
回到首页

# 芋道源码 —— 知识星球

我是一段不羁的公告！
记得给艿艿这 3 个项目加油，添加一个 STAR 噢。
https://github.com/YunaiV/SpringBoot-Labs
https://github.com/YunaiV/onemall
https://github.com/YunaiV/ruoyi-vue-pro
2021-01-28
Spring Boot

# 精尽 Spring Boot 源码分析 —— 配置加载

## 1. 概述

在使用 Spring Boot 时，我们可以很方便的在 application.properties 或 application.yml 配置文件中，添加相应的应用所需的配置。那么，究竟 Spring Boot 是如何实现该功能的呢，今儿我们就通过 Spring Boot 的源码，一探究竟！

> 艿艿的高能提示：这篇会非常长，建议胖友保持耐心。另外，最好边调试边看~

## 2. Profiles

在讲配置加载之前，不得不先提下 Spring Profiles 功能。

> 如果不熟悉的胖友，先看看 《详解 Spring 中的 Profile》 文章。
> 关于这一块，之前在 《【死磕 Spring】—— 环境 & 属性：PropertySource、Environment、Profile》 中，已经有详细的源码解析。

Spring Boot 在 Spring Framework 的基础之上，可以手动附加新的 Profile 。在 《精尽 Spring Boot 源码分析 —— SpringApplication》 的 # 。代码如下：

```
// SpringApplication.java

/**
 * 附加的 profiles 的数组
 */
private Set<String> additionalProfiles = new HashSet<>();

protected void configureProfiles(ConfigurableEnvironment environment, String[] args) {
    environment.getActiveProfiles(); // ensure they are initialized 保证已经被初始化
    // 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())); // <X>
    // 设置 activeProfiles
    environment.setActiveProfiles(StringUtils.toStringArray(profiles));
}
```

additionalProfiles 属性，可以设置创建的 SpringApplication 可以附加的 additionalProfiles 属性。
<X> 处，在原有 Spring Framework 中设置的 String Profiles 的基础上，又附加上了 SpringApplication.additionalProfiles 配置的。

芳芳：这个小节，讲的有点绕，胖友辛苦理解下~

# 3. ConfigFileApplicationListener

Spring Boot 实现 application.properties 或 application.yml 配置文件的加载，关键在于 ConfigFileApplicationListener 类。在 《精尽 Spring Boot 源码分析 —— ApplicationListener》 中，我们已经简单介绍过它：

> org.springframework.boot.context.config.ConfigFileApplicationListener ，实现 SmartApplicationListener、Ordered、EnvironmentPostProcessor 接口，实现 Spring Boot 配置文件的加载。
>
> 注意哟，ConfigFileApplicationListener 即是一个 SmartApplicationListener 实现类，又是一个 EnvironmentPostProcessor 实现类。

## 3.1 onApplicationEvent

实现 #onApplicationEvent(ApplicationEvent event) 方法，分别对 ApplicationEnvironmentPreparedEvent、ApplicationPreparedEvent 事件进行处理。代码如下：

```
// ConfigFileApplicationListener.java

@Override
public boolean supportsEventType(Class<? extends ApplicationEvent> eventType) {
    return ApplicationEnvironmentPreparedEvent.class.isAssignableFrom(eventType)
            || ApplicationPreparedEvent.class.isAssignableFrom(eventType);
}

@Override
public void onApplicationEvent(ApplicationEvent event) {
    // <1> 如果是 ApplicationEnvironmentPreparedEvent 事件，说明 Spring 环境准备好了，则执行相应的处理
    if (event instanceof ApplicationEnvironmentPreparedEvent) {
        onApplicationEnvironmentPreparedEvent((ApplicationEnvironmentPreparedEvent) event);
    }
    // <2> 如果是 ApplicationPreparedEvent 事件，说明 Spring 容器初始化好了，则进行相应的处理。
    if (event instanceof ApplicationPreparedEvent) {
        onApplicationPreparedEvent(event);
    }
}
```

