

### 第03课：@ConfigurationProperties 实现原理与实战

在 SpringBoot 中，当需要获取到配置文件数据时，除了可以用 Spring 自带的 @Value 注解外，SpringBoot 还提供了一种更加方便的方式：@ConfigurationProperties。只要在 Bean 上添加上了这个注解，指定好配置文件的前缀，那么对应的配置文件数据就会自动填充到 Bean 中。举个例子，现在有如下配置：

```
myconfig.name=test
myconfig.age=22
myconfig.desc=这是我的测试描述
```

添加对应的配置类，并添加上注解 @ConfigurationProperties，指定前缀为 myconfig：

```
@Component
@ConfigurationProperties(prefix = "myconfig")
public class MyConfig {
    private String name;
    private Integer age;
    private String desc;
    //get/set 略
    @Override
    public String toString() {
        return "MyConfig [name=" + name + ", age=" + age + ", desc=" + desc + "];"
    }
}
```

添加使用：

```
public static void main(String[] args) throws Exception {
    SpringApplication springApplication = new SpringApplication(Application.class)
    ;
    // 非web环境
    springApplication.setWebEnvironment(false);
    ConfigurableApplicationContext application = springApplication.run(args);

    MyConfig config = application.getBean(MyConfig.class);
    log.info(config.toString());
    application.close();
}
```

可以看到输出log：

```
com.cml.chat.lesson.lesson3.Application - MyConfig [name=test, age=22, desc=这是我的
测试描述]
```

对应的属性都注入了配置中的值，而且不需要其他操作。是不是非常神奇？那么下面来剖析下 @ConfigurationProperties 到底做了什么？

首先进入 @ConfigurationProperties 源码中，可以看到如下注释提示：

```
@org.springframework.boot.context.properties.ConfigurationProperties
@Target(value={METHOD, TYPE})
@Retention(value=RUNTIME)
@Documented

Annotation for externalized configuration. Add this to a class definition or a @Bean method in a @Configuratio
class if you want to bind and validate some external Properties (e.g. from a .properties file).

Note that contrary to @Value, SpEL expressions are not evaluated since property values are externalized.

Author:
    Dave Syer
See Also:
    ConfigurationPropertiesBindingPostProcessor
    EnableConfigurationProperties
```

See Also 中给我们推荐了 ConfigurationPropertiesBindingPostProcessor 和 EnableConfigurationProperties 两个类，EnableConfigurationProperties 先放到一边，因为后面的文章中会详解 EnableXX 框架的实现原理，这里就先略过。那么重点来看看 ConfigurationPropertiesBindingPostProcessor，光看类名是不是很亲切？不知上篇文章中讲的 BeanPostProcessor 是否还有印象，没有的话赶紧回头看看哦。

## ConfigurationPropertiesBindingPostProcessor

一看就知道和 BeanPostProcessor 有扯不开的关系，进入源码可以看到，该类实现的 BeanPostProcessor 和其他多个接口：

```
public class ConfigurationPropertiesBindingPostProcessor implements BeanPostProcessor,
    BeanFactoryAware, EnvironmentAware, ApplicationContextAware, InitializingBean,
    DisposableBean, ApplicationListener<ContextRefreshedEvent>, PriorityOrdered
```

这里是不是非常直观，只看类的继承关系就可以猜出大概这个类做了什么。BeanFactoryAware、EnvironmentAware、ApplicationContextAware 是 Spring 提供的获取 Spring 上下文中指定对象的方法而且优先于 BeanPostProcessor 调用，至于如何工作的后面的文章会进行详解，这里只要先了解一下作用就可以了。

此类同样实现了 InitializingBean 接口，从上篇文章中已经知道了 InitializingBean 是在 BeanPostProcessor.postProcessBeforeInitialization 之后调用，那么 postProcessBeforeInitialization 目前就是我们需要关注的重要入口方法。

