无

> 1. 概述本文我们来分享 @ConfigurationProperties 注解，如何将配置文件自动设置到被注解的类。

本文我们来分享 @ConfigurationProperties 注解，如何将配置文件自动设置到被注解的类。代码如下：

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface ConfigurationProperties {

    @AliasFor("prefix")
    String value() default "";

    @AliasFor("value")
    String prefix() default "";

    boolean ignoreInvalidFields() default false;

    boolean ignoreUnknownFields() default true;

}
```

@ConfigurationProperties 注解有两种使用方法，可见 《关与 @EnableConfigurationProperties 注解》 文章。总结来说：

- 第一种， @Component + @ConfigurationProperties 。
- 第二种， @EnableConfigurationProperties + ConfigurationProperties 。

实际情况下，更多的是使用第一种。当然，第二种的 @EnableConfigurationProperties 的效果，也是将指定的类，实现和 @Component 被注解的类是一样的，创建成 Bean 对象。这样， @ConfigurationProperties 就可以将配置文件自动设置到该 Bean 对象咧。

org.springframework.boot.context.properties.@EnableConfigurationProperties 注解，可以将指定带有 @ConfigurationProperties 的类，注册成 BeanDefinition ，从而创建成 Bean 对象。代码如下：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(EnableConfigurationPropertiesImportSelector.class)
public @interface EnableConfigurationProperties {
```

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

- 从 `@Import` 注解上，可以看到使用 EnableConfigurationPropertiesImportSelector 处理。详细的解析，见 「2.2 EnableConfigurationPropertiesImportSelector」 。

2.1 ConfigurationPropertiesAutoConfiguration

默认情况下， `@EnableConfigurationProperties` 会通过 org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration 类，进行开启。代码如下：

```
@Configuration
@EnableConfigurationProperties
public class ConfigurationPropertiesAutoConfiguration {
}
```

- 看，看看，看看看， `<X>` 哟~

2.2 EnableConfigurationPropertiesImportSelector

org.springframework.boot.context.properties.EnableConfigurationPropertiesImportSelector ，实现 ImportSelector 接口，处理 `@EnableConfigurationProperties` 注解。代码如下：

```
private static final String[] IMPORTS = {
                ConfigurationPropertiesBeanRegistrar.class.getName(),
                ConfigurationPropertiesBindingPostProcessorRegistrar.class.getName() };

@Override
public String[] selectImports(AnnotationMetadata metadata) {
        return IMPORTS;
}
```

- 返回的 `IMPORTS` 的是两个 ImportBeanDefinitionRegistrar 实现类。分别是：
    - ConfigurationPropertiesBeanRegistrar ，在 「2.3 ConfigurationPropertiesBeanRegistrar」 中详细解析。
    - ConfigurationPropertiesBindingPostProcessorRegistrar ，在 「2.4 ConfigurationPropertiesBindingPostProcessorRegistrar」 中详细解析。

2.3 ConfigurationPropertiesBeanRegistrar

ConfigurationPropertiesBeanRegistrar ，是 EnableConfigurationPropertiesImportSelector 的内部静态类，实现 ImportBeanDefinitionRegistrar 接口，将 `@EnableConfigurationProperties` 注解指定的类，逐个注册成对应的 BeanDefinition 对象。代码如下：

```
@Override
public void registerBeanDefinitions(AnnotationMetadata metadata, BeanDefinitionRegistry registry) {
    getTypes(metadata)
            .forEach((type) -> register(registry, (ConfigurableListableBeanFactory) registry, type));
}
```

- `<1>` 处，调用 `#getTypes(AnnotationMetadata metadata)` 方法，获得 `@EnableConfigurationProperties` 注解指定的类的数组。代码如下：

```
private List<Class<?>> getTypes(AnnotationMetadata metadata) {

    MultiValueMap<String, Object> attributes = metadata.getAllAnnotationAttributes(EnableConfigurationProperties.class.getName(), false);

    return collectClasses((attributes != null) ? attributes.get("value")
            : Collections.emptyList());
}

private List<Class<?>> collectClasses(List<?> values) {
```

```
    return values.stream().flatMap((value) -> Arrays.stream((Object[]) value))
            .map((o) -> (Class<?>) o).filter((type) -> void.class != type)
            .collect(Collectors.toList());
}
```

