

Spring Boot 实现多环境配置动态解析

一、什么是外部化配置？

SpringBoot官方解释

Spring Boot lets you externalize your configuration so that 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. Property values can be injected directly into your beans by using the `@Value` annotation, accessed through Spring's `Environment` abstraction, or be [bound to structured objects](#) through `@ConfigurationProperties`.

Spring Boot允许您将配置外部化，这样您就可以在不同的环境中使用相同的应用程序代码。您可以使用properties文件、YAML文件、环境变量和命令行参数来外部化配置。属性值可以使用@Value注释直接注入bean，可以通过Spring的环境抽象进行访问，也可以通过@ConfigurationProperties绑定到结构化对象。

引用: <https://docs.spring.io/spring-boot/docs/2.0.2.RELEASE/reference/htmlsingle/#boot-features-external-config>

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的代码配置，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。所以我们需要外部化配置来解决这个问题，比如我们在SpringBoot中经常用到application.properties来进行外部化配置。

二、如何使用外部化配置？

官方应用场景

Spring Boot 官方说明应用场景有以下3种方式：

1. Bean 的@Value 注入
2. Spring Environment 读取
3. @ConfigurationProperties 绑定到结构化对象

实际应用场景

1. 用于 XML Bean 定义的属性占位符
2. 用于 @Value 注入
3. 用于 Environment 读取
4. 用于 @ConfigurationProperties Bean 绑定
5. 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。

示例1：使用XML Bean定义的属性占位符

我们通过一个熟悉的 Spring 示例先看看用于 XML Bean 定义的属性占位符：

1. 使用PropertyPlaceholderConfigurer进行外部化配置

配有 PropertyPlaceholderConfigurer Bean 的 Spring 上下文 XML 配置文件（META-INF/spring/spring-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>
```

2. 定义模型类 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 + '\'' +
            '}';
    }
}
```

```
}  
}
```

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

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

4. 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.yuandengta.boot.model.User">  
        <property name="id" value="${user.id}"/>  
        <property name="name" value="${user.name}"/>  
    </bean>  
</beans>
```

5. 实现 Spring Framework 引导类

```
public class XmlConfigBootstrap {  
    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.err.println("用户对象 : " + user);  
        // 关闭上下文  
        applicationContext.close();  
    }  
}
```

示例2：使用SpringBoot注解加载属性

上面的示例就是通过一种外部化配置来帮助我们实例化一个Bean，那么，如果我们调整为SpringBoot的时候又是怎样的呢？

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

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

2. 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 SpringBootConfigBootstrap {
    public static void main(String[] args) {
        ConfigurableApplicationContext context =
            new SpringApplicationBuilder(SpringBootConfigBootstrap.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();
    }
}
```

但是使用这种方式进行外部化配置的时候，我们会发现，user对象的name不是我们所期望的Roc字样，这是为什么呢？其实SpringBoot在使用外部化配置的时候，会有一个优先级顺序，优先级最高的顺序是我们的JVM命令中的参数，但是这里我们没有使用JVM命令参数，那么是什么把user.name给替换了呢？答案是我们的环境变量参数，这个参数的优先级比user-context.xml的优先级要高很多，所以导致了这个原因。

PropertySource 加载顺序

定位外部化配置属性源,我们知道外部属性配置源是一个一个的PropertySource，PropertySource 是带有名称的属性源， Properties 文件、Map、YAML 文件 什么是 Environment 抽象？Environment 与 PropertySources 是1对1， PropertySources 与 PropertySource 是 1 对 N，所以我们经常在 SpringBoot应用中可以使用Environment来读取所有的配置。其实我们还可以自定义SpringBoot的各个Listener或者Initializer中来定义自己的配置员的优先级

每个PropertySource有自己的优先级顺序，大致的优先级顺序如下，参考[官方文档](#)

