# ing系列-6 占位符使用和原理

Ewen Seong 已于 2024-06-02 14:41:47 修改 阅读量4.9k 收藏 15 点赞数 10
分类专栏： Spring系列 文章标签： spring java 后端

Spring系列 专栏收录该内容 19 订阅 1

化和不变是软件设计的一个原则，将不变的 部分形成模版，将变化的部分抽出为配置文件；不同的环境使用不同的配置文件，方便维护且不需要重新编
符为其提供了一个解决方案。
为Spring系列文章的第六篇，内容包含占位符的使用和背后原理；其中，原理部分会伴随着Spring源码进行。

## 立符

论的占位符指 `${}`，常见于SpringBoot的 `application.properties` (或 `application.yml` )配置文件、或自定义 `*.properties` 配置文件中，也常见于@Value等
Spring项目中，常见于Spring的配置文件，可以用在bean的定义上。占位符中的变量在程序启动过程中进行解析，developer需要引入配置文件使得解析过

## 用方式

### pring项目：

以Spring系列-4 国际化中的国际化Bean的配置过程为例进行介绍。

### 配置文件

urces资源路径下准备一个配置文件，文件内容如下：

```
# default.properties
basename=i18n/messages
defaultEncoding=UTF-8
```

n定义的配置文件中，通过<context>标签引入资源文件：

```
<context:property-placeholder location="classpath:default.properties"/>
```

以下方式：

```
<bean class="org.springframework.context.support.PropertySourcesPlaceholderConfigurer">
    <property name="location" value="classpath:default.properties"/>
</bean>
```

### on属性：

n属性为PropertySourcesPlaceholderConfigurer对象指定了配置文件路径；当需要指定多个配置文件时，应使用逗号分隔开。另外，也可以使用通配符形式

```
<context:property-placeholder location="classpath*:My*.properties"/>
```

况下，如果指定了文件而没找到时——抛出异常；使用通配符时，即使匹配结果为空也不会抛出异常。

：当在一个<context>标签中指定了多个配置文件时，处理占位符时会按照配置顺序依次向配置文件中进行匹配，第一次完成匹配时即返回；否则一直向下

配置顺序决定了配置文件的优先级，靠前的优先级较高。

### -unresolvable和ignore-resource-not-found属性：

ion外，有两个属性需要关注：ignore-unresolvable和ignore-resource-not-found.
unresolvable表示解析失败时是否忽略(不抛出异常-返回原字符串)；默认值false表示解析失败时抛出异常。ignore-resource-not-found表示获取不到配置文
认值false表示获取文件失败时抛出异常。二者经常组合出现，因为存在逻辑上的优先级顺序：当ignore-unresolvable配置为true时，无论文件是否存在-解
，即ignore-resource-not-found处于逻辑失效状态；当ignore-resource-not-found配置为false时，ignore

**个PropertySourcesPlaceholderConfigurer对象**

多个 <context:property-placeholder>标签，即配置多个PropertySourcesPlaceholderConfigurer实例时，需要注意配置好ignore-unresolvable和ignore-reso
现意料之外的结果。

要注意：这与一个<context>标签中配置多个配置文件不同；每个<context>标签对应一个独立的Bean对象。

过简单案例介绍，已知道原因的读者，可跳过过该案例：

义的配置文件如下：

```
<context:property-placeholder location="classpath:default.properties"/>
<context:property-placeholder location="classpath:local.properties"/>

<bean id="testPhc" class="com.seong.context.TestPlaceHolderConfigurer">
    <property name="name" value="${name}"/>
    <property name="age" value="${age}"/>
</bean>
```

会按照配置顺序，先后向IOC容器注入 default.properties 对应的Bean对象(使用**default解析器**表示)和 local.properties 对应的Bean对象(使用**local解析器**表
efinition时，会按照IOC顺序依次调用两个PropertySourcesPlaceholderConfigurer对象去处理 `${name}` 和 `${age}` .