<1> 处，如果是 ApplicationEnvironmentPreparedEvent 事件，说明 Spring 环境准备好了，则调用 #onApplicationEnvironmentPreparedEvent(ApplicationEnvironmentPreparedEvent) 方法，执行相应的处理。详细解析，见 「3.2 onApplicationEnvironmentPreparedEvent」 。
<2> 处，如果是 ApplicationPreparedEvent 事件，说明 Spring 容器初始化好了，则调用 #onApplicationPreparedEvent(ApplicationPreparedEvent) 方法，进行相应的处理。详细解析，见 「3.3 onApplicationPreparedEvent」 。

## 3.2 onApplicationEnvironmentPreparedEvent

#onApplicationEnvironmentPreparedEvent(ApplicationEnvironmentPreparedEvent) 方法，处理 ApplicationEnvironmentPreparedEvent 事件。代码如下：

```
// ConfigFileApplicationListener.java

private void onApplicationEnvironmentPreparedEvent(ApplicationEnvironmentPreparedEvent event) {
    // <1.1> 加载指定类型 EnvironmentPostProcessor 对应的，在 `META-INF/spring.factories` 里的类名的数组
    List<EnvironmentPostProcessor> postProcessors = loadPostProcessors();
    // 加入自己
    postProcessors.add(this);
    // 排序 postProcessors 数组
    AnnotationAwareOrderComparator.sort(postProcessors);
    // 遍历 postProcessors 数组，逐个执行。
    for (EnvironmentPostProcessor postProcessor : postProcessors) {
        postProcessor.postProcessEnvironment(event.getEnvironment(), event.getSpringApplication());
    }
}
```

<1.1> 处，调用 #loadPostProcessors() 方法，加载指定类型 EnvironmentPostProcessor 对应的，在 META-INF/spring.factories 里的类名的数组。代码如下：

```
// ConfigFileApplicationListener.java

List<EnvironmentPostProcessor> loadPostProcessors() {
 return SpringFactoriesLoader.loadFactories(EnvironmentPostProcessor.class, getClass().getClassLoader());
}
```

- 默认情况下，返回的是 SystemEnvironmentPropertySourceEnvironmentPostProcessor、SpringApplicationJsonEnvironmentPostProcessor、CloudFoundryVcapEnvironmentPostProcessor 类。

<1.2> 处，加入自己，到 postProcessors 数组中。因为自己也是一个 EnvironmentPostProcessor 实现类。

<2> 处，排序 postProcessors 数组。

<3> 处，遍历 postProcessors 数组，调用 EnvironmentPostProcessor#postProcessEnvironment(ConfigurableEnvironment environment, SpringApplication application) 方法，逐个执行。

那么，我们开始来逐个看看每个 EnvironmentPostProcessor 实现类。考虑到 EnvironmentPostProcessor 应有的独立性（尊严！），我们单独开了 「4. EnvironmentPostProcessor」 小节，所以胖友先一起跳过来看看。

芮芮：这块涉及的逻辑非常多。本文的 4、5、6、7 小节，都和 「3.2 onApplicationEnvironmentPreparedEvent」 有关。

## 3.3 onApplicationPreparedEvent

#onApplicationPreparedEvent(ApplicationEvent event) 方法，处理 ApplicationPreparedEvent 事件。代码

如下：

```
// ConfigFileApplicationListener.java

private void onApplicationPreparedEvent(ApplicationEvent event) {
    // 修改 logger 文件，暂时可以无视。
    this.logger.switchTo(ConfigFileApplicationListener.class);
    // 添加 PropertySourceOrderingPostProcessor 处理器
    addPostProcessors(((ApplicationPreparedEvent) event).getApplicationContext());
}

protected void addPostProcessors(ConfigurableApplicationContext context) {
    context.addBeanFactoryPostProcessor(new PropertySourceOrderingPostProcessor(context));
}
```