- ~

- <2> 处，遍历，逐个调用 #register(BeanDefinitionRegistry registry, ConfigurableListableBeanFactory beanFactory, Class<?> type) 方法，注册每个类对应的 BeanDefinition 对象。代码如下：

```
private void register(BeanDefinitionRegistry registry, ConfigurableListableBeanFactory beanFactory, Class<?> type) {

    String name = getName(type);


    if (!containsBeanDefinition(beanFactory, name)) {
        registerBeanDefinition(registry, name, type);
    }
}
```

  - <2.1> 处，调用 #getName(Class<?> type) 方法，通过 @ConfigurationProperties 注解，获得最后要生成的 BeanDefinition 的名字。代码如下：

```
private String getName(Class<?> type) {
    ConfigurationProperties annotation = AnnotationUtils.findAnnotation(type, ConfigurationProperties.class);
    String prefix = (annotation != null) ? annotation.prefix() : "";
    return (StringUtils.hasText(prefix) ? prefix + "-" + type.getName() : type.getName());
}
```

    - 格式为 prefix- 类全名 or 类全名。

  - <2.2> 处，调用 #containsBeanDefinition(ConfigurableListableBeanFactory beanFactory, String name) 方法，判断是否已经有该名字的 BeanDefinition 的名字。代码如下：

```
private boolean containsBeanDefinition(ConfigurableListableBeanFactory beanFactory, String name) {

    if (beanFactory.containsBeanDefinition(name)) {
        return true;
    }

    BeanFactory parent = beanFactory.getParentBeanFactory();
    if (parent instanceof ConfigurableListableBeanFactory) {
        return containsBeanDefinition((ConfigurableListableBeanFactory) parent, name);
    }
    return false;
}
```

    - 如果不存在，才执行后续的注册 BeanDefinition 逻辑。

  - <2.3> 处，调用 #registerBeanDefinition(BeanDefinitionRegistry registry, String name, Class<?> type) 方法，注册 BeanDefinition 。代码如下：

```
private void registerBeanDefinition(BeanDefinitionRegistry registry, String name, Class<?> type) {

    assertHasAnnotation(type);

    GenericBeanDefinition definition = new GenericBeanDefinition();
    definition.setBeanClass(type);

    registry.registerBeanDefinition(name, definition);
}

private void assertHasAnnotation(Class<?> type) {
    Assert.notNull(
            AnnotationUtils.findAnnotation(type, ConfigurationProperties.class),
            () -> "No " + ConfigurationProperties.class.getSimpleName()
                    + " annotation found on  '" + type.getName() + "'.");
}
```

    - ~

2.4 ConfigurationPropertiesBindingPostProcessorRegistrar

org.springframework.boot.context.properties.ConfigurationPropertiesBindingPostProcessorRegistrar ，实现 ImportBeanDefinitionRegistrar 接口，代码如下：

```
@Override
public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {
    if (!registry.containsBeanDefinition(ConfigurationPropertiesBindingPostProcessor.BEAN_NAME)) {

        registerConfigurationPropertiesBindingPostProcessor(registry);

        registerConfigurationBeanFactoryMetadata(registry);
```

```
        registerConfigurationBeanFactoryMetadata(registry);
    }
}
```

- <1> 处，调用 #registerConfigurationPropertiesBindingPostProcessor(BeanDefinitionRegistry registry) 方法，注册 ConfigurationPropertiesBindingPostProcessor BeanDefinition 。代码如下：

```
private void registerConfigurationPropertiesBindingPostProcessor(BeanDefinitionRegistry registry) {

    GenericBeanDefinition definition = new GenericBeanDefinition();
    definition.setBeanClass(ConfigurationPropertiesBindingPostProcessor.class);
    definition.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);


    registry.registerBeanDefinition(ConfigurationPropertiesBindingPostProcessor.BEAN_NAME, definition);
}
```