先看看源码：

```

@Override
public Object postProcessBeforeInitialization(Object bean, String beanName)
    throws BeansException {
    //直接通过查找添加了ConfigurationProperties注解的类
    ConfigurationProperties annotation = AnnotationUtils
        .findAnnotation(bean.getClass(), ConfigurationProperties.class);
    if (annotation != null) {
        postProcessBeforeInitialization(bean, beanName, annotation);
    }
    //查找使用工厂bean中是否有ConfigurationProperties注解
    annotation = this.beans.findFactoryAnnotation(beanName,
        ConfigurationProperties.class);
    if (annotation != null) {
        postProcessBeforeInitialization(bean, beanName, annotation);
    }
    return bean;
}

private void postProcessBeforeInitialization(Object bean, String beanName,
    ConfigurationProperties annotation) {
    Object target = bean;
    PropertiesConfigurationFactory<Object> factory = new PropertiesConfigurationFa
ctory<Object>(
        target);
    factory.setPropertySources(this.propertySources);
    factory.setValidator(determineValidator(bean));
    // If no explicit conversion service is provided we add one so that (at least)
    // comma-separated arrays of convertibles can be bound automatically
    factory.setConversionService(this.conversionService == null
        ? getDefaultConversionService() : this.conversionService);
    if (annotation != null) {
        factory.setIgnoreInvalidFields(annotation.ignoreInvalidFields());
        factory.setIgnoreUnknownFields(annotation.ignoreUnknownFields());
        factory.setExceptionIfInvalid(annotation.exceptionIfInvalid());
        factory.setIgnoreNestedProperties(annotation.ignoreNestedProperties());
        if (StringUtils.hasLength(annotation.prefix())) {
            factory.setTargetName(annotation.prefix());
        }
    }
    try {
        factory.bindPropertiesToTarget();
    }
    catch (Exception ex) {
        String targetClass = ClassUtils.getShortName(target.getClass());
        throw new BeanCreationException(beanName, "Could not bind properties to "
            + targetClass + " (" + getAnnotationDetails(annotation) + ")", ex)
;
    }
}

```

```
}
```

在 `postProcessBeforeInitialization` 方法中，会先去找所有添加了 `ConfigurationProperties` 注解的类对象，找到后调用 `postProcessBeforeInitialization` 进行属性数据装配。

那么现在可以将实现拆分成如何寻找和如何装配两部分来说明，首先先看下如何查找到 `ConfigurationProperties` 注解类呢？

## 查找 `ConfigurationProperties`

在 `postProcessBeforeInitialization` 方法中先通过 `AnnotationUtils` 查找类是否添加了 `@ConfigurationProperties` 注解，然后再通过

```
this.beans.findFactoryAnnotation(beanName,  
    ConfigurationProperties.class);
```

继续查找。

下面详解这两步查找的作用。

- `AnnotationUtils`: `AnnotationUtils.findAnnotation(bean.getClass(), ConfigurationProperties.class)`;这个是 Spring 中常用的工具类了，通过反射的方式获取类上的注解，如果此类添加了注解 `@ConfigurationProperties` 那么这个方法会返回这个注解对象和类上配置的注解属性。
- `beans.findFactoryAnnotation`: 这里的 `beans` 是 `ConfigurationBeanFactoryMetadata` 对象。在 Spring 中，可以以工厂 bean 的方式添加 Bean，这个类的作用就是在工程 Bean 中找到 `@ConfigurationProperties` 注解。下面分析下实现过程。

`ConfigurationBeanFactoryMetadata`:

```

public class ConfigurationBeanFactoryMetaData implements BeanFactoryPostProcessor
{

    private ConfigurableListableBeanFactory beanFactory;

    private Map<String, MetaData> beans = new HashMap<String, MetaData>();

    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)
        throws BeansException {
        this.beanFactory = beanFactory;
        //迭代所有的bean定义，找出那些是工厂bean的对象添加到beans中
        for (String name : beanFactory.getBeanDefinitionNames()) {
            BeanDefinition definition = beanFactory.getBeanDefinition(name);
            String method = definition.getFactoryMethodName();
            String bean = definition.getFactoryBeanName();
            if (method != null && bean != null) {
                this.beans.put(name, new MetaData(bean, method));
            }
        }
    }

    public <A extends Annotation> Map<String, Object> getBeansWithFactoryAnnotation(
        Class<A> type) {
        Map<String, Object> result = new HashMap<String, Object>();
        for (String name : this.beans.keySet()) {
            if (findFactoryAnnotation(name, type) != null) {
                result.put(name, this.beanFactory.getBean(name));
            }
        }
        return result;
    }

    public <A extends Annotation> A findFactoryAnnotation(String beanName,
        Class<A> type) {
        Method method = findFactoryMethod(beanName);
        return (method == null ? null : AnnotationUtils.findAnnotation(method, type));
    }

    //略...

    private static class MetaData {
        private String bean;
        private String method;
        //构造方法和其他方法略...
    }
}