1. 您的主目录上的[Devtools全局设置属性](#)（`~/.spring-boot-devtools.properties` 当devtools处于活动状态时）。
2. [@TestPropertySource](#) 测试中的注释。
3. [@SpringBootTest#properties](#) 测试中的注释属性。
4. 命令行参数。例如：`--user.name=yuandengta-pro`
5. 来自的属性 `SPRING_APPLICATION_JSON`（嵌入在环境变量或系统属性中的嵌入式JSON）。
6. `ServletConfig` 初始化参数。
7. `ServletContext` 初始化参数。
8. 来自的JNDI属性 `java:comp/env`。
9. Java系统属性（`System.getProperties()`）。例如：`-Duser.name=yuandengta-vm`
10. 操作系统环境变量。例如：`USER_NAME=yuandengta-env`
11. 一个 `RandomValuePropertySource`，只有在拥有性能 `random.*`。
12. 打包的jar（`application-{profile}.properties` 和YAML变体）之外的[特定于配置文件的应用程序属性](#)。
13. 打包在jar中的[特定于配置文件的应用程序属性](#)（`application-{profile}.properties` 和YAML变体）。
14. 打包的jar（`application.properties` 和YAML变体）之外的应用程序属性。
15. 打包在jar中的应用程序属性（`application.properties` 和YAML变体）。

16. `@PropertySource` @Configuration 类 上的注释。
17. 默认属性（通过设置指定 `SpringApplication.setDefaultProperties`）。

3. 外部化配置方式

使用@Value进行外部化配置

如何使用Value来进行外部化配置呢？

用于 @Value 注入的方式有以下几种：

我们在下面演示一个使用了@Value构造器注入和默认值支持的使用方式，其中最复杂的是对age的使用，如果说能够找到user.age的配置，那么就使用user.age，否则就使用my.user.age，但是如果my.user.age还是没有结果的话，那么它的默认值就是32：

1. @Value 字段注入（Field Injection）
2. @Value 构造器注入（Constructor Injection）
3. @Value 方法注入（Method Injection）
4. @Value 默认值支持

```
package com.yuandengta.boot;

import com.yuandengta.boot.model.User;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.WebApplicationType;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.annotation.Bean;

@EnableAutoConfiguration
//@Import(User.class)
public class ValueAnnotationBootstrap {

    /*@Bean
    public User user() {
        return new User();
    }*/

    @Bean
    // @Value方法注入
    public User user(@Value("${user.id}") Long id, @Value("${user.name}") String
name) {
        User user = new User();
        user.setId(id);
        user.setName(name);
        return user;
    }

    public static void main(String[] args) {
        ConfigurableApplicationContext context =
            new SpringApplicationBuilder(ValueAnnotationBootstrap.class)
                .web(WebApplicationType.NONE) // 非 web 应用
                .run(args);
    }
}
```

```

        User user = context.getBean("user", User.class);
        //User user = context.getBean(User.class);
        System.err.println("用户对象 : " + user);
        // 关闭上下文
        context.close();
    }
}

```

使用Environment进行外部化配置

获取 Environment Bean，Environment也可以帮助我们获取外部化的配置，其具体使用方式如下：

1. Environment 方法/构造器依赖注入
2. Environment @Autowired 依赖注入
3. EnvironmentAware 接口回调
4. BeanFactory 依赖查找 Environment

执行顺序

- 1) @Autowired
- 2) BeanFactoryAware
- 3) EnvironmentAware

```

@EnableAutoConfiguration
public class EnvironmentAnnotationBootstrap implements EnvironmentAware
/*BeanFactoryAware*/ {

    //通过自动装配获取environment
    //@Autowired
    //private Environment environment;

    //通过EnvironmentAware获取environment
    private Environment environment;

    @Override
    public void setEnvironment(Environment environment) {
        this.environment = environment;
    }

    //通过BeanFactoryAware获取environment
    //private Environment environment;

    /*@Override
    public void setBeanFactory(BeansException {
        this.environment = beanFactory.getBean(Environment.class);
    }*/

    @Bean
    public User user() {
        Long id = environment.getRequiredProperty("user.id", Long.class);
        String name = environment.getRequiredProperty("user.name");
        User user = new User();
        user.setId(id);
        user.setName(name);
    }
}

```

```

        return user;
    }

    public static void main(String[] args) {
        ConfigurableApplicationContext context = new
        SpringApplicationBuilder(EnvironmentAnnotationBootstrap.class)
            .web(WebApplicationType.NONE)
            .run(args);
        User user = context.getBean("user", User.class);
        System.out.println("用户对象: " + user);
        context.close();
    }
}

```

使用 @ConfigurationProperties进行外部化配置

1. @ConfigurationProperties 类级别标注
2. @ConfigurationProperties @Bean 方法声明
注：假如@ConfigurationProperties 需要注解的类是第三方jar包，则需要用到方法声明
3. @ConfigurationProperties 嵌套类型绑定