Bean对象是完全独立的且解析过程在时间上先后进行，互补干扰；整个解析过程如下：
**t解析器**解析时，如果解析正常，即default.properties文件中配置了 `name` 和 `age` 变量，则将testPhc的BeanDefinition对象中的占位符替换为配置的value. 然
发现没有占位符号，直接退出解析过程，表现为整个解析过程正常。
**t解析器**解析失败时，即default.properties文件中未配置 `name` 或 `age` 变量，会直接抛出异常，不再进入其他解析器。
整个解析过程中只有**default解析器**生效，其他Bean对象都被逻辑失效了(等价于仅配置了**default解析器**)。
最后一个PropertySourcesPlaceholderConfigurer对象前的所有PropertySourcesPlaceholderConfigurer对象的属性设置为true，来解决上述问题，如下所示

```
<context:property-placeholder location="default.properties" ignore-unresolvable="true"/>
<context:property-placeholder location="location.properties"/>
```

最后一个PropertySourcesPlaceholderConfigurer对象的ignore-unresolvable属性也可以设置为true；但作为最后一个解析器，需要保持当解析失败时抛出异

量：尽量让异常尽早抛出，能在编译期的不要延迟到启动时，能在启动时抛出的不要延迟到运行过程中。

## 占位符

际化bean对象：

```
<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename" value="i18n/messages"/>
    <property name="defaultEncoding" value="UTF-8"/>
</bean>
```

置文件实现等效配置：

```
<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename" value="${basename}"/>
    <property name="defaultEncoding" value="${defaultEncoding}"/>
</bean>
```

，在Bean定义的配置文件中, 所有值对象(包括bean的id、class等属性)都可以使用占位符形式，甚至也包括引入配置文件的标签：

```
<context:property-placeholder location="default.properties"/>
<context:property-placeholder location="${location}"/>
```

需要注意如果占位符解析失败，会抛出异常；比如上面的location必须要求在default.properties进行了配置。

## pringBoot项目

Boot项目中的配置数据可以来自application.properties(或application.yml)或手动引入的自定义配置文件。

## 置文件

ropertySource 注解可手动引入配置文件：

```
@Configuration
@PropertySource(value = {"classpath:default.properties", "classpath:location.properties"})
public class PropertiesConfiguration {
}
```

application.yml文件中进行配置：

```
# application.yml
placeholder:
  serverName: PlaceHolderServer
  url: http://127.0.0.1:8080/phs
```

### 占位符

Boot中占位符常见 **@Value**注解和 **@ConfigurationProperties**等注入场景, 其中 **@ConfigurationProperties**注解是SpringBoot引入的.

**Value注解**

```
// PlaceHolderBean.java文件
@Data
@Component
public class PlaceHolderBean {
    @Value("${placeholder.serverName}")
    private String serverName;

    @Value("${placeholder.url}")
    private String url;
}
```

**Value**注解，可以将配置文件中的 placeholder.serverName 和 placeholder.url 变量分别赋值给PlaceHolderBean对象的serverName和url属性，测试用例

```
tarted PlaceHolderTest in 2.762 seconds (JVM running for 3.887)
laceHolderBean is PlaceHolderBean(serverName=PlaceHolderServer, url=http://127.0.0.
```

**ConfigurationProperties注解**

```
// PlaceHolderBean.java文件
@Data
@Component
@ConfigurationProperties(prefix = "placeholder")
public class PlaceHolderProperties {
    private String serverName;

    private String url;
}

// 测试用例
```

通过 @ConfigurationProperties 注解将配置的 placeholder.serverName 和 placeholder.url 属性值分别赋值给PlaceHolderProperties对象的serverName属

```
n 2.695 seconds (JVM running for 3.731)
PlaceHolderProperties(serverName=PlaceHolderServer, url=http://127.0.0.1:8
```

**Value注解和@ConfigurationProperties注解**

比需要注意：@Value来自Spring, 而@ConfigurationProperties来自SpringBoot.
位符要求变量名以及大小写完全匹配，如@Value及前面涉及的Spring项目中使用的占位符；但@ConfigurationProperties忽略大小写且会忽略中划线，如下

```
# application.yml
placeholder:
  serverName: PlaceHolderServer
  url: http://127.0.0.1:8080/phs
#等价于:
placeholder:
  SerVer-NA-m-e: PlaceHolderServer
  URl: http://127.0.0.1:8080/phs
```

卜，@Value不需要强行关联变量名与属性名(通过配置注解的value属性关联)，而@ConfigurationProperties需要进行变量名与属性名称的关联；
'alue除了可以使用占位符之外，还可以直接对属性注入字符串；
建议大家面向Java编程，而不面向Spring编程：一些特殊场景除外，提倡使用@ConfigurationProperties替代@Value。同理，提倡使用构造函数注入而非@

### 意点

#### 数据来源

到占位符变量的数据来源有配置文件，除此之外还包括系统属性、应用属性、环境变量.