  - 关于 ConfigurationPropertiesBindingPostProcessor 类，我们在 「4. ConfigurationPropertiesBindingPostProcessor 相关」 中，详细解析。

- <2> 处，调用 #registerConfigurationBeanFactoryMetadata(BeanDefinitionRegistry registry) 方法，注册 ConfigurationBeanFactoryMetadata BeanDefinition 。代码如下：

```
private void registerConfigurationBeanFactoryMetadata(BeanDefinitionRegistry registry) {

    GenericBeanDefinition definition = new GenericBeanDefinition();
    definition.setBeanClass(ConfigurationBeanFactoryMetadata.class);
    definition.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);

    registry.registerBeanDefinition(ConfigurationBeanFactoryMetadata.BEAN_NAME, definition);
}
```

  - 关于 ConfigurationBeanFactoryMetadata 类，我们在 「3. ConfigurationBeanFactoryMetadata」 中，详细解析。

org.springframework.boot.context.properties.ConfigurationBeanFactoryMetadata ，初始化配置类创建 Bean 的每个方法的元数据。

3.1 postProcessBeanFactory

实现 #postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) 方法，代码如下：

```
private ConfigurableListableBeanFactory beanFactory;



private final Map<String, FactoryMetadata> beansFactoryMetadata = new HashMap<>();

@Override
public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {

    this.beanFactory = beanFactory;

    for (String name : beanFactory.getBeanDefinitionNames()) {

        BeanDefinition definition = beanFactory.getBeanDefinition(name);

        String method = definition.getFactoryMethodName();
        String bean = definition.getFactoryBeanName();

        if (method != null && bean != null) {
            this.beansFactoryMetadata.put(name, new FactoryMetadata(bean, method));
        }
    }
}
```

- <1> 处，初始化 beanFactory 属性。
- <2> 处，遍历所有的 BeanDefinition 的名字们，初始化 beansFactoryMetadata 属性。
- <2.1> 处，获得 BeanDefinition 对象。
- <2.2> 处，获得 BeanDefinition 的 factoryMethodName 、 factoryBeanName 属性。
  - factoryBeanName 属性，是创建该 Bean 的工厂 Bean 的名字。
  - factoryMethodName 属性，是创建 Bean 的工厂 Bean 的方法名。

- 以如下的 Configuration 类，举个例子：

```
@Configuration
public class TestConfiguration {

    @Bean
    public Object testObject() {
        return new Object();
    }

}
```

    - 每个 `@Bean` 注解的方法，都是一个 factoryBeanName + factoryMethodName 。
    - factoryBeanName 属性，为 "testConfiguration" 。
    - factoryMethodName 属性，为 "testObject" 。

- <2.3> 处，都非空的情况下，添加到 beansFactoryMetadata 中。

- FactoryMetadata 是 ConfigurationBeanFactoryMetadata 的内部静态类。代码如下：

```
private static class FactoryMetadata {

    private final String bean;

    private final String method;

}
```

3.2 findFactoryMethod

#findFactoryMethod(String beanName) 方法，获得指定 Bean 的创建方法。代码如下：

```
public Method findFactoryMethod(String beanName) {

    if (!this.beansFactoryMetadata.containsKey(beanName)) {
        return null;
    }
    AtomicReference<Method> found = new AtomicReference<>(null);

    FactoryMetadata metadata = this.beansFactoryMetadata.get(beanName);

    Class<?> factoryType = this.beanFactory.getType(metadata.getBean());
    if (ClassUtils.isCglibProxyClass(factoryType)) {
        factoryType = factoryType.getSuperclass();
    }

    String factoryMethod = metadata.getMethod();
    ReflectionUtils.doWithMethods(factoryType, (method) -> {
        if (method.getName().equals(factoryMethod)) {
            found.compareAndSet(null, method);
        }
    });
    return found.get();
}
```

