超越外部化配置

理解“外部化配置”

Spring Boot 官方说明

> Spring Boot allows you to externalize your configuration so you can work with the same application code in different environments. You can use properties files, YAML files, environment variables and command-line arguments to externalize configuration.
>
> > 以上文字来源：https://docs.spring.io/spring-boot/docs/2.0.2.RELEASE/reference/htmlsingle/#boot-features-external-config

小马哥解读

何谓”外部化配置”，官方文档并没有正面解释。根据小马哥的个人经验，做出如下解释，供诸君参考。

通常，对于可扩展性应用，尤其是中间件，它们的功能性组件是可配置化的，如：认证信息、端口范围、线程池规模以及连接时间等。假设需要设置 Spring 应用的 `Profile` 为 `"dev"`，可通过调用 Spring `ConfigurableEnvironment` 的 `setActiveProfiles("dev")` 方法实现。这种方式是一种显示地代码配置，配置数据来源于应用内部实现，所以称之为"内部化配置"。“内部化配置” 虽能达成目的，然而配置行为是可以枚举的，必然缺少相应的弹性。

应用“外部化配置”

Spring Boot 官方说明应用场景

- **Bean 的 `@Value` 注入**
- **Spring `Environment` 读取**
- **`@ConfigurationProperties` 绑定到结构化对象**

实际应用场景

- 用于 XML Bean 定义的属性占位符
- 用于 `@Value` 注入
- 用于 `Environment` 读取
- 用于 `@ConfigurationProperties` Bean 绑定

- 用于 `@ConditionalOnProperty` 判断

用于 XML Bean 定义的属性占位符

一个熟悉的 Spring 示例

- 配有 `PropertyPlaceholderConfigurer` Bean 的 Spring 上下文 XML 配置文件（`META-INF/spring/user-context.xml`）

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- 属性占位符配置-->
    <bean
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <!-- Properties 文件 classpath 路径 -->
        <property name="location" value="classpath:/META-INF/default.properties"/>
        <!-- 文件字符编码 -->
        <property name="fileEncoding" value="UTF-8"/>
    </bean>

</beans>
```

- 模型类 `User`

```
public class User {

    private Long id;

    private String name;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
```

```
    }

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", name='" + name + '\'' +
                '}';
    }
}
```

- 配置属性文件（`META-INF/default.properties`）

```
# 用户配置属性
user.id = 1
user.name = 小马哥
```

- `User` Spring 上下文XML 配置文件（`META-INF/spring/user-context.xml`）

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- User Bean -->
    <bean id="user"
class="com.imooc.diveinspringboot.externalized.configuration.domain.User">
        <property name="id" value="${user.id}"/>
        <property name="name" value="${user.name}"/>
    </bean>

</beans>
```

- 实现 Spring Framework 引导类

```
public class SpringXmlConfigPlaceholderBootstrap {

    public static void main(String[] args) {

        String[] locations = {"META-INF/spring/spring-context.xml", "META-
INF/spring/user-context.xml"};

        ClassPathXmlApplicationContext applicationContext = new
ClassPathXmlApplicationContext(locations);

        User user = applicationContext.getBean("user", User.class);

        System.out.println("用户对象 : " + user);
        // 关闭上下文
        applicationContext.close();
```

```
    }
}
```

调整为 Spring Boot 示例

- 使用 `application.properties` 替换 `META-INF/default.properties`

```
# 用户配置属性（Spring Boot）
user.id = 1
user.name = 小马哥
```

- 实现 Spring Boot 引导类

```
@ImportResource("META-INF/spring/user-context.xml") // 加载 Spring 上下文 XML 文件
@EnableAutoConfiguration
public class XmlPlaceholderExternalizedConfigurationBootstrap {

    public static void main(String[] args) {

        ConfigurableApplicationContext context =
                new
SpringApplicationBuilder(XmlPlaceholderExternalizedConfigurationBootstrap.class)
                        .web(WebApplicationType.NONE) // 非 Web 应用
                        .run(args);

        User user = context.getBean("user", User.class);

        System.out.println("用户对象 : " + user);
        // 关闭上下文
        context.close();
    }
}
```