松散绑定，优先级配置

Java System Properties: -Denable.user.bean_init=false

OS Environment Variables: ENABLE_USER_BEAN_INIT=false

application.properties: enable.user.bean-init=true

使用 @ConditionalOnProperty 判断

@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。

```

@EnableAutoConfiguration
//类级别标注，需要配合@Bean和@ConfigurationProperties注解使用
//@EnableConfigurationProperties

//嵌套类型绑定，需要使用ConfigurationProperties，但是不需要使用@Bean注解，getBean的时候使用ClassType
@EnableConfigurationProperties(User.class)
public class ConfigurationPropertiesBootstrap {

    /* @Bean
    //@ConfigurationProperties(prefix = "user") //方法声明
    public User user() {
        return new User();
    }*/

    public static void main(String[] args) {
        ConfigurableApplicationContext context = new
        SpringApplicationBuilder(ConfigurationPropertiesBootstrap.class)
            .web(WebApplicationType.NONE)
            .run(args);
    }
}

```

```

        //User user = context.getBean("user", User.class);
        User user = context.getBean(User.class); //嵌套类型绑定, getBean的时候使用
        ClassType
        //System.out.println("用户对象: " + user);
        context.close();
    }
}

```

三、Environment源码分析

前面 springBoot 启动流程中, 我们提到了有个 prepareEnvironment 方法:

```

//详细环境的准备
private ConfigurableEnvironment prepareEnvironment(SpringApplicationRunListeners
listeners,
                                ApplicationArguments
applicationArguments) {
    // 获取或者创建应用环境
    ConfigurableEnvironment environment = getOrCreateEnvironment();
    // 配置应用环境, 配置propertySource和activeProfiles
    configureEnvironment(environment, applicationArguments.getSourceArgs());
    //listeners环境准备, 广播ApplicationEnvironmentPreparedEvent
    ConfigurationPropertySources.attach(environment);
    listeners.environmentPrepared(environment);
    //将环境绑定给当前应用程序
    bindToSpringApplication(environment);
    //对当前的环境类型进行判断, 如果不一致进行转换
    if (!this.isCustomEnvironment) {
        environment = new
EnvironmentConverter(getClassLoader()).convertEnvironmentIfNecessary(environment
,
                                deduceEnvironmentClass());
    }
    //配置propertySource对它自己的递归依赖
    ConfigurationPropertySources.attach(environment);
    return environment;
}

```

接下来对上面三步进行详细分析:

1、初始化 environment

```

// 获取或者创建应用环境, 根据应用程序的类型可以分为servlet环境、标准环境(特殊的非web环境)和响
应式环境
private ConfigurableEnvironment getOrCreateEnvironment() {
    //存在则直接返回
    if (this.environment != null) {
        return this.environment;
    }
    //根据webApplicationType创建对应的Environment
    switch (this.webApplicationType) {
        case SERVLET:

```



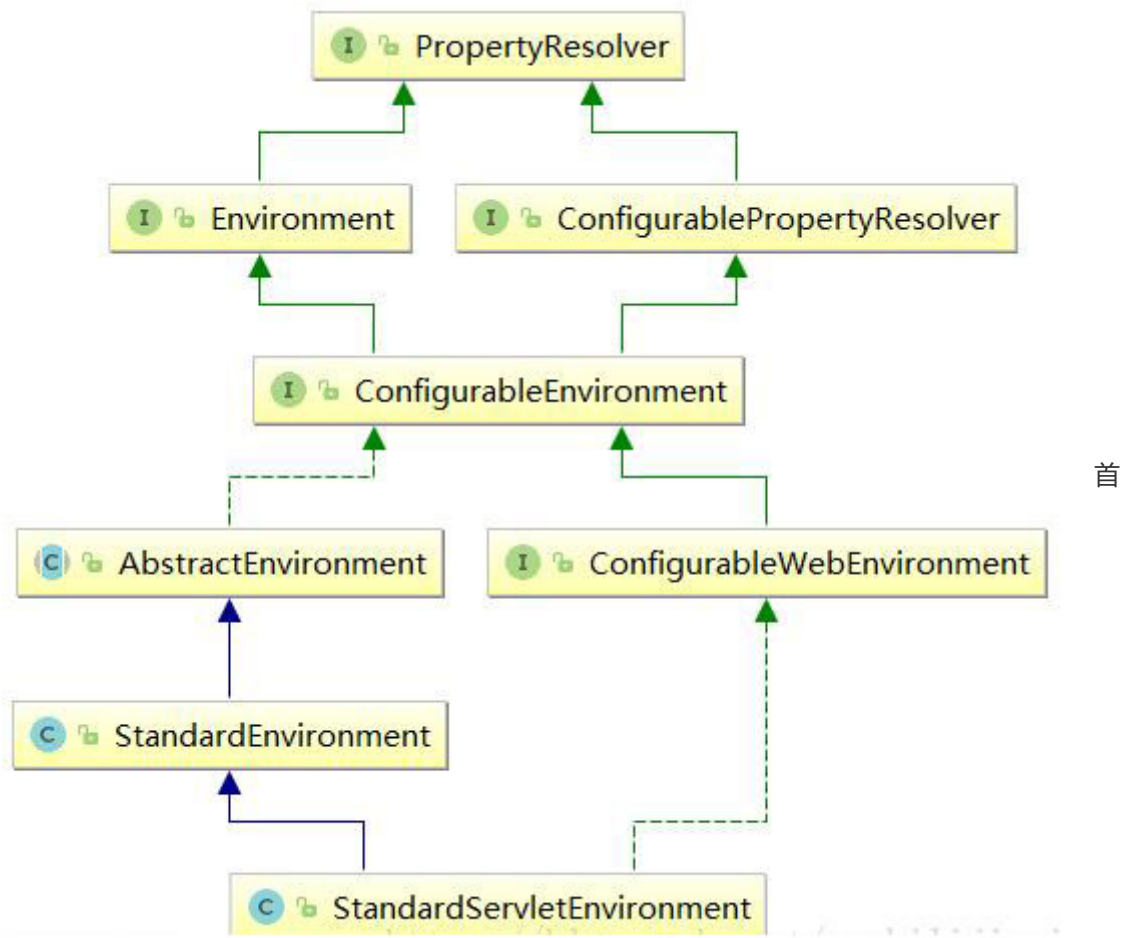
```

        return new StandardServletEnvironment();
    case REACTIVE:
        return new StandardReactiveWebEnvironment();
    default:
        return new StandardEnvironment();
    }
}

```

可以看到根据类型进行匹配 `environment` ,获取到 `StandardServletEnvironment` ,该实例直接注入到spring容器,所以上面示例代码的输出类型就是 `StandardServletEnvironment`。

`StandardServletEnvironment` 是整个 `springboot` 应用运行环境的实现类,后面所有关于配置和环境的操作都基于此类。看一下该类的结构:



先, `StandardServletEnvironment` 的初始化必定会导致父类方法的初始化: `AbstractEnvironment`:

```

public AbstractEnvironment() {
    //从名字可以看出加载我们的自定义配置文件
    customizePropertySources(this.propertySources);
    if (logger.isDebugEnabled()) {
        logger.debug("Initialized " + getClass().getSimpleName() + " with
PropertySources " + this.propertySources);
    }
}

```

在构造方法中调用自定义配置文件, `spring` 的一贯做法, 模板模式, 调用的是实例对象的自定义逻辑:

```

@Override
protected void customizePropertySources(MutablePropertySources
propertySources) {
    propertySources.addLast(new
StubPropertySource(SERVLET_CONFIG_PROPERTY_SOURCE_NAME));
    propertySources.addLast(new
StubPropertySource(SERVLET_CONTEXT_PROPERTY_SOURCE_NAME));
    if (JndiLocatorDelegate.isDefaultJndiEnvironmentAvailable()) {
        propertySources.addLast(new
JndiPropertySource(JNDI_PROPERTY_SOURCE_NAME));
    }
    super.customizePropertySources(propertySources);
}

```

因为该配置类是基于web环境，所以先加载和 servlet有关的参数，`addLast` 放在最后：

```

/** System environment property source name: {@value} */
public static final String SYSTEM_ENVIRONMENT_PROPERTY_SOURCE_NAME =
"systemEnvironment";

/** JVM system properties property source name: {@value} */
public static final String SYSTEM_PROPERTIES_PROPERTY_SOURCE_NAME =
"systemProperties";