3.3 findFactoryAnnotation

#findFactoryAnnotation(String beanName, Class<A> type) 方法，获得指定 Bean 的创建方法上的注解。代码如下：

```
public <A extends Annotation> A findFactoryAnnotation(String beanName, Class<A> type) {

    Method method = findFactoryMethod(beanName);
```

```
        return (method != null) ? AnnotationUtils.findAnnotation(method, type) : null;
    }
```

3.4 getBeansWithFactoryAnnotation

#getBeansWithFactoryAnnotation(Class<A> type) 方法，获得 beansFactoryMetadata 中的每个 Bean 的方法上的指定注解。代码如下：

```
    public <A extends Annotation> Map<String, Object> getBeansWithFactoryAnnotation(Class<A> type) {
        Map<String, Object> result = new HashMap<>();

        for (String name : this.beansFactoryMetadata.keySet()) {

            if (findFactoryAnnotation(name, type) != null) {
                result.put(name, this.beanFactory.getBean(name));
            }
        }
        return result;
    }
```

至此，我们基本能够明白，ConfigurationBeanFactoryMetadata 就是提供一些元数据的。

> 芳芳：因为 ConfigurationPropertiesBindingPostProcessor 涉及到好几个类，所以一起放在本小节来看看。

4.1 ConfigurationPropertiesBindingPostProcessor

org.springframework.boot.context.properties.ConfigurationPropertiesBindingPostProcessor ，实现 BeanPostProcessor、PriorityOrdered、ApplicationContextAware、InitializingBean 接口，将配置文件注入到 @ConfigurationProperties 注解的 Bean 的属性中。

4.1.1 基本属性

```
    public static final String VALIDATOR_BEAN_NAME = "configurationPropertiesValidator";

    private ConfigurationBeanFactoryMetadata beanFactoryMetadata;

    private ApplicationContext applicationContext;

    private ConfigurationPropertiesBinder configurationPropertiesBinder;

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        this.applicationContext = applicationContext;
    }

    @Override
    public void afterPropertiesSet() throws Exception {


        this.beanFactoryMetadata = this.applicationContext.getBean(ConfigurationBeanFactoryMetadata.BEAN_NAME, ConfigurationBeanFactoryMetadata.class);
        this.configurationPropertiesBinder = new ConfigurationPropertiesBinder(this.applicationContext, VALIDATOR_BEAN_NAME);
    }
```

- <1> 处，设置 applicationContext 属性。
- <2> 处，设置 beanFactoryMetadata 属性。即，我们在 「3. ConfigurationBeanFactoryMetadata」 中看到的。
- <3> 处，创建 ConfigurationPropertiesBinder 对象，设置到 configurationPropertiesBinder 属性。TODO ConfigurationPropertiesBinder

4.1.2 postProcessBeforeInitialization

实现 #postProcessBeforeInitialization(Object bean, String beanName) 方法，代码如下：

```
@Override
public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {

    ConfigurationProperties annotation = getAnnotation(bean, beanName, ConfigurationProperties.class);
    if (annotation != null) {

        bind(bean, beanName, annotation);
    }
    return bean;
}
```

- <1> 处，调用 #getAnnotation(Object bean, String beanName, Class<A> type) 方法，获得 Bean 上的 @ConfigurationProperties 属性。代码如下：

```
private <A extends Annotation> A getAnnotation(Object bean, String beanName, Class<A> type) {

    A annotation = this.beanFactoryMetadata.findFactoryAnnotation(beanName, type);

    if (annotation == null) {
        annotation = AnnotationUtils.findAnnotation(bean.getClass(), type);
    }
    return annotation;
}
```

- <2> 处，调用 #bind(Object bean, String beanName, ConfigurationProperties annotation) 方法，将配置文件注入到 @ConfigurationProperties 注解的 Bean 的属性中。代码如下：

```
private void bind(Object bean, String beanName, ConfigurationProperties annotation) {

    ResolvableType type = getBeanType(bean, beanName);

    Validated validated = getAnnotation(bean, beanName, Validated.class);

    Annotation[] annotations = (validated != null)
            ? new Annotation[] { annotation, validated }
            : new Annotation[] { annotation };

    Bindable<?> target = Bindable.of(type).withExistingValue(bean).withAnnotations(annotations);
    try {
        this.configurationPropertiesBinder.bind(target);
    } catch (Exception ex) {
        throw new ConfigurationPropertiesBindException(beanName, bean, annotation, ex);
    }
}
```