添加 PropertySourceOrderingPostProcessor 处理器。关于 PropertySourceOrderingPostProcessor 类，见 「3.3.1 PropertySourceOrderingPostProcessor」 中。

### 3.3.1 PropertySourceOrderingPostProcessor

PropertySourceOrderingPostProcessor ，是 ConfigFileApplicationListener 内部类，实现 BeanFactoryPostProcessor、Ordered 接口，将 DEFAULT_PROPERTIES 的 PropertySource 属性源，添加到 environment 的尾部。代码如下：

```
// ConfigFileApplicationListener.java
private static final String DEFAULT_PROPERTIES = "defaultProperties";

// ConfigFileApplicationListener#PropertySourceOrderingPostProcessor.java

private class PropertySourceOrderingPostProcessor implements BeanFactoryPostProcessor, Ordered {

    private ConfigurableApplicationContext context;

    PropertySourceOrderingPostProcessor(ConfigurableApplicationContext context) {
        this.context = context;
    }

    @Override
    public int getOrder() {
        return Ordered.HIGHEST_PRECEDENCE;
    }

    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {
        reorderSources(this.context.getEnvironment());
    }

    private void reorderSources(ConfigurableEnvironment environment) {
        // 移除 DEFAULT_PROPERTIES 的属性源
        PropertySource<?> defaultProperties = environment.getPropertySources().remove(DEFAULT_PROPERTIES);
        // 添加到 environment 尾部
        if (defaultProperties != null) {
            environment.getPropertySources().addLast(defaultProperties);
        }
    }
```

```
}
```

那么 DEFAULT_PROPERTIES 对应的 PropertySource 属性源，究竟是哪里来的呢？答案见 SpringApplication.defaultProperties 相关，代码如下：

```
// SpringApplication.java

/**
 * 默认的属性集合
 */
private Map<String, Object> defaultProperties;

protected void configurePropertySources(ConfigurableEnvironment environment, String[] args) {
    MutablePropertySources sources = environment.getPropertySources();
 // 配置的 defaultProperties
 if (this.defaultProperties != null && !this.defaultProperties.isEmpty()) {
        sources.addLast(new MapPropertySource("defaultProperties", this.defaultProperties));
    }

 // ... 省略无关代码
}
```