```

该类又是对 `StandardEnvironment` 的扩展，这里会调用

```
super.customizePropertySources(propertySources);
```

```

@Override
protected void customizePropertySources(MutablePropertySources
propertySources) {
    propertySources.addLast(new
MapPropertySource(SYSTEM_PROPERTIES_PROPERTY_SOURCE_NAME,
getSystemProperties()));
    propertySources.addLast(new
SystemEnvironmentPropertySource(SYSTEM_ENVIRONMENT_PROPERTY_SOURCE_NAME,
getSystemEnvironment()));
}
/** System environment property source name: {@value} */
public static final String SYSTEM_ENVIRONMENT_PROPERTY_SOURCE_NAME =
"systemEnvironment";

/** JVM system properties property source name: {@value} */
public static final String SYSTEM_PROPERTIES_PROPERTY_SOURCE_NAME =
"systemProperties";

```

可以看到放入顺序是永远放在最后面，也就是先加入的在前面。`systemEnvironment` 是在 `systemProperties` 前面，这点很重要。因为前面的配置会覆盖后面的配置，也就是说系统变量中的配置比系统环境变量中的配置优先级更高。如下：

```

✓ environment = {StandardServletEnvironment@2504} "StandardServletEnvironment {activeProfiles=[], defaultProfiles=[default], pr
> f logger = {LogFactory$Log4jLog@2511}
  f activeProfiles = {LinkedHashSet@2512} size = 0
> f defaultProfiles = {LinkedHashSet@2513} size = 1
✓ f propertySources = {MutablePropertySources@2506} "[StubPropertySource {name='servletConfigInitParams'}, StubPropertySc
> f logger = {LogFactory$Log4jLog@2511}
  propertySourceList = {CopyOnWriteArrayList@2801} size = 4
    0 = {PropertySource$StubPropertySource@2807} "StubPropertySource {name='servletConfigInitParams'}"
    1 = {PropertySource$StubPropertySource@2808} "StubPropertySource {name='servletContextInitParams'}"
    2 = {MapPropertySource@2809} "MapPropertySource {name='systemProperties'}"
    3 = {SystemEnvironmentPropertySource@2810} "SystemEnvironmentPropertySource {name='systemEnvironment'}"
  f propertyResolver = {PropertySourcesPropertyResolver@2517}

```

2、加载默认配置

```

protected void configureEnvironment(ConfigurableEnvironment environment, String[]
args) {
    //加载启动命令行配置属性
    configurePropertySources(environment, args);
    //设置active属性, 用于切换开发/测试/生产环境
    configureProfiles(environment, args);
}

```

这里接收的参数是 `ConfigurableEnvironment`, 也就是 `StandardServletEnvironment` 的父类。继续跟进 `configurePropertySources` 方法:

```

protected void configurePropertySources(ConfigurableEnvironment
environment, String[] args) {
    //获取配置存储集合
    MutablePropertySources sources = environment.getPropertySources();
    //判断是否有默认配置, 默认为空
    if (this.defaultProperties != null && !this.defaultProperties.isEmpty())
    {
        sources.addLast(
            new MapPropertySource("defaultProperties",
this.defaultProperties));
    }
    //加载命令行配置
    if (this.addCommandLineProperties && args.length > 0) {
        String name =
CommandLinePropertySource.COMMAND_LINE_PROPERTY_SOURCE_NAME;
        if (sources.contains(name)) {
            PropertySource<?> source = sources.get(name);
            CompositePropertySource composite = new
CompositePropertySource(name);
            composite.addPropertySource(new SimpleCommandLinePropertySource(
"springApplicationCommandLineArgs", args));
            composite.addPropertySource(source);
            sources.replace(name, composite);
        }
        else {
            sources.addFirst(new SimpleCommandLinePropertySource(args));
        }
    }
}


```

上述代码主要做两件事：一是判断 `SpringBootApplication` 是否指定了默认配置，二是加载默认的命令行配置。

上面有个核心关键类出现了，`MutablePropertySources`，`mutable`中文是可变的意思，该类封装了属性资源集合：

```
public class MutablePropertySources implements PropertySources {
    private final Log logger;
    private final List<PropertySource<?>> propertySourceList = new
    CopyOnWriteArrayList<>();
}
```