  - <2.1> 处，调用 #getBeanType(Object bean, String beanName) 方法，解析 Bean 的类型。代码如下：

```
private ResolvableType getBeanType(Object bean, String beanName) {

    Method factoryMethod = this.beanFactoryMetadata.findFactoryMethod(beanName);

    if (factoryMethod != null) {
        return ResolvableType.forMethodReturnType(factoryMethod);
    }

    return ResolvableType.forClass(bean.getClass());
}
```

    - 两种情况，见注释。

  - <2.2> 处，调用 #getAnnotation(Object bean, String beanName, Class<A> type) 方法，获得 Bean 上的 @Validated 属性。@ConfigurationProperties 注解，可以配合 @Validated 注解，一起使用，从而实现校验的功能。具体可以看看 《Enable ConfigurationProperties validation with @Validated on the factory method》 文章。
  - <2.3> 处，创建 Annotation 数组。
  - <2.4> 处，创建 Bindable 对象。我们先不用去理解 Bindable 是个锤子，至少我们看到了 withExistingValue(bean) 设置了 Bean 对象， withAnnotations(annotations) 设置了 Annotation 注解数组。
  - <2.5> 处，调用 ConfigurationPropertiesBinder#bind(Bindable<?> target) 方法，将配置文件注入到 @ConfigurationProperties 注解的 Bean 的属性中。详细解析，见 「4.2 ConfigurationPropertiesBinder」 。

4.2 ConfigurationPropertiesBinder

org.springframework.boot.context.properties.ConfigurationPropertiesBinder ，处理 @ConfigurationProperties 注解的 Bean 的属性的注入。其类上的注释如下：

4.2.1 构造方法

```
private final ApplicationContext applicationContext;

private final PropertySources propertySources;

private final Validator configurationPropertiesValidator;

private final boolean jsr303Present;


private volatile Validator jsr303Validator;

private volatile Binder binder;

ConfigurationPropertiesBinder(ApplicationContext applicationContext, String validatorBeanName) {
        this.applicationContext = applicationContext;
        this.propertySources = new PropertySourcesDeducer(applicationContext).getPropertySources();
        this.configurationPropertiesValidator = getConfigurationPropertiesValidator(applicationContext, validatorBeanName);
        this.jsr303Present = ConfigurationPropertiesJsr303Validator.isJsr303Present(applicationContext);
}
```

- `<1>` 处，设置 applicationContext 属性。
- `<2>` 处，创建 org.springframework.boot.context.properties.PropertySourcesDeducer 对象，然后调用 PropertySourcesDeducer#getPropertySources() 方法，获得 PropertySource 数组，之后设置给 propertySources 属性。关于 PropertySourcesDeducer.java 类，胖友点击链接，自己看看即可。
- `<3>` 处，调用 #getConfigurationPropertiesValidator(ApplicationContext applicationContext, String validatorBeanName) 方法，获得配置的 Validator 对象。代码如下：

```
private Validator getConfigurationPropertiesValidator(ApplicationContext applicationContext, String validatorBeanName) {
        if (applicationContext.containsBean(validatorBeanName)) {
                return applicationContext.getBean(validatorBeanName, Validator.class);
        }
        return null;
}
```

  - 从上面的文章，可以知道 validatorBeanName 为 "configurationPropertiesValidator" 。即，创建的 Validator Bean 的对象。
  - 一般情况下，我们不会配置该 Bean 对象，所以返回 null 。因此吧，可以暂时无视这个 configurationPropertiesValidator 属性~。
- `<4>` 处，调用 ConfigurationPropertiesJsr303Validator#isJsr303Present(ApplicationContext applicationContext) 方法，是否有引入 Jsr 303 Validator 相关的依赖。关于它，详细解析见 「4.3 ConfigurationPropertiesJsr303Validator」 中。