# 4. EnvironmentPostProcessor

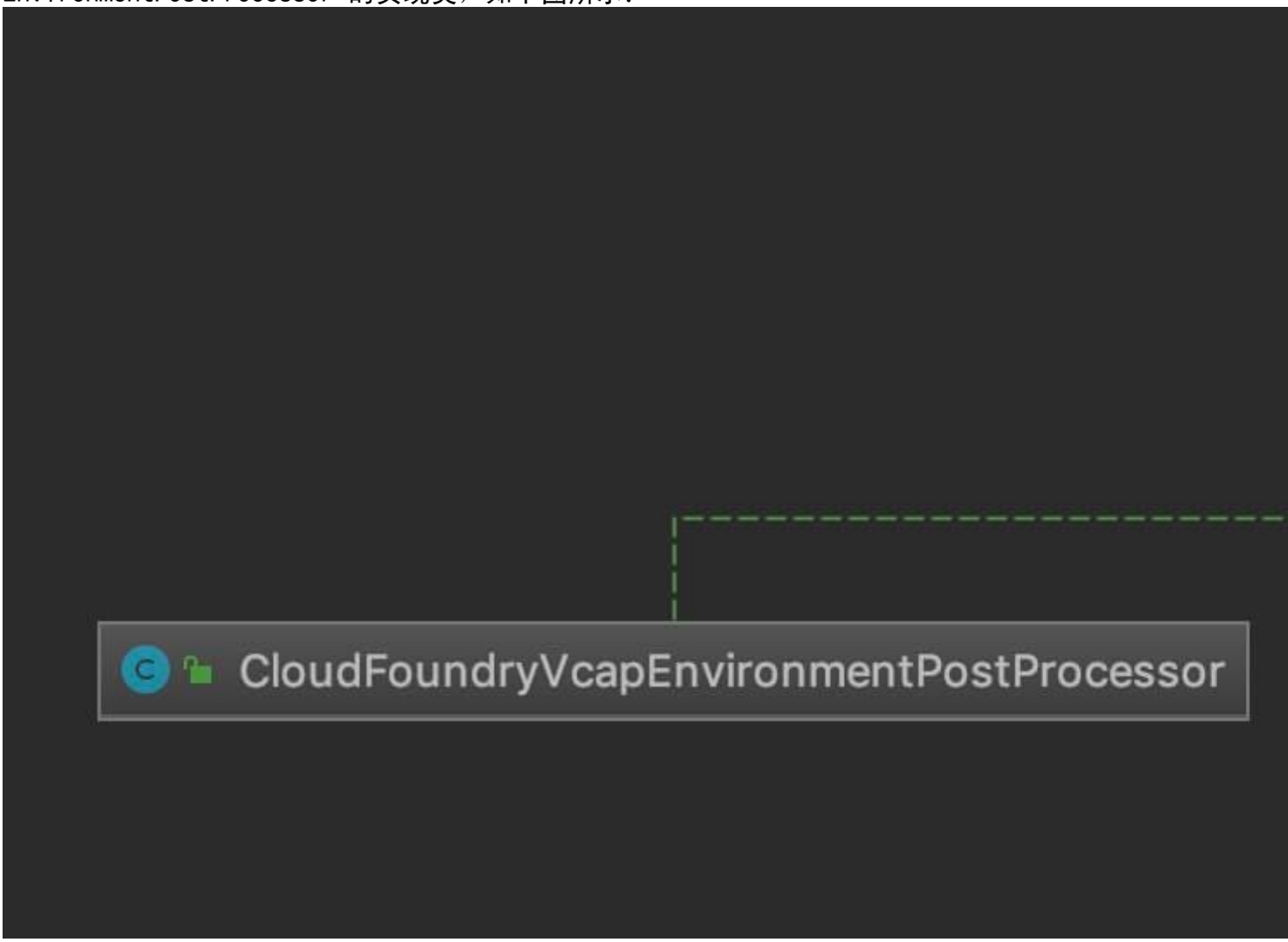
org.springframework.boot.context.config.EnvironmentPostProcessor 接口，在 Environment 加载完成之后，如果我们需要对其进行一些配置、增加一些自己的处理逻辑，那么请使用 EnvironmentPostProcessor 。代码如下：

```
// EnvironmentPostProcessor.java

/**
 * Allows for customization of the application's {@link Environment} prior to the
 * application context being refreshed.
 * <p>
 * EnvironmentPostProcessor implementations have to be registered in
 * {@code META-INF/spring.factories}, using the fully qualified name of this class as the
 * key.
 * <p>
 * {@code EnvironmentPostProcessor} processors are encouraged to detect whether Spring's
 * {@link org.springframework.core.Ordered Ordered} interface has been implemented or if
 * the {@link org.springframework.core.annotation.Order @Order} annotation is present and
 * to sort instances accordingly if so prior to invocation.
 *
 * @author Andy Wilkinson
 * @author Stephane Nicoll
 * @since 1.3.0
 */
@FunctionalInterface
public interface EnvironmentPostProcessor {

 /**
     * Post-process the given {@code environment}.
     * @param environment the environment to post-process
```

```
     * @param application the application to which the environment belongs
     */
    void postProcessEnvironment(ConfigurableEnvironment environment, SpringApplication application);

}
```

从实现上和作用上来说，和 《【死磕 Spring】—— IoC 之深入分析 BeanPostProcessor》 都是非常类似的。

EnvironmentPostProcessor 的实现类，如下图所示：




## 4.1 CloudFoundryVcapEnvironmentPostProcessor

org.springframework.boot.cloud.CloudFoundryVcapEnvironmentPostProcessor ，实现 EnvironmentPostProcessor、Ordered 接口，实现对 Cloud Foundry 的支持。因为我们不使用 Cloud Foundry ，所以可以跳过对 CloudFoundryVcapEnvironmentPostProcessor 源码的了解。感兴趣的胖友，可以结合 《Spring Boot 参考指南（部署到云）》 文章，对源码进行研究。

