32-2、springboot核心的27个注解

（1）spring的6个注解

（1-1）@Configuration

- spring3.0提供的配置注解，用于替换掉配置bean定义的spring.xml配置文件，被此注解修饰的类，内部如果定义一个或多个@Bean注解的方法，这些方法会被扫描，用于bean实例化，加入到Spring容器。

（1-2）@ComponentScan

- 不指定扫描的包路径时，默认扫描本包及其子包下的@Controller，@Service，@Repository，@Component注解的类，用于实例化，加入到Spring容器。
- @SpringBootApplication注解也包含了@ComponentScan注解，所以在使用中我们也可以通过@SpringBootApplication注解的scanBasePackages属性进行配置扫描的包路径。

（1-3）@Conditional

- 参考：https://blog.csdn.net/lzb348110175/article/details/114384719
- spring4.0提供的条件注解，修饰类和方法。条件注解@Conditional中的value是一个实现了Condition接口的类型的数组，所以value指定的都是实现了Condition接口的类，此类实现了matches方法，当方法返回true，则会实例化条件注解修饰的bean；当方法返回false，则不实例化条件注解修饰的bean。Spring Boot注解中的以@Conditional*开头的注解，都是通过集成了@Conditional来实现相应功能的。

复制代码

```
//此注解可以标注在类和方法上
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Conditional {
    Class<? extends Condition>[] value();
}

```

```java

public interface Condition {
    boolean matches(ConditionContext var1, AnnotatedTypeMetadata var2);
}
```

java

- 从容器中获取两个bean实例

复制代码

```java
@Data
public class Person {
    private String name;
    private Integer age;
}


@Configuration
public class BeanConfig {

    @Bean(name = "bill")
    public Person person1(){
        return new Person("Bill Gates",62);
    }

    @Bean("linus")
    public Person person2(){
        return new Person("Linus",48);
    }
}

public class ConditionalTest {

```

```java
    AnnotationConfigApplicationContext applicationContext
      = new AnnotationConfigApplicationContext(BeanConfig.class);

    @Test
    public void test(){
        Map<String, Person> map
          = applicationContext.getBeansOfType(Person.class);
        System.out.println(map);
    }
}

//结果
{bill=Person{name='Bill Gates',age=62}, linus=Person{name='Linus',age=48}}
```

- 需求：如果想根据当前的操作系统（window，linux），来决定是实例化name是bill的Person对象，还是name是linus的Person对象，这就需要我们用到@Conditional注解了