**器环境变量：**

```
ng@EwendeMacBook-Pro local % echo $HOME
ers/seong
ng@EwendeMacBook-Pro local % echo $USER
ng
```



**试用例：**

```
@Data
@Component
public class EnvProperties {
    @Value("${HOME}")
    private String home;

    @Value("${USER}")
    private String user;
}

@Slf4j
```

**下结果：**

```
.seong.test.PlaceHolderTest            : Started PlaceHolderTest in 2.637 seconds (JVM runn
.seong.test.PlaceHolderTest            : envProperties is EnvProperties(home=/Users/seong,
```

示EnvProperties的属性值与机器对应的环境变量值保持一致。

#### 默认值

给占位符提供了默认配置，待解析字符串的**第一个冒号为分隔符**，分隔符后的值为默认值。

```
<bean id="placeHolderServer" class="org.seong.context.DefaultPlaceHolderServer">
    <property name="name" value="${servername:placeholder:001}"/>
    <property name="url" value="${url}"/>
</bean>
```

若未配置 `servername` 变量，则placeHolderServer这个bean的name属性被设置为默认值 `placeholder:001` ；若未配置url变量，则抛出异常。

#### 嵌套使用

解析表达式时会递归调用，先解析最内层的变量——得到一个**中间值**(配置文件中变量配置的值)，再解析外围；这使得${}可以嵌套使用。这个过程中，解析
归解析操作，这使得配置文件中的变量也可以引用其他变量。

**使用**

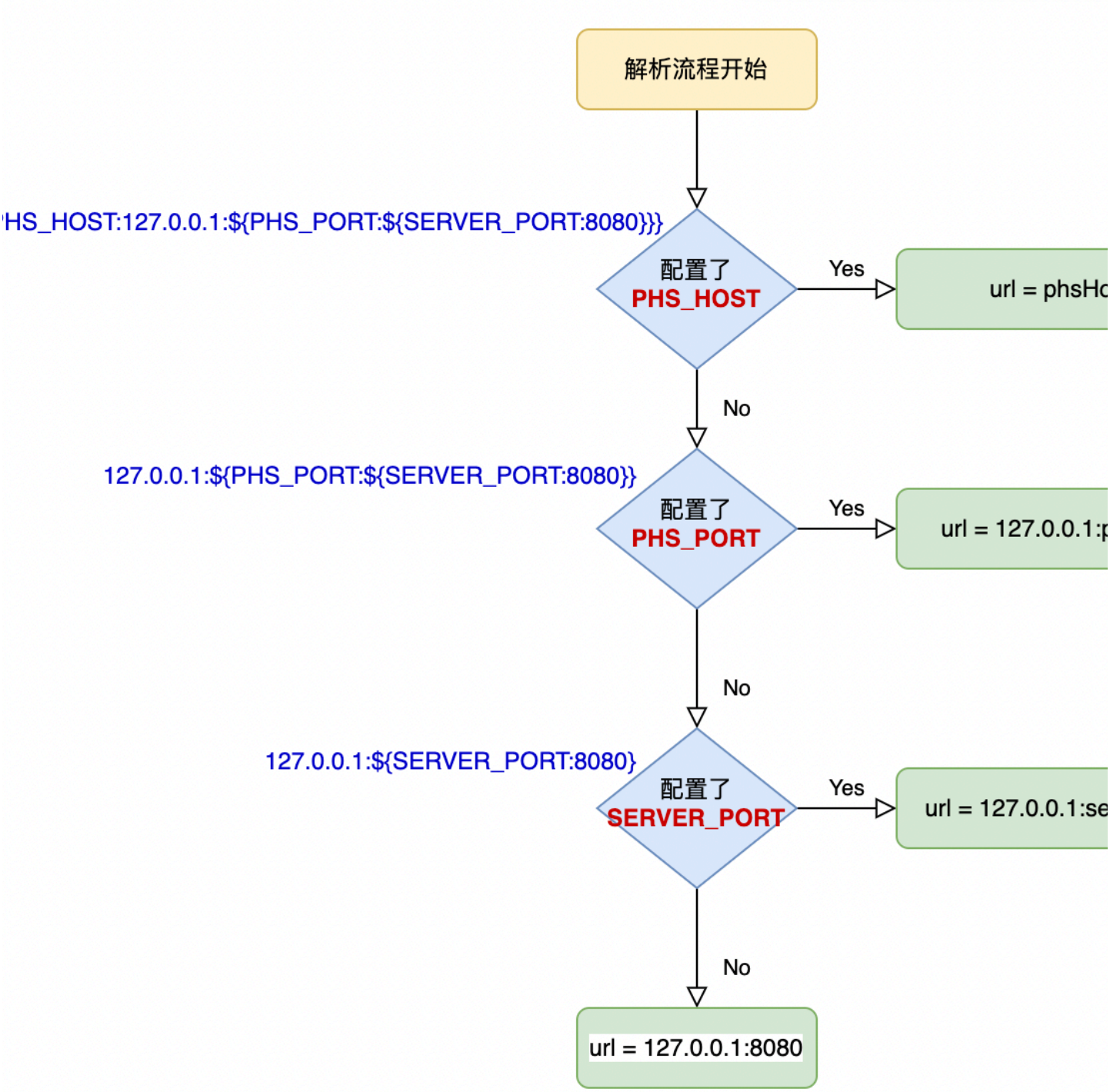
```
<bean id="placeHolderServer" class="org.seong.context.DefaultPlaceHolderServer">
    <property name="name" value="${servername:placeholder:001}"/>
    <property name="url" value="${PHS_HOST:127.0.0.1:${PHS_PORT:${SERVER_PORT:8080}}}"/>
</bean>
```

