。这对于避免在 JSON 输出中包含敏感数据或不需要的字段非常有用。

#### **2、**反序列化时忽略字段****

在将 JSON 数据反序列化为 Java 对象时，标记为 @JsonIgnore 的字段将不会被填充。这意味着即使 JSON 数据中包含该字段，Jackson 也不会将其值赋给对象的对应字段。

### 使用示例

示例 Java 类





@JsonIgnore注解：在User类中，@JsonIgnore注解应用于password字段。这意味着：

在序列化过程中，password字段不会包含在生成的JSON字符串中。

在反序列化过程中，即使 JSON 数据中包含 password 字段的值，该值也不会被赋给对象的 password 字段。

**序列化**：当你调用 mapper.writeValueAsString(user) 时，User 对象被转换为 JSON 字符串。结果字符串将不包含 password 字段，如下所示：

{"name":"Alice","age":30}

**反序列化**：当你将 JSON 字符串反序列化为 User 对象时，password 字段被忽略。即使 JSON 字符串中有 password 字段，其值也不会被设置到 User 对象的 password 字段。

### 总结

@JsonIgnore 是一个强大的工具，可以帮助你控制 JSON 数据的序列化和反序列化过程，确保某些字段在 JSON 表示中被忽略。这对于保护敏感数据和简化 JSON 表示非常有用。

## @JsonIgnoreProperties

### 基本概念

用于在序列化（将对象转换为 JSON）和反序列化（将 JSON 转换为对象）过程中忽略指定的属性。

在 Java 类与 JSON 之间的相互转换中，常常会遇到某些字段不需要进行序列化或反序列化的情况。例如某个字段包含敏感信息或者与业务逻辑无关，可以选择忽略这些字段，不进行序列化或反序列化操作。使用 @JsonIgnoreProperties 注解后，Jackson 库在进行序列化和反序列化时，会忽略指定的属性，不进行处理。从而在生成的 JSON 或者转换回的对象中忽略了被标记的属性。这个注解在实际开发中广泛应用，可以灵活地控制序列化和反序列化的行为，使得生成的 JSON 数据或者转换回的对象符合预期的需求。

### 属性

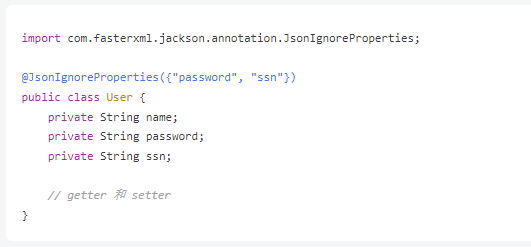
#### ignoreUnknown

用于在反序列化过程中忽略未知的属性。

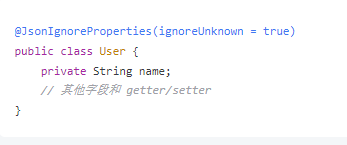
在进行 JSON 反序列化时，有时候源 JSON 数据中可能包含目标类中没有对应的属性。默认情况下当遇到未知属性时，Jackson 库会抛出异常。然而，有些情况下我们可能希望忽略这些未知属性，并继续进行反序列化操作。通过设置 ignoreUnknown = true，可以使 @JsonIgnoreProperties 注解在反序列化过程中忽略未知的属性。这样即使源 JSON 数据中包含了目标类中没有的属性，Jackson 也不会抛出异常，而是简单地忽略这些未知属性。

### 示例

#### 1、忽略多个字段



#### 2、忽略未知属性



### 应用场景

当你希望在类级别上统一管理多个字段的忽略行为时，@JsonIgnoreProperties 是合适的选择。如果你希望忽略 JSON 数据中的未知属性，也可以使用这个注解。

## @JsonIgnore和@JsonIgnoreProperties的区别

1、@JsonIgnore: 用于忽略单个字段或属性，在序列化和反序列化时都有效，优先级高于类级别的忽略。

2、@JsonIgnoreProperties: 用于类级别，支持批量忽略字段和处理未知属性，优先级低于字段级别的 @JsonIgnore。

## @JsonInclude

### 基本概念

控制在序列化过程中包含哪些属性。

### 作用范围

类或者字段上。

### 作用

#### 1、选择性序列化

@JsonInclude 注解可以用来控制哪些字段在序列化时应该被包含在 JSON 输出中，哪些字段应该被忽略。

#### 2、减少JSON输出的体积

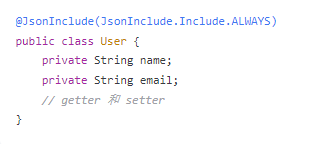
通过配置 @JsonInclude，可以确保只有满足特定条件的字段被序列化，减少 JSON 数据的体积和复杂度。