4.2.2 bind

#bind(Bindable<?> target) 方法，处理 @ConfigurationProperties 注解的 Bean 的属性的注入。代码如下：

```
public void bind(Bindable<?> target) {

    ConfigurationProperties annotation = target.getAnnotation(ConfigurationProperties.class);
    Assert.state(annotation != null, () -> "Missing @ConfigurationProperties on " + target);

    List<Validator> validators = getValidators(target);

    BindHandler bindHandler = getBindHandler(annotation, validators);

    getBinder().bind(annotation.prefix(), target, bindHandler);
}
```

- `<1>` 处，获得 @ConfigurationProperties 注解的属性。
- `<2>` 处，调用 #getValidators(Bindable<?> target) 方法，获得 Validator 数组。代码如下：

```
private List<Validator> getValidators(Bindable<?> target) {
    List<Validator> validators = new ArrayList<>(3);

    if (this.configurationPropertiesValidator != null) {
        validators.add(this.configurationPropertiesValidator);
    }

    if (this.jsr303Present && target.getAnnotation(Validated.class) != null) {
```

```
        validators.add(getJsr303Validator());
    }

    if (target.getValue() != null && target.getValue().get() instanceof Validator) {
        validators.add((Validator) target.getValue().get());
    }
    return validators;
}


private Validator getJsr303Validator() {
    if (this.jsr303Validator == null) {
        this.jsr303Validator = new ConfigurationPropertiesJsr303Validator(this.applicationContext);
    }
    return this.jsr303Validator;
}
```

    - 三个来源。

- <3> 处，调用 #getBindHandler(ConfigurationProperties annotation, List<Validator> validators) 方法，获得 BindHandler 对象。代码如下：

```
private BindHandler getBindHandler(ConfigurationProperties annotation, List<Validator> validators) {
        BindHandler handler = new IgnoreTopLevelConverterNotFoundBindHandler();

        if (annotation.ignoreInvalidFields()) {
                handler = new IgnoreErrorsBindHandler(handler);
        }

        if (!annotation.ignoreUnknownFields()) {
                UnboundElementsSourceFilter filter = new UnboundElementsSourceFilter();
                handler = new NoUnboundElementsBindHandler(handler, filter);
        }

        if (!validators.isEmpty()) {
                handler = new ValidationBindHandler(handler, validators.toArray(new Validator[0]));
        }

        for (ConfigurationPropertiesBindHandlerAdvisor advisor : getBindHandlerAdvisors()) {
                handler = advisor.apply(handler);
        }
        return handler;
}

private List<ConfigurationPropertiesBindHandlerAdvisor> getBindHandlerAdvisors() {
        return this.applicationContext.getBeanProvider(ConfigurationPropertiesBindHandlerAdvisor.class)
                        .orderedStream().collect(Collectors.toList());
}
```

    - <X> 处，通过将 handler 包装成 ValidationBindHandler 对象，从而实现 Validator 功能的提供。
    - <Y> 处，此处的 org.springframework.boot.context.properties.ConfigurationPropertiesBindHandlerAdvisor 接口，通过实现它，并注册到 Spring 容器中，可以对 handler 进一步处理。😈 当然，大多数情况下，包括 Spring Boot 也并未提供其实现，我们不需要这么做。所以呢，这块我们又可以无视落。
    - 🙂 另外，关于 BindHandler 是什么，我们先不用去研究。后续，我们放在另外的文章，来慢慢讲解~