解析时由外向内，但developer在阅读和编写时应按照由外到内的顺序进行。

文件：placeHolderServer对象的url属性对应占位符字符串为 `${PHS_HOST:127.0.0.1:${PHS_PORT:${SERVER_PORT:8080}}}` ； 不妨假设变量的结果为驼峰形式

```
# 配置变量
PHS_HOST=phsHost
PHS_PORT=phsPort
SERVER_PORT=serverPort
```

否配置了PHS_HOST、PHS_PORT、SERVER_PORT变量可以得到4种不同的结果:




**件嵌套使用**

护方便，配置文件中的变量也可以抽取出公共部分，如下所示：

```
#application.yml
SERVER_IP: 127.0.0.1
SERVER_PORT: 8080
SERVER_NAME: phc

URL: http://${SERVER_IP}:${SERVER_PORT}/${SERVER_NAME}
API_URL: ${URL}/api
RPC_URL: ${URL}/rpc
```

**理**

**置知识**

原理前，需要先熟悉PropertySource相关的几个类：**Properties**、**PropertySource**(PropertiesPropertySource、ResourcePropertySource、MutablePrope**tyResolver**(PropertySourcesPropertyResolver).

### roperties类型介绍

```
public class Properties extends Hashtable<Object,Object> {
}
```

ties继承了Hashtable，因此可以看作一个特殊的Map类型(功能加强的Map), 因此Properties也基于键值对的存储结构提供了很多接口，这里只介绍与本文有

```
// 向内存中添加键值对
public synchronized Object setProperty(String key, String value) {//...}

// 根据key从内存中读取数据
public String getProperty(String key) {//...}

// load方法会从InputStream流对象中读取数据，写入到Properties中:
public synchronized void load(InputStream inStream) throws IOException {//...}
```

**ample:**
urces资源路径下准备文件：

```
// default.properties文件
name=root
passwd=root
```

码如下：

```
@Slf4j
public class PropertiesTest {
    @Test
    public void testProperties() throws IOException {
        Properties properties = new Properties();
        properties.setProperty("key-1", "value-1");
        properties.load(this.getClass().getClassLoader().getResourceAsStream("default.properties"));
        properties.setProperty("key-2", "value-2");
        LOGGER.info("properties is {}.", properties);
    }
}
```

下结果：

```
com.seong.test.PropertiesTest - properties is {passwd=root, name=root, key-2=value-2, k
```