## 4.2 SystemEnvironmentPropertySourceEnvironmentPostProcessor

芋艿：选看~

org.springframework.boot.env.SystemEnvironmentPropertySourceEnvironmentPostProcessor ，实现 EnvironmentPostProcessor、Ordered 接口，实现将 environment 中的 systemEnvironment 对应的 PropertySource 属性源对象，替换成 OriginAwareSystemEnvironmentPropertySource 对象。代码如下：

```
// SystemEnvironmentPropertySourceEnvironmentPostProcessor.java

@Override
public void postProcessEnvironment(ConfigurableEnvironment environment, SpringApplication application) {
    // <1> 获得 systemEnvironment 对应的 PropertySource 属性源
    String sourceName = StandardEnvironment.SYSTEM_ENVIRONMENT_PROPERTY_SOURCE_NAME;
    PropertySource<?> propertySource = environment.getPropertySources().get(sourceName);
    // <2> 将原始的 PropertySource 对象，替换成 OriginAwareSystemEnvironmentPropertySource 对象
    if (propertySource != null) {
        replacePropertySource(environment, sourceName, propertySource);
    }
}

@SuppressWarnings("unchecked")
private void replacePropertySource(ConfigurableEnvironment environment, String sourceName, PropertySource<?> property
    Map<String, Object> originalSource = (Map<String, Object>) propertySource.getSource();
    // 创建 SystemEnvironmentPropertySource 对象
    SystemEnvironmentPropertySource source = new OriginAwareSystemEnvironmentPropertySource(sourceName, originalSourc
    // 进行替换
    environment.getPropertySources().replace(sourceName, source);
}
```

<1> 处，获得 systemEnvironment 对应的 PropertySource 属性源。
<2> 处，调用 #replacePropertySource(...) 方法，将原始的 PropertySource 对象，替换成 OriginAwareSystemEnvironmentPropertySource 对象。

其中，OriginAwareSystemEnvironmentPropertySource 是 SystemEnvironmentPropertySourceEnvironmentPostProcessor 内部类，继承 SystemEnvironmentPropertySource 类，实现 OriginLookup 接口，代码如下：

```
// SystemEnvironmentPropertySourceEnvironmentPostProcessor#OriginAwareSystemEnvironmentPropertySource.java
protected static class OriginAwareSystemEnvironmentPropertySource
     extends SystemEnvironmentPropertySource implements OriginLookup<String> {

    OriginAwareSystemEnvironmentPropertySource(String name, Map<String, Object> source) {
     super(name, source);
    }

 @Override
 public Origin getOrigin(String key) {
        // 解析 key 对应的 property
        String property = resolvePropertyName(key);
     // 判断是否存在 property 对应的值。如果存在，则返回 SystemEnvironmentOrigin 对象
     if (super.containsProperty(property)) {
         return new SystemEnvironmentOrigin(property);
        }
     // 不存在，则返回 null
     return null;
    }

}
```

- 重心是对 org.springframework.boot.origin.OriginLookup 接口的 #getOrigin(String key) 方法的实现，实现查找 key 对应的真正的 property 。难道两者还会不同，答案是的。例如说：传入的 key=foo.bar.baz ，返回的是 property=FOO_BAR_BAZ 。
- 答案见 SystemEnvironmentPropertySource#checkPropertyName(String name) 方法，内部会进行各种灵活的替换“查找”。代码如下：

```
// SystemEnvironmentPropertySource.java

@Nullable
private String checkPropertyName(String name) {
 // Check name as-is
 if (containsKey(name)) {
     return name;
    }
 // Check name with just dots replaced
    String noDotName = name.replace('.', '_');
 if (!name.equals(noDotName) && containsKey(noDotName)) {
     return noDotName;
    }
 // Check name with just hyphens replaced
    String noHyphenName = name.replace('-', '_');
 if (!name.equals(noHyphenName) && containsKey(noHyphenName)) {
     return noHyphenName;
    }
 // Check name with dots and hyphens replaced
    String noDotNoHyphenName = noDotName.replace('-', '_');
 if (!noDotName.equals(noDotNoHyphenName) && containsKey(noDotNoHyphenName)) {
     return noDotNoHyphenName;
    }
 // Give up
 return null;
}
```