该类又是如何使用的呢？



```
104
105     private final Set<String> activeProfiles = new LinkedHashSet<>();
106
107     private final Set<String> defaultProfiles = new LinkedHashSet<>(getReservedDefaultProfiles());
108
109     private final MutablePropertySources propertySources = new MutablePropertySources(this.logger);
110
111     private final ConfigurablePropertyResolver propertyResolver =
112         new PropertySourcesPropertyResolver(this.propertySources);
113
114
```

这里的设计很巧妙，将 `MutablePropertySources` 传递到文件解析器 `propertyResolver` 中，同时 `AbstractEnvironment` 又实现了文件解析接口 `ConfigurablePropertyResolver`，所以 `AbstractEnvironment` 就有了文件解析的功能。所以 `StandardServletEnvironment` 文件解析功能实际委托给了 `PropertySourcesPropertyResolver` 来实现。

继续看一下 `configureProfiles(environment, args);` 方法：

```
protected void configureProfiles(ConfigurableEnvironment environment,
String[] args) {
    environment.getActiveProfiles(); // ensure they are initialized
    // But these ones should go first (last wins in a property key clash)
    Set<String> profiles = new LinkedHashSet<>(this.additionalProfiles);
    profiles.addAll(Arrays.asList(environment.getActiveProfiles()));
    environment.setActiveProfiles(StringUtils.toStringArray(profiles));
}
```

该方法主要将 `SpringBootApplication` 中指定的 `additionalProfiles` 文件加载到 `environment` 中，一般默认为空。该变量的用法，在项目启动类中，需要显示创建 `SpringApplication` 实例，如下：

```
SpringApplication springApplication = new
SpringApplication(MyApplication.class);
//设置profile变量
springApplication.setAdditionalProfiles("prd");
springApplication.run(MyApplication.class, args);
```

3、通知环境监听器，加载项目中的配置文件

触发监听器：

```
listeners.environmentPrepared(environment);
```

在SpringBoot启动流程中，有方法通知的监听器，和配置文件有关的监听器类型为 `ConfigFileApplicationListener`，监听到事件时执行的方法：

```
@Override
public void postProcessEnvironment(ConfigurableEnvironment environment,
    SpringApplication application) {
    //加载项目中的配置文件
    addPropertySources(environment, application.getResourceLoader());
    configureIgnoreBeanInfo(environment);
    bindToSpringApplication(environment, application);
}
```

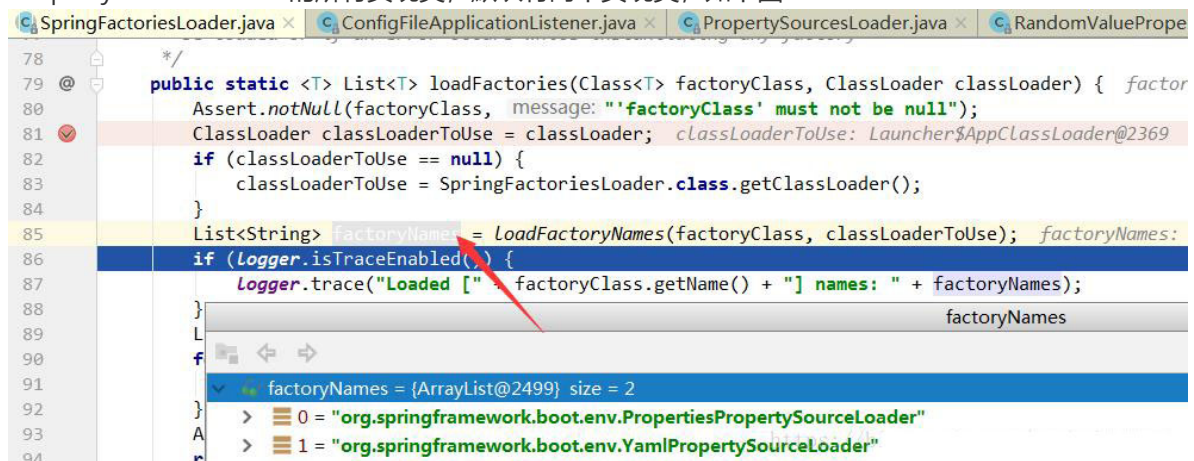
继续跟进去，会发现一个核心内部类 `Loader`，配置文件加载也就委托给该内部类来处理：

```
private class Loader {

    private final Log logger = ConfigFileApplicationListener.this.logger;
    //当前环境
    private final ConfigurableEnvironment environment;
    //类加载器，可以在项目启动时通过 SpringApplication 构造方法指定，默认采用
    Launcher.AppClassLoader加载器
    private final ResourceLoader resourceLoader;
    //资源加载工具类
    private final List<PropertySourceLoader> propertySourceLoaders;
    //LIFO队列
    private Queue<String> profiles;
    //已处理过的文件
    private List<String> processedProfiles;
    private boolean activatedProfiles;

    Loader(ConfigurableEnvironment environment, ResourceLoader
    resourceLoader) {
        this.environment = environment;
        //获取类加载器
        this.resourceLoader = resourceLoader == null ? new
        DefaultResourceLoader()
            : resourceLoader;
        //获取propertySourceLoaders
        this.propertySourceLoaders = SpringFactoriesLoader.loadFactories(
            PropertySourceLoader.class, getClass().getClassLoader());
    }
    //.....
}
```