复制代码

```java
/**
 * 自定义WindowsCondition条件类，实现Condition接口，重写matches方法
 */
public class WindowsCondition implements Condition {

    /**
     * @param conditionContext:判断条件能使用的上下文环境
     * @param annotatedTypeMetadata:注解所在位置的注释信息
     * */
    @Override
    public boolean matches(ConditionContext conditionContext, AnnotatedTypeMetadata annotatedTypeMetadata) {
        //获取ioc使用的beanFactory
```

```
        ConfigurableListableBeanFactory beanFactory = conditionContext.getBeanFactory();
        //获取类加载器
        ClassLoader classLoader = conditionContext.getClassLoader();
        //获取当前环境信息
        Environment environment = conditionContext.getEnvironment();
        //获取bean定义的注册类
        BeanDefinitionRegistry registry = conditionContext.getRegistry();

        //获得当前系统名
        String property = environment.getProperty("os.name");
        //包含Windows则说明是windows系统，返回true
        if (property.contains("Windows")){
            return true;
        }
        return false;
    }
}


/**
 * 自定义LinuxCondition条件类，实现Condition接口，重写matches方法
 */
public class LinuxCondition implements Condition {

    /**
     * @param conditionContext:判断条件能使用的上下文环境
     * @param annotatedTypeMetadata:注解所在位置的注释信息
     * */
    @Override
    public boolean matches(ConditionContext conditionContext, AnnotatedTypeMetadata
annotatedTypeMetadata) {
        //获取ioc使用的beanFactory
```

```
        ConfigurableListableBeanFactory beanFactory = conditionContext.getBeanFactory();
        //获取类加载器
        ClassLoader classLoader = conditionContext.getClassLoader();
        //获取当前环境信息
        Environment environment = conditionContext.getEnvironment();
        //获取bean定义的注册类
        BeanDefinitionRegistry registry = conditionContext.getRegistry();

        //获得当前系统名
        String property = environment.getProperty("os.name");
        //包含Linux则说明是linux系统，返回true
        if (property.contains("Linux")){
            return true;
        }
        return false;
    }
}


//////////////////////////////////////////////////////////////////////


//@Conditional条件注解标注在方法上
@Configuration
public class BeanConfig {

    //只有一个类时，大括号可以省略
    //如果WindowsCondition条件类的实现方法返回true，则注入person1方法的这个bean
    @Conditional({WindowsCondition.class})
    @Bean(name = "bill")
    public Person person1(){
        return new Person("Bill Gates",62);
```

```
    }

    //如果LinuxCondition条件类的实现方法返回true，则注入person2方法的这个bean
    @Conditional({LinuxCondition.class})
    @Bean("linus")
    public Person person2(){
        return new Person("Linus",48);
    }
}


@Test
public void test1(){
  String osName = applicationContext.getEnvironment().getProperty("os.name");
  System.out.println("当前系统为: " + osName);
  Map<String, Person> map = applicationContext.getBeansOfType(Person.class);
  System.out.println(map);
}
//打印结果
window系统中运行:
当前系统为: Windows
{bill=Person{name='Bill Gates',age=62}}

Linux系统中运行:
当前系统为: Linux
{linus=Person{name='Linus',age=48}}

也可以在IDEA中模拟os环境
Edit Configurations→VM options→-Dos.name=Windows/Linux

//结论：根据@Conditional条件注解中，指定的条件类中处理逻辑，来决定条件注解修饰方法是否实例化

```

```
108
109 ////////////////////////////////////////////////////////////////////////
110
111
112 //@Conditional条件注解标注在类上
113 试一下如果改写BeanConfig，将@Conditional标注在BeanConfig类上，如下，如果当前是window环境，则返回
    true，则两个Person实例都会被注入，如果将类上的WindowsCondition.class改为 LinuxCondition.class，则返
    回false，则两个Person实例都不会被注入。
114
115 @Conditional({WindowsCondition.class})
116 @Configuration
117 public class BeanConfig {
118
119     @Bean(name = "bill")
120     public Person person1(){
121         return new Person("Bill Gates",62);
122     }
123
124     @Bean("linus")
125     public Person person2(){
126         return new Person("Linus",48);
127     }
128 }
129
130
131 ////////////////////////////////////////////////////////////////////////
132
133
134 public class ObstinateCondition implements Condition {
135     @Override
136     public boolean matches(ConditionContext conditionContext, AnnotatedTypeMetadata
    annotatedTypeMetadata) {
```

```java
            return false;//写死
        }
    }

//@Conditional条件注解同时使用多个条件类，需要全部为true（即当前环境是window并且ObstinateCondition的matches方法返回true），才能实例化两个Person，注入到容器。
@Conditional({WindowsCondition.class,ObstinateCondition.class})
@Configuration
public class BeanConfig {

    @Bean(name = "bill")
    public Person person1(){
        return new Person("Bill Gates",62);
    }

    @Bean("linus")
    public Person person2(){
        return new Person("Linus",48);
    }
}
```

java

(1-4) @Import

- 通过导入的方式来，把实例对象，加入到spring容器中，可以在需要的时候将没有被spring容器管理的类导入到spring容器中。

```
复制代码
```

```java

//类定义
public class Square {}
public class Circular {}

//导入
@Import({Square.class,Circular.class})
@Configuration
public class MainConfig{}
```

java

(1-5) @ImportResource

- 和@Import一样，不同的是，此注解是导入配置文件的形式，配置文件中是bean的定义

复制代码

```java
@ImportResource("classpath:spring-redis.xml")      //导入xml配置

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

java

(1-6) @Component

- 是一个元注解，可以用来修饰其它注解，还可以修饰的类，被修饰的类会看做组件，注解@ComponentScan进行扫描时，组件就会被扫描到进行实例化。而像@Controller，@Service，@Repository注解去修饰类，本质都是@Component元注解在修饰类，只是因为每层代表的意义不同，所以基于@Component又进行定制了注解。
- 因为@Component修饰的类泛指组件，所以当组件不好归类的时候，我们可以使用这个注解进行标注，作用就相当于 XML中的配置。

(2) springboot的21个注解

(2-1) @SpringBootApplication

- 32-1中已经详细描述了

(2-2) @ConditionalOnBean

复制代码