### 使用方式

@JsonInclude 注解可以用在类级别或者字段级别上。它的配置选项主要有以下几种：

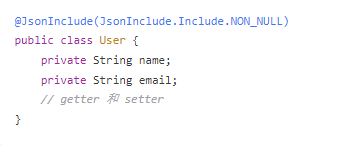
#### 1、JsonInclude.Include.ALWAYS

默认选项。所有字段都会被包括在序列化结果中，不论字段的值是什么。



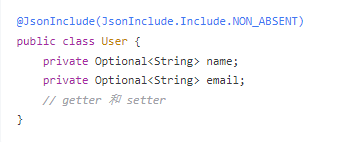
#### 2、JsonInclude.Include.NON\_NULL

仅序列化非 null 的字段。如果字段的值为 null，该字段将不会出现在 JSON 输出中。



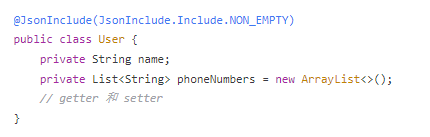
#### 3、JsonInclude.Include.NON\_ABSENT

仅序列化非“缺失”的字段。在 Java 8 中，对于 Optional 类型，这个选项意味着只有 Optional 包含值时才序列化字段。如果是 Optional.empty()，则不会序列化。



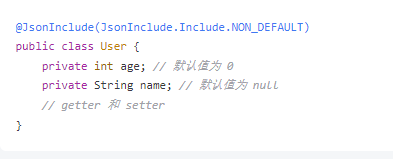
#### 4、JsonInclude.Include.NON\_EMPTY

仅序列化非空的字段。对于 String 类型来说，空字符串 ("") 会被排除；对于集合和数组类型来说，空集合和空数组也会被排除。

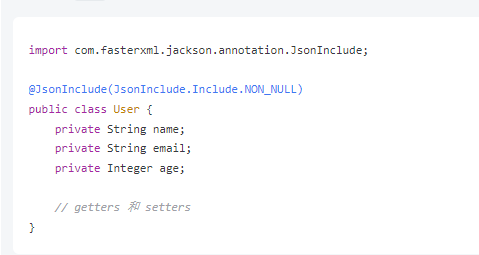


#### 5、JsonInclude.Include.NON\_DEFAULT

仅序列化与默认值不同的字段。对于基本数据类型，默认值为 0、false 或 null；对于对象类型，默认值为 null。



### 示例



如果 User 对象的 email 字段为 null，则在序列化为 JSON 时，email 字段将不会出现在 JSON 输出中。假设对象的 name 为 "Alice"，age 为 30，则序列化后的 JSON 结果将会是：



### 总结

1、@JsonInclude 注解用于控制序列化时字段的包含策略，减少不必要的数据输出。

2、通过配置不同的 JsonInclude.Include 选项，你可以灵活地控制 JSON 输出的内容，确保输出结果只包含你关注的信息。

3、使用 @JsonInclude 可以提高数据传输的效率，减少网络带宽的消耗和处理时间。

## @JsonSerialize

## @JsonRawValue

## @JsonCreator

### 基本概念

用于标记一个构造方法、工厂方法或静态工厂方法，指示Jackson在反序列化过程中使用该方法来创建对象。

### 作用

告诉Jackson在将JSON数据转换为Java对象时，使用被注解的方法来完成对象的创建。通常情况下，Jackson会通过调用无参构造方法来创建对象，然后通过setter方法设置属性值。但有时候，我们可能希望使用不同的方式来创建对象，例如通过某些特定的参数值来构造对象，或者使用静态工厂方法来创建对象。