上面 propertySourceLoaders 通过 SpringFactoriesLoader 获取当前项目中类型为 PropertySourceLoader 的所有实现类，默认有两个实现类，如下图：



```
78  */
79  @
80  public static <T> List<T> loadFactories(Class<T> factoryClass, ClassLoader classLoader) { factor
81      Assert.notNull(factoryClass, message: "'factoryClass' must not be null");
82      ClassLoader classLoaderToUse = classLoader; classLoaderToUse: Launcher$AppClassLoader@2369
83      if (classLoaderToUse == null) {
84          classLoaderToUse = SpringFactoriesLoader.class.getClassLoader();
85      }
86      List<String> factoryNames = LoadFactoryNames(factoryClass, classLoaderToUse); factoryNames:
87      if (Logger.isTraceEnabled()) {
88          logger.trace("Loaded [" + factoryClass.getName() + "] names: " + factoryNames);
89      }
90      factoryNames
91      factoryNames = {ArrayList@2499} size = 2
92      > 0 = "org.springframework.boot.env.PropertiesPropertySourceLoader"
93      > 1 = "org.springframework.boot.env.YamlPropertySourceLoader"
```

继续来看主要解析方法：load()：

```
public void load() {
    this.profiles = Collections.asLifoQueue(new LinkedList<Profile>());
    this.processedProfiles = new LinkedList<>();
    this.activatedProfiles = false;
    this.loaded = new LinkedHashMap<>();
    //初始化逻辑
    initializeProfiles();
    //定位解析资源文件
    while (!this.profiles.isEmpty()) {
        Profile profile = this.profiles.poll();
        load(profile, this::getPositiveProfileFilter,
            addToLoaded(MutablePropertySources::addLast, false));
        this.processedProfiles.add(profile);
    }
    //对加载过的配置文件进行排序
    load(null, this::getNegativeProfileFilter,
        addToLoaded(MutablePropertySources::addFirst, true));
    addLoadedPropertySources();
}
```

跟进去上面初始化方法：

```
private void initializeProfiles() {
    Set<Profile> initialActiveProfiles = initializeActiveProfiles();

    this.profiles.addAll(getUnprocessedActiveProfiles(initialActiveProfiles));
    //如果为空，添加默认的profile
    if (this.profiles.isEmpty()) {
        for (String defaultProfileName :
            this.environment.getDefaultProfiles()) {
            Profile defaultProfile = new Profile(defaultProfileName,
                true);
            if (!this.profiles.contains(defaultProfile)) {
                this.profiles.add(defaultProfile);
            }
        }
    }
    // The default profile for these purposes is represented as null. we
    add it
```



```

        // last so that it is first out of the queue (active profiles will
then
        // override any settings in the defaults when the list is reversed
later).

        //这里添加一个为null的profile，主要是加载默认的配置文
        this.profiles.add(null);
    }

```

上面主要做了两件事情：

1) 判断是否指定了profile，如果没有，添加默认环境：default。后面的解析流程会解析 default 文件，比如： application-default.yml、application-default.properties。

注意：在第2步中我们提到了 additionalProfiles 属性，如果我们通过该属性指定了profile，这里就不会加载默认的配置文，根据我们指定的profile进行匹配。