- <4> 处，调用 #getBinder() 方法，获得 Binder 对象。代码如下：

```
private Binder getBinder() {
        if (this.binder == null) {

                this.binder = new Binder(getConfigurationPropertySources(),
                                getPropertySourcesPlaceholdersResolver(),
                getConversionService(),
                                getPropertyEditorInitializer());
        }
        return this.binder;
}

private Iterable<ConfigurationPropertySource> getConfigurationPropertySources() {
        return ConfigurationPropertySources.from(this.propertySources);
}

private PropertySourcesPlaceholdersResolver getPropertySourcesPlaceholdersResolver() {
        return new PropertySourcesPlaceholdersResolver(this.propertySources);
}

private ConversionService getConversionService() {
        return new ConversionServiceDeducer(this.applicationContext).getConversionService();
}
```

```
private Consumer<PropertyEditorRegistry> getPropertyEditorInitializer() {
        if (this.applicationContext instanceof ConfigurableApplicationContext) {
                return ((ConfigurableApplicationContext) this.applicationContext).getBeanFactory()::copyRegisteredEditorsTo;
        }
        return null;
}
```

- `<X>` 处，创建 ConversionServiceDeducer 创建，然后调用 ConversionServiceDeducer#getConversionService() 方法，获得 ConversionService 对象。ConversionService 是 Spring 中，用来作为类型转换器的。关于 org.springframework.boot.context.properties.ConversionServiceDeducer 类，胖友点击链接，简单看看即可。当然，也可以不看~

- `<4>` 处，调用 Binder#bind(String name, Bindable<T> target, BindHandler handler) 方法，执行绑定逻辑，处理 @ConfigurationProperties 注解的 Bean 的属性的注入。😈 至此，撒花~

4.3 ConfigurationPropertiesJsr303Validator

org.springframework.boot.context.properties.ConfigurationPropertiesJsr303Validator ，实现 Validator 接口， @ConfigurationProperties + @Validated 注解的 Bean 的 JSR303 的 Validator 实现类。其类上的注释如下：

4.3.1 构造方法

```
private final Delegate delegate;

ConfigurationPropertiesJsr303Validator(ApplicationContext applicationContext) {
        this.delegate = new Delegate(applicationContext);
}

private static class Delegate extends LocalValidatorFactoryBean {

    Delegate(ApplicationContext applicationContext) {

        setApplicationContext(applicationContext);

        setMessageInterpolator(new MessageInterpolatorFactory().getObject());

        afterPropertiesSet();
    }

}
```

4.3.2 isJsr303Present

#isJsr303Present(ApplicationContext applicationContext) 方法，校验是否支持 JSR303 。代码如下：

```
private static final String[] VALIDATOR_CLASSES = { "javax.validation.Validator",
                "javax.validation.ValidatorFactory",
                "javax.validation.bootstrap.GenericBootstrap" };

public static boolean isJsr303Present(ApplicationContext applicationContext) {
        ClassLoader classLoader = applicationContext.getClassLoader();
        for (String validatorClass : VALIDATOR_CLASSES) {
                if (!ClassUtils.isPresent(validatorClass, classLoader)) {
                        return false;
                }
        }
        return true;
}
```

- 通过判断，是否引入了相关的依赖。

4.3.3 supports

实现 #supports(Class<?> type) 方法，判断是否支持指定类的校验。代码如下：

```
@Override
public boolean supports(Class<?> type) {
        return this.delegate.supports(type);
```

```
    }
```

4.3.4 validate

实现 `#validate(Object target, Errors errors)` 方法，执行校验。代码如下：

```
@Override
public void validate(Object target, Errors errors) {

        this.delegate.validate(target, errors);
}
```

呼呼，终于写了一篇相对短一点的文章，舒服~ 关于本文看到的 Binder、BinderHandler、Bindable 等等类，属于 `org.springframework.boot.context.properties.bind` 包，后续我们根据需要，会对这块在进行详细的解析~

参考和推荐如下文章：

- oldflame-Jm 《Spring boot 源码分析 - ConfigurationProperties》
- 梦想 2018 《spring @EnableConfigurationProperties 实现原理》
- 一个努力的码农
    - 《spring boot 源码解析 13-@ConfigurationProperties 是如何生效的》
    - 《spring boot 源码解析 14 - 默认错误页面处理流程, 自定义, 及 EnableAutoConfigurationImportSelector 处理》