### 代码示例

#### 1、通过构造方法创建对象

如果一个类没有默认的无参构造方法，但有其他有参构造方法，可以使用@JsonCreator注解标记其中一个构造方法，告诉Jackson在反序列化时使用该构造方法来创建对象。

#### 2、通过静态工厂方法创建对象

## @JsonAlias

### 基本概念

@JsonAlias是Jackson库中的一个注解，用于为Java对象的属性指定JSON字段的别名。它允许在反序列化（将JSON转换为Java对象）时，将JSON中不同名称的字段映射到同一个Java属性上。

### 主要作用

#### 1、多名称映射

允许一个Java属性对应JSON中的多个字段名

#### 2、提高兼容性

处理JSON字段名有变化或不同版本的情况

#### 3、简化数据转换

无需修改原始JSON数据就能适应Java对象模型

### 与@JsonProperty的区别

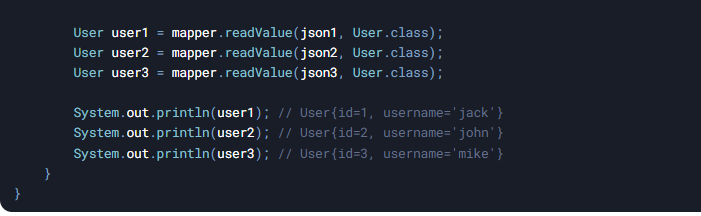
1、@JsonProperty：用于指定属性与JSON字段之间的双向映射（序列化和反序列化都有效）

2、@JsonAlias：仅影响反序列化过程，为属性提供额外的可接受字段名

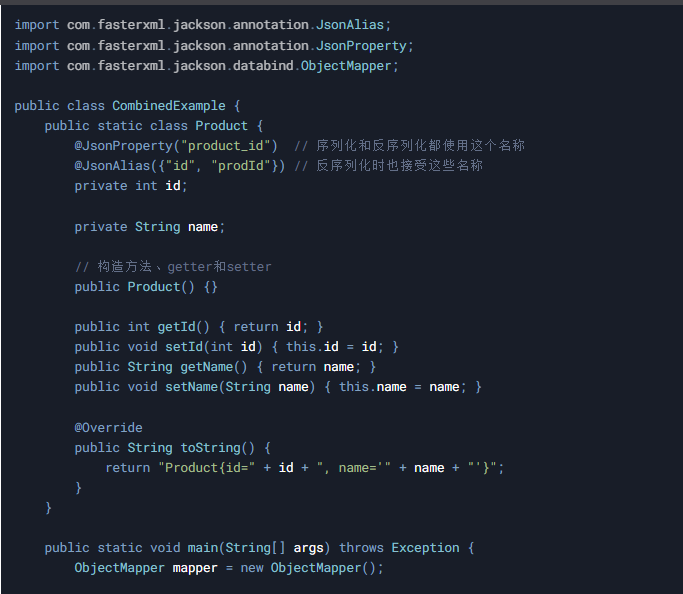
### 示例代码

#### 1、基本用法





#### 2、与@JsonProperty结合使用





### 注意事项

1、@JsonAlias只影响反序列化过程，不影响序列化。

2、如果JSON中有多个别名字段匹配同一个Java属性，最后一个出现的值会覆盖前面的值

3、当同时使用@JsonProperty和@JsonAlias时，@JsonProperty的名称也会被包含在可接受的名称列表中

4、如果没有任何名称匹配，Jackson会尝试使用Java属性的名称进行匹配

5、@JsonAlias在处理不同来源或不同版本的JSON数据时特别有用，可以增加代码的灵活性和兼容性。

## @JsonFormat

### 基本概念

@JsonFormat注解是一个时间格式化注解，比如我们存储在mysql中的数据是date类型，当我们读取出来封装在实体类中的时候，就会变成英文时间格式，而不是yyyy-MM-dd HH:mm:ss这样的中文时间，因此我们需要用到JsonFormat注解来格式化我们的时间。一般用于一个字段命名上面。