- 各种符号，转成 _ 来查找对应的属性。

那么具体有什么样的逻辑上的需要呢？暂时还没怎么看到，嘿嘿。所以，知道就好，暂时先不去深究。

## 4.3 SpringApplicationJsonEnvironmentPostProcessor

芳芳：选看~

org.springframework.boot.env.SpringApplicationJsonEnvironmentPostProcessor ，实现 EnvironmentPostProcessor、Ordered 接口，解析 environment 中的 spring.application.json 或 SPRING_APPLICATION_JSON 对应的 JSON 格式的属性值，创建新的 PropertySource 对象，添加到其中。代码如下：

```
// SpringApplicationJsonEnvironmentPostProcessor.java

/**
 * 默认的 {@link #order} 的值
 *
 * The default order for the processor.
 */
```

```
public static final int DEFAULT_ORDER = Ordered.HIGHEST_PRECEDENCE + 5;

/**
 * 顺序
 */
private int order = DEFAULT_ORDER;

@Override
public void postProcessEnvironment(ConfigurableEnvironment environment, SpringApplication application) {
    MutablePropertySources propertySources = environment.getPropertySources();
    propertySources.stream().map(JsonPropertyValue::get).filter(Objects::nonNull)
            .findFirst().ifPresent((v) -> processJson(environment, v));
}
```

map(JsonPropertyValue::get).filter(Objects::nonNull) 代码段，调用 JsonPropertyValue#get(PropertySource<?> propertySource) 方法，environment 中的 spring.application.json 或 SPRING_APPLICATION_JSON 对应的 JSON 格式的属性值。代码如下：

```
// SpringApplicationJsonEnvironmentPostProcessor.java

public static final String SPRING_APPLICATION_JSON_PROPERTY = "spring.application.json";
public static final String SPRING_APPLICATION_JSON_ENVIRONMENT_VARIABLE = "SPRING_APPLICATION_JSON";

private static class JsonPropertyValue {

    private static final String[] CANDIDATES = { SPRING_APPLICATION_JSON_PROPERTY, SPRING_APPLICATION_JSON_ENV

    private final PropertySource<?> propertySource;

    private final String propertyName;

    private final String json;

    JsonPropertyValue(PropertySource<?> propertySource, String propertyName, String json) {
        this.propertySource = propertySource;
        this.propertyName = propertyName;
        this.json = json;
    }

    public String getJson() {
        return this.json;
    }

    public Origin getOrigin() {
        return PropertySourceOrigin.get(this.propertySource, this.propertyName);
    }

    public static JsonPropertyValue get(PropertySource<?> propertySource) {
        // 遍历 CANDIDATES 数组
        for (String candidate : CANDIDATES) {
            // 获得 candidate 对应的属性值
            Object value = propertySource.getProperty(candidate);
            if (value instanceof String
                    && StringUtils.hasLength((String) value)) {
                // 创建 JsonPropertyValue 对象，然后返回
                return new JsonPropertyValue(propertySource, candidate, (String) value);
            }
        }
```

```
            return null;
        }

    }
```

- 比较简单，胖友看下就明白列。

调用 `#processJson(ConfigurableEnvironment environment, JsonPropertyValue propertyValue)` 方法，执行处理 JSON 字符串。详细解析，见 「4.3.1 processJson」 。