为什么结果不符合期望？

用于 `@Value` 注入

`@Value` 字段注入（Field Injection）

`@Value` 构造器注入（Constructor Injection）

`@Value` 方法注入（Method Injection）

`@Value` 默认值支持

用于 `Environment` 读取

获取 `Environment` Bean

`Environment` **方法/构造器依赖注入**

`Environment` `@Autowired` **依赖注入**

`EnvironmentAware` **接口回调**

`BeanFactory` **依赖查找** `Environment`

执行顺序：

1. `@Autowired`
2. `BeanFactoryAware`
3. `EnvironmentAware`

用于 `@ConfigurationProperties` Bean 绑定

@ConfigurationProperties 类级别标注

@ConfigurationProperties @Bean 方法声明

@ConfigurationProperties 嵌套类型绑定

松散绑定

优先级配置

- Java System Properties
- `-Duser.city.post_code=0731`
- OS Environment Variables
  - `USER_CITY_POST_CODE=001`
- application.properties
  - `user.city.post-code=0571`

@ConfigurationProperties Bean 校验

用于 @ConditionalOnProperty 判断

`@ConditionalOnProperty` `prefix` `name` 要与 application.properties 完全一致，在环境变量里面，允许松散绑定。

扩展“外部化配置”

定位外部化配置属性源

- 如何理解 `PropertySource` 顺序？
  - `@TestPropertySource#properties`
  - `@SpringBootTest#properties`
  - `@TestPropertySource#locations`

> - PropertySource[名称:configurationProperties] : ConfigurationPropertySourcesPropertySource {name='configurationProperties'}
> - PropertySource[名称:Inlined Test Properties] : MapPropertySource {name='Inlined Test Properties'}
> - PropertySource[名称:class path resource [META-INF/default.properties]] : ResourcePropertySource {name='class path resource [META-INF/default.properties]'}
> - PropertySource[名称:systemProperties] : MapPropertySource {name='systemProperties'}
> - PropertySource[名称:systemEnvironment] : OriginAwareSystemEnvironmentPropertySource {name='systemEnvironment'}
> - PropertySource[名称:random] : RandomValuePropertySource {name='random'}
> - PropertySource[名称:applicationConfig: [classpath:/application.properties]] : OriginTrackedMapPropertySource {name='applicationConfig: [classpath:/application.properties]'}

- 如何理解 `PropertySource` ？

带有名称的属性源， `Properties` 文件、Map、YAML 文件

- 什么是 `Envriroment` 抽象？

`Environment` 与 `PropertySources` 是1对1， `PropertySources` 与 `PropertySource` 是 1 对 N

`ConfigurableEnvironment` 与 `MutablePropertySources`

PropertySources 的使用时机

PropertySources 在 Spring 上下文生命周期内的使用实际

理解 Spring Boot Environment 生命周期

Spring Framework 中，尽量在 `org.springframework.context.support.AbstractApplicationContext#prepareBeanFactory` 方法前初始化。

Spring Boot 中，尽量在 `org.springframework.boot.SpringApplication#refreshContext(context)` 方法前初始化。

扩展外部化配置属性源

基于 SpringApplicationRunListener#environmentPrepared 扩展外部化配置属性源

基于 ApplicationEnvironmentPreparedEvent 扩展外部化配置属性源

基于 EnvironmentPostProcessor 扩展外部化配置属性源

基于 ApplicationContextInitializer 扩展外部化配置属性源

基于 SpringApplicationRunListener#contextPrepared 扩展外部化配置属性源

基于 SpringApplicationRunListener#contextLoaded 扩展外部化配置属性源

基于 ApplicationPreparedEvent 扩展外部化配置属性源

实时扩展外部化配置属性源

`Environment` OK

`@Value` X

`@ConfiguratinProperties` X