### 属性

#### shape

定义如何序列化值（如STRING, NUMBER, OBJECT等）

#### pattern

日期时间格式模式（如"yyyy-MM-dd"）

#### timezone

时区ID（如"GMT+8"或"America/New\_York"）

#### locale

本地化设置（如"zh\_CN"）

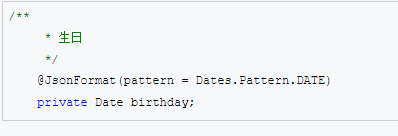
#### with

要启用的特性

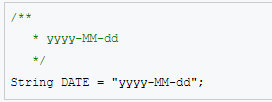
#### without

要禁用的特性

### 使用方法



这里Dates.Pattern.DATE是枚举类型取值，对应值为：



要想使用此注解，需要引入依赖：



jackson-databind

就可以在实体类属性上面使用@JsonFormat注解了，要注意的是，它只会在类似@ResponseBody返回json数据的时候才会返回格式化的yyyy-MM-dd HH:mm:ss时间，你直接使用System.out.println()输出的话，仍然是类似“Fri Dec 01 21:05:20 CST 2017”这样的时间样式。

## @DateTimeFormat和@JsonFormat区别

### 基本说明

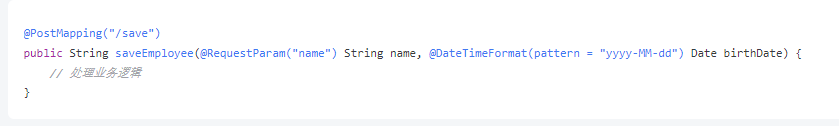
都是用于处理日期时间格式化的问题，但它们的作用对象和使用场景略有不同。

### @DateTimeFormat

1、@DateTimeFormat 是 Spring 框架中用于将 String 类型的日期时间数据转换为 Java 中的 Date 类型的注解。

2、主要用于将前端传递过来的日期字符串（如表单提交的日期）自动转换为后端接收参数的 Date 类型，以便在后续的业务逻辑中进行处理。

3、通常用在 Controller 层的方法参数上，指定日期的格式化方式。



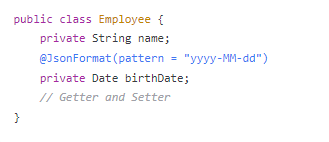
在这个例子中，@DateTimeFormat(pattern = "yyyy-MM-dd") 指定了日期字符串的格式为 "yyyy-MM-dd"，Spring 将自动将传入的日期字符串转换为 Date 类型。

### @JsonFormat

1、@JsonFormat 是 Jackson 库中提供的注解，用于控制序列化和反序列化 JSON 数据时的日期格式。

2、主要用于在将 Java 对象转换为 JSON 或反之时，指定日期类型字段的格式，以便在 JSON 数据中按照指定格式显示日期时间信息。

3、可以用在实体类的日期类型字段上，例如：



4、在这个例子中，@JsonFormat(pattern = "yyyy-MM-dd") 指定了将该字段序列化为 JSON 时的日期格式。

### 总结

总结来说，@DateTimeFormat 主要用于处理前端传递的日期字符串到后端的日期类型参数的转换，而 @JsonFormat 主要用于控制日期类型字段在 JSON 序列化和反序列化过程中的格式化。两者分别在不同的层面上处理日期时间格式化的问题，帮助开发者更方便地处理日期数据。

## @JsonValue

### 基本概念

我们可以使用Jackson库来将Java对象转换为JSON格式的字符串，或者将JSON格式的字符串转换为Java对象。在某些情况下，我们需要对Java对象进行特殊的序列化操作，以便控制输出的JSON格式。这时就可以使用Jackson库提供的注解来实现这个功能，其中包括@JsonValue注解。

@JsonValue注解的作用是指定一个方法或字段作为Java对象的默认序列化值。如果使用该注解标注一个非静态的方法（非构造方法），则该方法的返回值会被序列化为JSON字符串；如果使用该注解标注一个字段，则该字段的值会被序列化为JSON字符串。当Java对象被序列化为JSON字符串时，Jackson库会自动调用被@JsonValue注解标注的方法或字段，并将其返回值作为JSON格式的字符串表示。