```java

@Conditional(OnBeanCondition.class)
public @interface ConditionalOnBean
```

- 组合了@Conditional 注解。@ConditionalOnBean(A.class)，代表当前spring上下文（ ApplicatinonContext ）中存在A的实例对象时，才会实例化@ConditionalOnBean修饰的类或方法。
- 底层逻辑就是实现了Condition接口的OnBeanCondition条件类的matches方法中对@ConditionalOnBean的处理，方法返回true，则存在A的实例，方法返回false，则不存在A的实例。

(2-3) @ConditionalOnMissingBean

复制代码

```java
@Conditional(OnBeanCondition.class)
public @interface ConditionalOnMissingBean
```

- @ConditionalOnMissingBean(A.class)，同@ConditionalOnBean相反，代表当前spring上下文（ ApplicatinonContext ）中不存在A的实例对象时，才会实例化@ConditionalOnMissingBean修饰的类或方法。
- 底层逻辑就是实现了Condition接口的OnBeanCondition条件类的matches方法中对@ConditionalOnMissingBean的处理，方法返回true，则代表不存在A的实例，方法返回false，则代表存在A的实例。

(2-4) @ConditionalOnClass

```
1
2 @Conditional(OnClassCondition.class)
3 public @interface ConditionalOnClass
```

复制代码 java

- @ConditionalOnClass(A.class)，当classpath中存在A类时，才会实例化@ConditionalOnClass修饰的Bean。
- 底层逻辑就是实现了Condition接口的OnClassCondition条件类的matches方法中对@ConditionalOnClass的处理，方法返回true，则代表存在A类，方法返回false，则代表不存在A类。

(2-5) @ConditionalOnMissingClass

```
1
2 @Conditional(OnClassCondition.class)
3 public @interface ConditionalOnMissingClass
```

复制代码 java

- @ConditionalOnMissingClass(A.class)，当classpath中不存在A类时，才会实例化@ConditionalOnMissingClass修饰的Bean。
- 底层逻辑就是实现了Condition接口的OnClassCondition条件类的matches方法中对@ConditionalOnMissingClass的处理，方法返回true，则代表不存在A类，方法返回false，则代表存在A类。

(2-6) @ConditionalOnWebApplication

- 当前项目类型是 WEB 项目才开启配置。
- 当前项目有以下 3 种类型
  - ANY(任何Web项目都匹配)
  - SERVLET (仅当是基础的Servlet项目才会匹配)
  - REACTIVE (只有基于响应的web项目才匹配)

(2-7) @ConditionalOnNotWebApplication

复制代码

```java
@Conditional(OnWebApplicationCondition.class)
public @interface ConditionalOnWebApplication

```

java

(2-8) @ConditionalOnProperty

复制代码

```java
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.TYPE, ElementType.METHOD })
@Documented
@Conditional(OnPropertyCondition.class)
public @interface ConditionalOnProperty {

    String[] value() default {}; //name的alias

    String prefix() default "";

    String[] name() default {};

    String havingValue() default "";
}
```

java

- 指定的属性有指定的值时，才会生效。具体操作就是name和havingValue实现的，name用来从application.properties中读取某个属性值，如果该值为空（配置文件中没有做配置），则返回false；如果值不为空，则将该值与havingValue指定的值进行比较，如果一样则返回true；否则返回false。如果返回值为false，则@ConditionalOnProperty修饰的类或方法不生效，为true则生效。

复制代码

```java
@Bean
@ConditionalOnProperty(name = "rocketmq.producer.enabled", havingValue = "true",
matchIfMissing = true)
```

```java
public RocketMQProducer mqProducer() {
 return new RocketMQProducer();
}
```