



当前位置: Java 技术驿站 (<http://cmsblogs.com>) > 死磕Java (<http://cmsblogs.com/?cat=189>) > 死磕 Spring (<http://cmsblogs.com/?cat=206>) > 正文

## 【死磕 Spring】—— IOC 之 深入分析 PropertyPlaceholderConfigurer (<http://cmsblogs.com/?p=3837>)

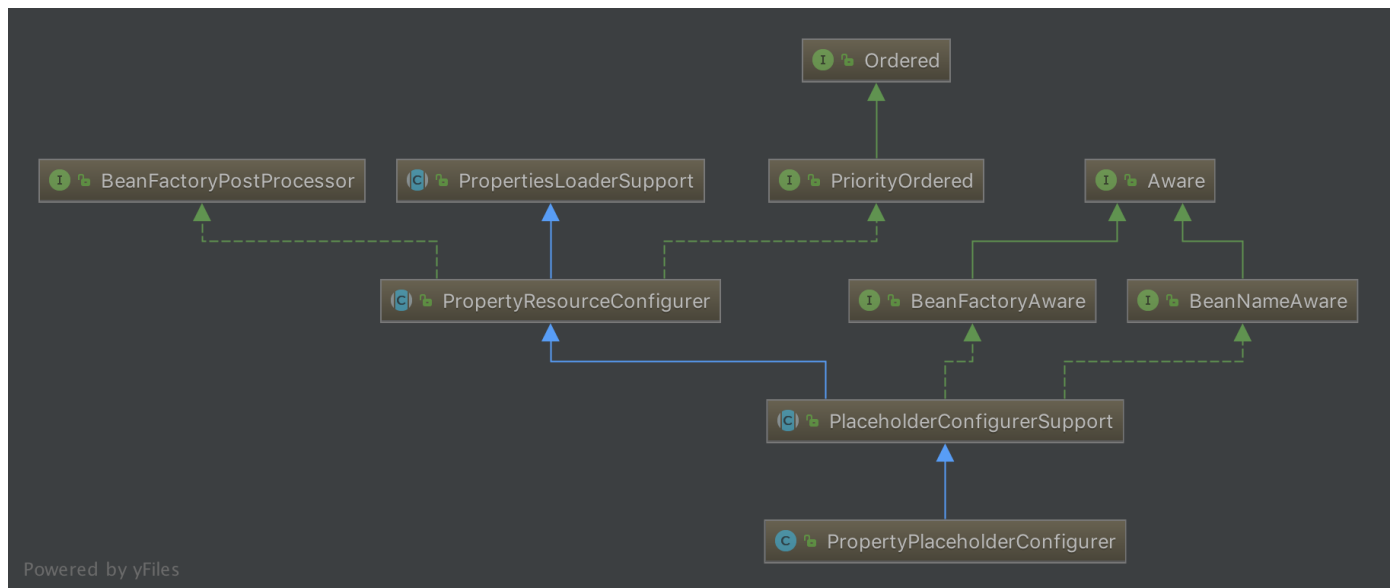
2019-01-05 分类: 死磕 Spring (<http://cmsblogs.com/?cat=206>) 阅读(6614) 评论(0)

原文出自: <http://cmsblogs.com> (<http://cmsblogs.com>)

在上文【死磕 Spring】----- IOC 之 深入分析 BeanFactoryPostProcessor (<http://cmsblogs.com/?p=3342>) 介绍了 BeanFactoryPostProcessor, 知道 BeanFactoryPostProcessor 作用域容器启动阶段, 可以对解析好的 BeanDefinition 进行定制化处理, 而其中 PropertyPlaceholderConfigurer 是其一个非常重要的应用, 也是其子类, 介绍如下:

PropertyPlaceholderConfigurer 允许我们用 Properties 文件中的属性来定义应用上下文 (配置文件或者注解)

什么意思, 就是说我们在 XML 配置文件 (或者其他方式, 如注解方式) 中使用占位符的方式来定义一些资源, 并将这些占位符所代表的资源配置到 Properties 中, 这样只需要对 Properties 文件进行修改即可, 这个特性非常, 在后面来介绍一种我们在项目中经常用到场景。



(<https://gitee.com/chenssy/blog-home/raw/master/image/201811/201809161001.png>)