Properties可以被用来从配置文件中加载资源入内存。

### ropertySource类型介绍

PropertySource被定义为资源对象，内部存在两个属性：name和泛型的source对象。通过 `equals` 和 `hashCode` 方法可知**name属性**被作为判断PropertySou

```
public abstract class PropertySource<T> {
    // @Getter
    protected final String name;
    // @Getter
    protected final T source;


    public boolean containsProperty(String name) {
        return this.getProperty(name) != null;
    }
```

一个抽象的 `getProperty(String propertyName)` 方法给自类实现，接口的功能是根据propertyName从so
同一概念(name仅作为PropertySource对象对外的身份)。PropertySource有两个比较重要的实现类：Pr

**rtySource的实现类PropertiesPropertySource：**

```
public class PropertiesPropertySource extends MapPropertySource {
    public PropertiesPropertySource(String name, Properties source) {
        super(name, source);
    }
    protected PropertiesPropertySource(String name, Map<String, Object> source) {
        super(name, source);
    }
    //...
}
```

tiesPropertySource的source属性为Properties类型；注意Properties是Hashtable的子类，自然也是Map类型的子类。
**MapPropertySource**实现了PropertySource定义的 `Object getProperty(String name)` 接口：

```
//MapPropertySource类
public Object getProperty(String key) {
    return ((Map)this.source).get(key);
}
```

y从Map类型的source对象中取值。

**rtySource的实现类ResourcePropertySource：**
rcePropertySource作为PropertiesPropertySource的子类，在PropertiesPropertySource基础上新增了读取资源文件的能力。

```
public class ResourcePropertySource extends PropertiesPropertySource {
    public ResourcePropertySource(Resource resource) throws IOException {
        super(getNameForResource(resource), PropertiesLoaderUtils.loadProperties(new EncodedResource(resource)));
        this.resourceName = null;
    }
    //...
}
```

`opertiesLoaderUtils.loadProperties(new EncodedResource(resource)))` 会根据传入的Resource对象指定的文件资源去加载、读取并生成Properties对象。

**rtySource的容器类MutablePropertySources：**
ePropertySources作为PropertySource的容器，在内部维持了一个PropertySource类型的列表，基于此对外提供了存储、管理、查询PropertySource对象自

```
public class MutablePropertySources implements PropertySources {
    private final List<PropertySource<?>> propertySourceList;
    //...
}
```

**PropertyResolver类型介绍**

**rtyResolver接口介绍**

```
public interface PropertyResolver {
    boolean containsProperty(String key);

    String getProperty(String key);
    String getProperty(String key, String defaultValue);
    <T> T getProperty(String key, Class<T> targetType);
    <T> T getProperty(String key, Class<T> targetType, T defaultValue);
    String getRequiredProperty(String key) throws IllegalStateException;
    <T> T getRequiredProperty(String key, Class<T> targetType) throws IllegalStateException;

    String resolvePlaceholders(String text);
    String resolveRequiredPlaceholders(String text) throws IllegalArgumentException;
}
```

tyResolver接口定义了**根据key获取value**以及**处理占位符字符串**的能力。

**rtyResolver的实现类PropertySourcesPropertyResolver：**

```
public class PropertySourcesPropertyResolver extends AbstractPropertyResolver {
    private final PropertySources propertySources;
```

```
    public PropertySourcesPropertyResolver(PropertySources propertySources) {
        this.propertySources = propertySources;
    }
    //...
}
```

PropertySourcesPropertyResolver对象时，需要传入一个PropertySources作为入参。

; `getProperty(String key)` 及其重载方法取值的实现原理：遍历propertySources对象内部的PropertySource对象，依次从中取值，直到取值成功或者遍历
tySource对象。

; `resolvePlaceholders(String text)` 的入参为待解析的字符串(包含占位符)，返回的字符串为解析后的结果。解析过程中需要通过 `String getProperty(St`
ertySource对象列表中取值。

核心方法在于：

```
protected String parseStringValue(String value, PropertyPlaceholderHelper.PlaceholderResolver placeholderResolver, @Nullable Set<String> vi
    int startIndex = value.indexOf(this.placeholderPrefix);
    if (startIndex == -1) {
        return value;
    }
    StringBuilder result = new StringBuilder(value);
    while (startIndex != -1) {
        int endIndex = this.findPlaceholderEndIndex(result, startIndex);
        if (endIndex != -1) {
            String placeholder = result.substring(startIndex + this.placeholderPrefix.length(), endIndex);
            String originalPlaceholder = placeholder;
```