```

通过以上代码可以得出 `ConfigurationBeanFactoryMetaData` 的工作机制，通过实现 `BeanFactoryPostProcessor`，在回调方法 `postProcessBeanFactory` 中，查找出所有通过工厂 Bean 实现的对象，并将其保存到 Beans Map 中，通过方法 `findFactoryAnnotation` 可以查询到工厂 Bean 中是否添加了对应的注解。那么这里的功能就是查找工厂 Bean 中有添加 `@ConfigurationProperties` 注解的类了。

## 属性值注入

通过上述步骤，已经确认了当前传入的 Bean 是否添加了 `@ConfigurationProperties` 注解。如果添加了则下一步就需要进行属性值注入了，核心代码在方法 `postProcessBeforeInitialization` 中：

```
private void postProcessBeforeInitialization(Object bean, String beanName,
        ConfigurationProperties annotation) {
    Object target = bean;
    PropertiesConfigurationFactory<Object> factory = new PropertiesConfigurationFactory<Object>(
        target);
    //重点，这里设置数据来源
    factory.setPropertySources(this.propertySources);
    factory.setValidator(determineValidator(bean));
    //设置转换器
    factory.setConversionService(this.conversionService == null
        ? getDefaultConversionService() : this.conversionService);
    if (annotation != null) {
        //将annotation中配置的属性配置到factory中
    }
    try {
        //这里是核心，绑定属性值到对象中
        factory.bindPropertiesToTarget();
    }
    catch (Exception ex) {
        //抛出异常
    }
}
```

继续跟进 `factory.bindPropertiesToTarget` 方法，在 `bindPropertiesToTarget` 方法中，调用的是 `doBindPropertiesToTarget` 方法：

```

private void doBindPropertiesToTarget() throws BindException {
    RelaxedDataBinder dataBinder
    //略...
    //1、获取bean中所有的属性名称
    Set<String> names = getNames(relaxedTargetNames);
    //2、将属性名称和前缀转换为配置文件的key值
    PropertyValues propertyValues = getPropertySourcesPropertyValues(names, relaxed
TargetNames);
    //3、通过上面两个步骤找到的属性从配置文件中获取数据通过反射注入到bean中
    dataBinder.bind(propertyValues);
    //数据校验
    if (this.validator != null) {
        dataBinder.validate();
    }
    //判断数据绑定过程中是否有错误
    checkForBindingErrors(dataBinder);
}

```

上面代码中使用 `dataBinder.bind` 方法进行属性值赋值，源码如下：

```

public void bind(PropertyValues pvs) {
    MutablePropertyValues mpvs = (pvs instanceof MutablePropertyValues) ?
        (MutablePropertyValues) pvs : new MutablePropertyValues(pvs);
    doBind(mpvs);
}
protected void doBind(MutablePropertyValues mpvs) {
    checkAllowedFields(mpvs);
    checkRequiredFields(mpvs);
    //进行赋值
    applyPropertyValues(mpvs);
}
protected void applyPropertyValues(MutablePropertyValues mpvs) {
    try {
        // Bind request parameters onto target object.
        getPropertyAccessor().setPropertyValues(mpvs, isIgnoreUnknownFields(), isI
gnoreInvalidFields());
    }
    catch (PropertyBatchUpdateException ex) {
        // Use bind error processor to create FieldErrors.
        for (PropertyAccessException pae : ex.getPropertyAccessExceptions()) {
            getBindingErrorProcessor().processPropertyAccessException(pae, get Inte
rnalBindingResult());
        }
    }
}
}

```

经过以上步骤连续的方法调用后，最终调用的是 `ConfigurablePropertyAccessor.setPropertyValues` 使用反射

进行设置属性值，到这里就不继续深入了。想要继续深入了解的可以继续阅读源码，到最后可以发现调用的是 `AbstractNestablePropertyAccessor.processLocalProperty` 中使用反射进行赋值。

通过上面的代码分析就非常清晰明了的解释了如何查找 `@ConfigurationProperties` 对象和如何使用反射的方式进行赋值。

## 总结

通过上面的步骤分析了 `@ConfigurationProperties` 分析了如何筛选 Bean 到如何注入属性值的过程，整个过程的难度还不算高，没有什么特别的难点，这又是一个非常好的 `BeanPostProcessor` 使用场景说明。

从本文中学习到 `BeanPostProcessor` 是在 `SpringBoot` 中运用，以及如何通过 `AnnotationUtils` 与 `ConfigurationBeanFactoryMetadata` 结合对系统中所有添加了指定注解的 Bean 进行扫描。