从 PropertyPlaceholderConfigurer 的结构图可以看出, 它间接实现了 Aware 和 BeanFactoryPostProcessor 两大扩展接口, 这里只需要关注 BeanFactoryPostProcessor 即可。我们知道 BeanFactoryPostProcessor 提供了 postProcessBeanFactory(), 在这个体系中该方法的是在 PropertyResourceConfigurer 中实现, 该类为属性资源的配置类, 他实现了 BeanFactoryPostProcessor 接口, 如下:

```

public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {
    try {
        Properties mergedProps = mergeProperties();

        // 转换合并属性
        convertProperties(mergedProps);

        // 子类处理
        processProperties(beanFactory, mergedProps);
    }
    catch (IOException ex) {
        throw new BeanInitializationException("Could not load properties", ex);
    }
}

```

- `mergeProperties()`：返回合并的 `Properties` 实例，`Properties` 实例维护这一组 key-value，其实就是 `Properties` 配置文件中的内容。
- `convertProperties()`：转换合并的值，其实就是将原始值替换为真正的值
- `processProperties()`：前面两个步骤已经将配置文件中的值进行了处理，那么该方法就是真正的替换过程，该方法由子类实现。

在 `PropertyPlaceholderConfigurer` 重写 `processProperties()`：

```

protected void processProperties(ConfigurableListableBeanFactory beanFactoryToProcess, Properties props)
    throws BeansException {

    StringValueResolver valueResolver = new PlaceholderResolvingStringValueResolver(props);
    doProcessProperties(beanFactoryToProcess, valueResolver);
}

```

首先构造一个 `PlaceholderResolvingStringValueResolver` 类型的 `StringValueResolver` 实例。`StringValueResolver` 为一个解析 `String` 类型值的策略接口，该接口提供了 `resolveStringValue()` 方法用于解析 `String` 值。`PlaceholderResolvingStringValueResolver` 为其一个解析策略，构造方法如下：

```

public PlaceholderResolvingStringValueResolver(Properties props) {
    this.helper = new PropertyPlaceholderHelper(
        placeholderPrefix, placeholderSuffix, valueSeparator, ignoreUnresolvablePlaceholders);
    this.resolver = new PropertyPlaceholderConfigurerResolver(props);
}

```

在构造 `String` 值解析器 `StringValueResolver` 时，将已经解析的 `Properties` 实例对象封装在 `PlaceholderResolver` 实例 `resolver` 中。`PlaceholderResolver` 是一个用于解析字符串中包含占位符的替换值的策略接口，该接口有一个 `resolvePlaceholder()` 方法，用于返回占位符的替换值。还有一个 `PropertyPlaceholderHelper` 工具，从名字上面看应该是进行替换的。

得到 String 解析器的实例 valueResolver 后，则会调用 doProcessProperties() 方法来进行诊治的替换操作，该方法在父类 PlaceholderConfigurerSupport 中实现，如下：

```
protected void doProcessProperties(ConfigurableListableBeanFactory beanFactoryToProcess,
    StringValueResolver valueResolver) {

    BeanDefinitionVisitor visitor = new BeanDefinitionVisitor(valueResolver);

    String[] beanNames = beanFactoryToProcess.getBeanDefinitionNames();
    for (String curName : beanNames) {
        // 校验
        // 1. 当前实例 PlaceholderConfigurerSupport 不在解析范围内
        // 2. 同一个 Spring 容器
        if (!(curName.equals(this.beanName) && beanFactoryToProcess.equals(this.beanFactory))) {
            BeanDefinition bd = beanFactoryToProcess.getBeanDefinition(curName);
            try {
                visitor.visitBeanDefinition(bd);
            }
            catch (Exception ex) {
                throw new BeanDefinitionStoreException(bd.getResourceDescription(), curName, ex.getMessage(), ex);
            }
        }
    }




    // 别名的占位符
    beanFactoryToProcess.resolveAliases(valueResolver);

    // 解析嵌入值的占位符，例如注释属性
    beanFactoryToProcess.addEmbeddedValueResolver(valueResolver);
}
```

流程如下：

1. 根据 String 值解析策略 valueResolver 得到 BeanDefinitionVisitor 实例。BeanDefinitionVisitor 是 BeanDefinition 的访问者，我们通过它可以实现对 BeanDefinition 内容的访问，内容很多，例如 Scope、PropertyValues、FactoryMethodName 等等。
2. 得到该容器的所有 BeanName，然后对其进行访问（visitBeanDefinition()）。
3. 解析别名的占位符
4. 解析嵌入值的占位符，例如注释属性