整体逻辑比较简单，通过递归操作□先解析最内侧的占位符，得到一个中间值propVal(来源于配置文件或者环境变量等)；propVal可能也包含占位符，因此
回后，会按照由外到内的顺序依次进行。

### 理

框架处理占位符问题时选择的目标对象是BeanDefinition，因此无论以何种方式引入的Bean，处理过程均可统一。具体的实现方案是引入一个BeanFactory
tySourcesPlaceholderConfigurer类，并将解析逻辑封装在其内部；在Spring容器启动过程中，通过 `invokeBeanFactoryPostProcessors(beanFactory);` 进行
**rtySourcesPlaceholderConfigurer中的postProcessBeanFactory方法：**

```
public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {
    if (this.propertySources == null) {
        this.propertySources = new MutablePropertySources();
        if (this.environment != null) {
            this.propertySources.addLast(
                new PropertySource<Environment>("environmentProperties", this.environment) {
                    @Override
                    @Nullable
                    public String getProperty(String key) {
                        return this.source.getProperty(key);
                    }
```

辑较为简单：

建一个MutablePropertySources资源对象容器；

装中加入名称为**environmentProperties**的PropertySource对象，包含了系统属性、应用属性、环境变量等信息；其中application.yml中配置的属性也包含在

装中加入名称为**localProperties**的PropertySource对象，包含了引入的配置文件中的信息；

MutablePropertySources作为构造参数创建一个PropertySourcesPropertyResolver解析器对象；

用 `processProperties(beanFactory, new PropertySourcesPropertyResolver(this.propertySources));` 处理占位符。

sProperties方法共两个入参：beanFactory和PropertySourcesPropertyResolver解析器对象；beanFactory容器能够获取所有的BeanDefinition信息，
tySourcesPropertyResolver解析器对象内部包含了所有的配置信息以及基于此封装的解析能力。

**ocessProperties方法：**

```
protected void processProperties(ConfigurableListableBeanFactory beanFactoryToProcess,
    final ConfigurablePropertyResolver propertyResolver) throws BeansException {
    // 根据配置信息设置解析器，使得valueResolver与配置保持一致
    doProcessProperties(beanFactoryToProcess, valueResolver);
}
```

**›ProcessProperties方法：**

```
// 简化后，仅突出主线逻辑
protected void doProcessProperties(ConfigurableListableBeanFactory beanFactoryToProcess, StringValueResolver valueResolver) {
      BeanDefinitionVisitor visitor = new BeanDefinitionVisitor(valueResolver);
      String[] beanNames = beanFactoryToProcess.getBeanDefinitionNames();
      for (String curName : beanNames) {
          BeanDefinition bd = beanFactoryToProcess.getBeanDefinition(curName);
          visitor.visitBeanDefinition(bd);
      }
  }
```

器包装为BeanDefinitionVisitor对象，遍历IOC容器中的所有BeanDefinition，调用 `visitor.visitBeanDefinition(bd);` 解析占位符。

**sitor.visitBeanDefinition(bd)方法：**

```
public void visitBeanDefinition(BeanDefinition 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());
    }
}
```

可以看出来，依次解析BeanDefinition的parentName、class、FactoryBeanName、FactoryMethodName、scope等Bean元信息；之后解析属性值以及构造
`xxx(beanDefinition);` 方法内部的实现通过解析器实现占位符的解析。

说一下，解析占位符的过程中涉及很多类，这些类的内部设计和相互引用编织得很巧妙，在写框架代码时具备很高的参考意义，建议详细体会。

'位符工具类,使用Spring框架中PropertyPlaceholderHelper类...
'ing框架中PropertyPlaceholderHelper占位符工具类 PropertyPlaceholderHelper 类是一个实用程序类,它提供了一种简单的方法来解析字符串中的属性占位符。属性占位符是通常

配置占位符
<根元素 --> <beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframe

配置文件对于占位符 ${} 的解析源码分析 个
由于公司新项目的搭建，而且选择了 spring boot，对于 spring boot 其中的好处就是我们无需在为其配置大量的 XML，完全可以基于注解的方式来进行配置，但也没有完全采用