### 说明

这个注解在jackson-annotations-2.13.5.jar包中

### 作用范围

用在方法、字段上。

### 作用

它标记了一个字段或方法，该字段或方法的返回值会成为序列化对象时的 JSON 值。

### 使用场景

1、枚举类：在枚举类中，你可以通过 @JsonValue 指定枚举的某个值作为 JSON 输出值。

2、自定义对象：在某些自定义类中，可以通过 @JsonValue 来指定该对象的一个字段或方法的值作为 JSON 的最终表示。

### 示例

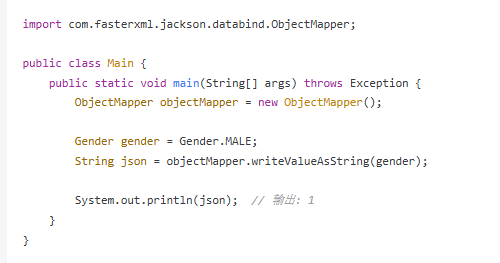
#### 1. 枚举类中的 @JsonValue

当你使用 @JsonValue 注解在枚举类的某个字段上时，Jackson 在序列化枚举类型时，会使用该字段的值。



在上面的代码中，Gender 枚举类中的 getValue 方法使用了 @JsonValue 注解。这样，在将 Gender 枚举对象转换为 JSON 时，1 或 2 将作为 JSON 中的值，而不是枚举名（MALE 或 FEMALE）

**使用 Jackson 序列化时的输出**



输出：1

在序列化 Gender.MALE 时，输出的是 1，因为 @JsonValue 注解标记了 getValue 方法作为 JSON 输出的值。

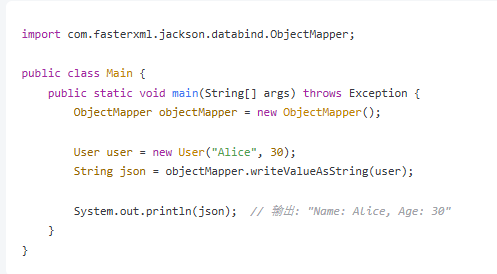
#### 2. 自定义类中的 @JsonValue

可以在自定义的类中使用 @JsonValue，指定一个字段或者方法作为整个类的序列化值。



User 类的 getFullInfo 方法被标记为 @JsonValue，所以当 User 对象序列化时，返回的是 getFullInfo 方法的返回值。

**使用 Jackson 序列化时的输出**



**输出**："Name: Alice, Age: 30"

在序列化 User 对象时，getFullInfo 方法的返回值被作为 JSON 输出。

# Swagger相关的注解

## @Api

### 基本概念

## @Tag

## @ApiOperation

## @ApiModelProperty

### 基本概念

### 作用范围

## @ApiParam

### 基本概念

### 使用范围

## @ApiModelProperty和@ApiParam的区别

### 用途

1、**@ApiParam** 注解主要用于描述 API 接口中的方法参数，用于生成 API 文档时显示参数的相关信息，如参数名称、描述、是否必填等。它通常用于控制器方法的参数上。

2、@ApiModelProperty 注解主要用于描述 POJO（Plain Old Java Object）类中的字段，用于生成 API 文档时显示字段的相关信息，如字段名称、描述、是否必填等。它通常用于实体类的字段上。

### 适用范围

1、@ApiParam 注解适用于方法参数，因此只能在控制器方法的参数上使用。

2、@ApiModelProperty 注解适用于类的字段，因此只能在实体类的字段上使用。

### 功能特性

1、@ApiParam 注解主要用于描述方法参数，以便用户理解该方法的输入参数是什么以及如何使用。它强调的是接口的输入。

2、@ApiModelProperty 注解主要用于描述实体类的字段，以便用户理解该类的结构和含义。它强调的是数据模型的定义和属性。

### 属性差异

1、@ApiParam 注解的属性通常包括 value（描述信息）、name（参数名）、required（是否必填）、defaultValue（默认值）、allowableValues（允许值范围）等。

2、@ApiModelProperty 注解的属性通常包括 value（描述信息）、name（字段名）、required（是否必填）、example（示例值）、hidden（是否隐藏）等。

### 总结

虽然 @ApiParam 和 @ApiModelProperty 注解都是 Swagger 提供的用于描述 API 文档的注解，但它们针对的对象不同，分别用于描述方法参数和实体类的字段。在实际开发中，根据需要选择合适的注解来描述方法参数或字段，以提高 API 文档的可读性和可理解性。

## @ApiImplicitParam

## @ApiImplicitParams