这个方法核心在于 visitBeanDefinition() 的调用，如下：

```

public void visitBeanDefinition(BeansDefinition beanDefinition) {
    visitParentName(beanDefinition);
    visitBeanClassName(beanDefinition);
    visitFactoryBeanName(beanDefinition);
    visitFactoryMethodName(beanDefinition);
    visitScope(beanDefinition);
    if (beanDefinition.hasPropertyValues()) {
        visitPropertyValues(beanDefinition.getPropertyValues());
    }
    if (beanDefinition.hasConstructorArgumentValues()) {
        ConstructorArgumentValues cas = beanDefinition.getConstructorArgumentValues();
        visitIndexedArgumentValues(cas.getIndexedArgumentValues());
        visitGenericArgumentValues(cas.getGenericArgumentValues());
    }
}

```

我们可以看到该方法基本访问了 BeanDefinition 中所有值得访问的东西了，包括 parent、class、factory-bean、factory-method、scope、property、constructor-arg，本篇文章的主题是 property，所以关注 visitPropertyValues() 即可。如下：

```

protected void visitPropertyValues(MutablePropertyValues pvs) {
    PropertyValue[] pvArray = pvs.getPropertyValues();
    for (PropertyValue pv : pvArray) {
        Object newVal = resolveValue(pv.getValue());
        if (!ObjectUtils.nullSafeEquals(newVal, pv.getValue())) {
            pvs.add(pv.getName(), newVal);
        }
    }
}

```

过程就是对属性数组进行遍历，调用 resolveValue() 对属性进行解析获取最新值，如果新值和旧值不等，则用新值替换旧值。resolveValue() 实现如下：

```

protected Object resolveValue(@Nullable Object value) {
    // 由于 Properties 中的是 String，所以把前面一堆 if 去掉
    else if (value instanceof String) {
        return resolveStringValue((String) value);
    }
    return value;
}

```

由于配置的是 String 类型，所以只需要看 String 相关的，resolveStringValue() 实现如下：



```
protected String resolveStringValue(String strVal) throws BeansException {
    if (this.valueResolver == null) {
        throw new IllegalStateException("No StringValueResolver specified - pass a resolver " +
            "object into the constructor or override the 'resolveStringValue' method");
    }
    String resolvedValue = this.valueResolver.resolveStringValue(strVal);
    return (strVal.equals(resolvedValue) ? strVal : resolvedValue);
}
```



valueResolver 是我们在构造 BeanDefinitionVisitor 实例时传入的 String 类型解析器 PlaceholderResolvingStringValueResolver，调用其 resolveStringValue() 如下：

```
public String resolveStringValue(String strVal) throws BeansException {
    String resolved = this.helper.replacePlaceholders(strVal, this.resolver);
    if (trimValues) {
        resolved = resolved.trim();
    }
    return (resolved.equals(nullValue) ? null : resolved);
}
```

helper 为 PropertyPlaceholderHelper 实例对象，而 PropertyPlaceholderHelper 则是处理应用程序中包含占位符的字符串工具类。在构造 helper 实例对象时需要传入了几个参数：placeholderPrefix、placeholderSuffix、valueSeparator，这些值在 PlaceholderConfigurerSupport 中定义如下：

```
protected String placeholderPrefix = "${";
protected String placeholderSuffix = "}";
protected String valueSeparator = ":";
```

调用 replacePlaceholders() 进行占位符替换，如下：

```
public String replacePlaceholders(String value, PlaceholderResolver placeholderResolver) {
    Assert.notNull(value, "'value' must not be null");
    return parseStringValue(value, placeholderResolver, new HashSet<>());
}
```

调用 parseStringValue()，这个方法是这篇博客最核心的地方，\${} 占位符的替换：



protected String parseStringValue(



String value, PlaceholderResolver placeholderResolver, Set&lt;String&gt; visitedPlaceholders) {

```

    StringBuilder result = new StringBuilder(value);

```

```

    // 获取前缀 "${" 的索引位置

```

```

    int startIndex = value.indexOf(this.placeholderPrefix);

```

```

    while (startIndex != -1) {

```

```

        // 获取 后缀 "}" 的索引位置

```

```

        int endIndex = findPlaceholderEndIndex(result, startIndex);