源码:占位符${}解析源码分析_spring ${}
spring项目中,我们经常使用到${}占位符实现属性值的获取,最常见的就是使用@Value("${propsKey}")的方式完成bean属性值的注入,如下: @Value("${appkey}") privateString appK

源码解读『占位符${…}替换』_spring 占位符字符串替换-CSDN博 ...
位符注入示例 一般我们使用xml注入会在spring配置文件中注明.properties文件的位置,一般我们会添加如下配置: <?xml version="1.0"encoding="UTF-8"?> <beans xmlns="http:

原码解析(一)---占位符解析替换 weix
类图 ①、PropertyResolver :Environment的顶层接口，主要提供属性检索和解析带占位符的文本。bean.xml配置中的所有占位符例如${}都由它解析 ②、ConfigurablePropertyR

--IOC容器（底层原理）
理指的是两个操作（1）Spring创建对象：1）配置对象：在Spring配置文件中定义需要创建的对象，包括对象的类名、属性值等信息。2）实例化对象：Spring容器根据配置文件

架面试题系列：Spring+SpringMVC+MyBatis-08.rar
映射**：使用@Param注解或者问号占位符来映射传入参数。 了解并熟练掌握这些知识点对于在面试中脱颖而出至关重要。Spring的DI和AOP能力可以帮助构建松散耦合、易于

入原理以及Spring Boot如何防止SQL注入（含详细示例代码）
queryForObject`方法接受一个预编译的SQL语句，将参数`id`作为占位符的值，而不是直接拼接到SQL字符串中。 2. 使用Spring Data JPA：Spring Data JPA是基于JPA（Java P

启动流程（原理）详解--结合web.xml加载配置分析 骏马逸动
容内容有点多，可以先总体大概浏览一遍，心中有个整体类目关系构图，再详细看，不至于迷失在细节中，最好本地也使用开发工具打开源码对着看。 引言： Spring框架已经成

实战之属性占位符配置器用法示例
召了Spring实战之属性占位符配置器用法,结合实例形式分析了spring属性占位符配置器的具体配置及使用技巧,需要的朋友可以参考下

PRINGBOOT配置文件占位符过程解析
召了基于SPRINGBOOT配置文件占位符过程解析,文中通过示例代码介绍的非常详细，对大家的学习或者工作具有一定的参考学习价值,需要的朋友可以参考下

试框架 Spring-Retry | @Retryable 注解详解
试框架 Spring Retry 的 @Retryable 注解详解。

ring Boot：那些注入不了的Spring占位符（${}表达式） 热门推荐
的占位符spring里的占位符通常表现的形式是：<bean id="dataSource" destroy-method="close" class="org.apache.commons.dbcp.BasicDataSource"> <property name="url"

占位符 得配置方式
配置需要先有spring-Context包 然后建立实体bean 必须有get和set方法 建立properties文件 值按照键值对的形式存放 如下 头文件如下 否则没有注释 <beans xmlns="htt

_属性占位符 wei
坚持虽是痛苦的事情，但却能渐渐地为你带来好处。——奥维德 属性占位符 Spring一直支持将属性定义到外部的属性的文件中，并使用占位符值将其插入到Spring bean中。

Boot配置_配置文件占位符
数 ${random.value} ${random.int} ${random.long} ${random.int(10)} ${random.int[1024,65536]} 2、占位符获取之前配置的值，如果没有可以是用:指定默认值 person.last-name

解析属性占位符 m0
装配中，占位符格式： ${ ... } 使用它包装的属性名称。 如XML配置bean <bean id="sgtPeppers" class = "com.BalnkDisc" c:_title="${disc.title}" c:_artist="${disc.artist}"

boot-starter-actuator原理
oot-starter-actuator是Spring Boot框架中的一个模块，它提供了一系列用于监控和管理应用程序的端点（endpoints），比如/health、/info、/metrics等。这些端点可以通过HTTP



Ewen Seong
码龄6年 暂无认证

6986 1万+ 17万+
周排名 总排名 访问 等级

ong: lua官网地址: https://www.lua.



36篇 2023年 18篇

17篇 2021年 13篇