2) 添加一个null的profile，主要用来加载没有指定profile的配置文，比如： application.properties 因为 profiles 采用了 LIFO 队列，后进先出。所以会先加载profile为null的配置文，也就是匹配 application.properties、application.yml。

继续跟进解析方法 load：

```

private void load(Profile profile, DocumentFilterFactory filterFactory,
    DocumentConsumer consumer) {
    //获取默认的配置文路径
    getSearchLocations().forEach((location) -> {
        boolean isFolder = location.endsWith("/");
        Set<String> names = (isFolder ? getSearchNames() : NO_SEARCH_NAMES);
        //循环加载
        names.forEach(
            (name) -> load(location, name, profile, filterFactory, consumer));
    });
}

```

可以看到springBoot2.0底层的新的改动都是基于 lambda 表达式实现。

继续跟进 getSearchLocations() 方法：

```

private Set<String> getSearchLocations() {
    if (this.environment.containsProperty(CONFIG_LOCATION_PROPERTY)) { environ
        return getSearchLocations(CONFIG_LOCATION_PROPERTY);
    }
    Set<String> locations = getSearchLocations( locations: size = 4
        CONFIG_ADDITIONAL_LOCATION_PROPERTY);
    locations.addAll(
        asResolvedSet(ConfigFileApplicationListener.this.searchLocations,
            DEFAULT_SEARCH_LOCATIONS));
    return locations; locations: size = 4
}

```

locations

```

locations = {LinkedHashSet@3394} size = 4
> 0 = "file:./config/"
> 1 = "file:./"
> 2 = "classpath:/config/"
> 3 = "classpath:/"

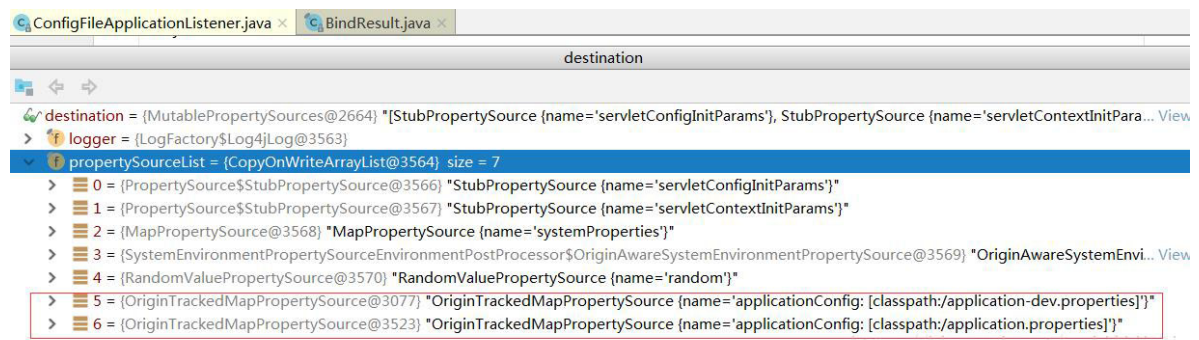
```

获取路径之后，会拼接配置文件名称，选择合适的 yml 或者 properties 解析器进行解析：(name) -> load(location, name, profile, filterFactory, consumer) 具体的解析逻辑比较简单，我们来梳理一下：

1) 获取默认的配置文件路径，有4种。 2) 遍历所有的路径，拼装配件名称。 3) 再遍历解析器，选择 yml 或者 properties 解析，将解析结果添加到集合 MutablePropertySources 当中。

最后解析的结果如下：

```
private void addLoadedPropertySources() {
    MutablePropertySources destination = this.environment.getPropertySources();
    List<MutablePropertySources> loaded = new ArrayList<>(this.loaded.values());
    Collections.reverse(loaded);
    String lastAdded = null;
    Set<String> added = new HashSet<>();
    for (MutablePropertySources sources : loaded) {
        for (PropertySource<?> source : sources) {
            if (added.add(source.getName())) {
                //调试模式查看destination
                addLoadedPropertySource(destination, lastAdded, source);
                lastAdded = source.getName();
            }
        }
    }
}
```



至此，springBoot中的资源文件加载完毕，解析顺序从上到下，所以前面的配置文件会覆盖后面的配置文件。可以看到 application.properties 的优先级最低，系统变量和环境变量的优先级相对较高。