```

```

        if (endIndex != -1) {

```

```

            // 截取 "${" 和 "}" 中间的内容，这也就是我们在配置文件中对应的值

```

```

            String placeholder = result.substring(startIndex + this.placeholderPrefix.length(), endIndex);

```

```

        dex);

```

```

        String originalPlaceholder = placeholder;

```

```

        if (!visitedPlaceholders.add(originalPlaceholder)) {

```

```

            throw new IllegalArgumentException(

```

```

                "Circular placeholder reference '" + originalPlaceholder + "' in property def

```

```

initions");

```

```

        }

```

```

        // 解析占位符键中包含的占位符，真正的值

```

```

        placeholder = parseStringValue(placeholder, placeholderResolver, visitedPlaceholders);

```

```

        // 从 Properties 中获取 placeholder 对应的值 propVal

```

```

        String propVal = placeholderResolver.resolvePlaceholder(placeholder);

```

```

        // 如果不存在

```

```

        if (propVal == null && this.valueSeparator != null) {

```

```

            // 查询 : 的位置

```

```

            int separatorIndex = placeholder.indexOf(this.valueSeparator);

```

```

            // 如果存在 :

```

```

            if (separatorIndex != -1) {

```

```

                // 获取 : 前面部分 actualPlaceholder

```

```

                String actualPlaceholder = placeholder.substring(0, separatorIndex);

```

```

                // 获取 : 后面部分 defaultValue

```

```

                String defaultValue = placeholder.substring(separatorIndex + this.valueSeparator.

```

```

length());

```

```

                // 从 Properties 中获取 actualPlaceholder 对应的值

```

```

                propVal = placeholderResolver.resolvePlaceholder(actualPlaceholder);

```

```

                // 如果不存在 则返回 defaultValue

```

```

                if (propVal == null) {

```

```

                    propVal = defaultValue;

```

```

                }

```

```

            }

```

```

        }

```

```

        if (propVal != null) {

```

```

            propVal = parseStringValue(propVal, placeholderResolver, visitedPlaceholders);

```

```

            result.replace(startIndex, endIndex + this.placeholderSuffix.length(), propVal);

```

```

            if (logger.isTraceEnabled()) {

```

```

                logger.trace("Resolved placeholder '" + placeholder + "'");

```



```

    }
    startIndex = result.indexOf(this.placeholderPrefix, startIndex + propVal.length());
}
else if (this.ignoreUnresolvablePlaceholders) {
    // 忽略值
    startIndex = result.indexOf(this.placeholderPrefix, endIndex + this.placeholderSuffix
.length());
}
else {
    throw new IllegalArgumentException("Could not resolve placeholder '" +
placeholder + "'" + " in value \"" + value + "\"");
}

//
visitedPlaceholders.remove(originalPlaceholder);
}
else {
    startIndex = -1;
}
}

// 返回propVal，就是替换之后的值
return result.toString();
}

```

### 流程如下

1. 获取占位符前缀 "\${" 的索引位置 startIndex
2. 如果前缀 "\${" 存在，则从 "{" 后面开始获取占位符后缀 "}" 的索引位置 endIndex
3. 如果前缀 "\${" 和后缀 "}" 都存在，则截取中间部分 placeholder
4. 从 Properties 中获取 placeHolder 对应的值 propVal
5. 如果 propVal 为空，则判断占位符中是否存在 ":", 如果存在则对占位符进行分割处理，前面部分为 actualPlaceholder，后面部分 defaultValue，尝试从 Properties 中获取 actualPlaceholder 对应的值 propVal，如果不存在，则将 defaultValue 的值赋值给 propVal
6. 返回 propVal，也就是 Properties 中对应的值

到这里占位符的解析就结束了，下篇我们将利用 PropertyPlaceholderConfigurer 来实现动态加载配置文件，这个场景也是非常常见的。

👍 赞(4)

¥ 打赏

【公告】版权声明 ([http://cmsblogs.com/?page\\_id=1908](http://cmsblogs.com/?page_id=1908))

标签： Spring 源码解析 (<http://cmsblogs.com/?tag=spring-%e6%ba%90%e7%a0%81%e8%a7%a3%e6%9e%90>)

死磕Java (<http://cmsblogs.com/?tag=%e6%ad%bb%e7%a3%95java>)

死磕Spring (<http://cmsblogs.com/?tag=%e6%ad%bb%e7%a3%95spring>)