## 4.3.1 processJson

`#processJson(ConfigurableEnvironment environment, JsonPropertyValue propertyValue)` 方法，执行处理 JSON 字符串。代码如下：

```
// SpringApplicationJsonEnvironmentPostProcessor.java

private void processJson(ConfigurableEnvironment environment, JsonPropertyValue propertyValue) {
    // <1> 解析 json 字符串，成 Map 对象
    JsonParser parser = JsonParserFactory.getJsonParser();
    Map<String, Object> map = parser.parseMap(propertyValue.getJson());
    // <2> 创建 JsonPropertySource 对象，添加到 environment 中
    if (!map.isEmpty()) {
        // <2.2>
        addJsonPropertySource(environment, new JsonPropertySource(propertyValue, flatten(map))); // <2.1>
    }
}
```

<1> 处，解析 json 字符串，成 Map 对象。

<2> 处，创建 JsonPropertySource 对象，添加到 `environment` 中。其中，JsonPropertySource 继承 MapPropertySource 类，实现 OriginLookup 接口，代码如下：

```
    // SpringApplicationJsonEnvironmentPostProcessor#JsonPropertySource.java

    private static class JsonPropertySource extends MapPropertySource
         implements OriginLookup<String> {

     private final JsonPropertyValue propertyValue;

        JsonPropertySource(JsonPropertyValue propertyValue, Map<String, Object> source) {
         super(SPRING_APPLICATION_JSON_PROPERTY, source);
         this.propertyValue = propertyValue;
        }

     @Override
     public Origin getOrigin(String key) {
         return this.propertyValue.getOrigin();
        }

    }
```

- 使用的 `name` 为 `SPRING_APPLICATION_JSON_PROPERTY=spring.application.json` 。

<2.1> 处，调用 #flatten(String prefix, Map<String, Object> result, Map<String, Object> map) 方法，将 JSON 解析后的 Map 可能存在的内嵌的 Map 对象，转换成多条 KV 格式的配置对。代码如下：

```
// SpringApplicationJsonEnvironmentPostProcessor.java

private Map<String, Object> flatten(Map<String, Object> map) {
    Map<String, Object> result = new LinkedHashMap<>();
    flatten(null, result, map);
    return result;
}

private void flatten(String prefix, Map<String, Object> result, Map<String, Object> map) {
    String namePrefix = (prefix != null) ? prefix + "." : "";
    map.forEach((key, value) -> extract(namePrefix + key, result, value));
}

@SuppressWarnings("unchecked")
private void extract(String name, Map<String, Object> result, Object value) {
    if (value instanceof Map) { // 内嵌的 Map 格式
        flatten(name, result, (Map<String, Object>) value);
    } else if (value instanceof Collection) { // 内嵌的 Collection
        int index = 0;
        for (Object object : (Collection<Object>) value) {
            extract(name + "[" + index + "]", result, object);
            index++;
        }
    } else { // 普通格式，添加到 result 中
        result.put(name, value);
    }
}
```

<2.2> 处，调用 #addJsonPropertySource(ConfigurableEnvironment environment, PropertySource<?> source) 方法，添加到 environment 中。代码如下：

```
// SpringApplicationJsonEnvironmentPostProcessor.java

private void addJsonPropertySource(ConfigurableEnvironment environment, PropertySource<?> source) {
    MutablePropertySources sources = environment.getPropertySources();
    // 获得需要添加到 source 所在的 PropertySource 之前的名字
    String name = findPropertySource(sources);
    // 添加到 environment 的 sources 中。
    // 这么做的效果是，source 高于 SYSTEM_PROPERTIES_PROPERTY_SOURCE_NAME && JNDI_PROPERTY_SOURCE_NAME 之前
    if (sources.contains(name)) {
        sources.addBefore(name, source);
    } else {
        sources.addFirst(source);
    }
}

private String findPropertySource(MutablePropertySources sources) {
    // 在 Servlet 环境下，且有 JNDI_PROPERTY_SOURCE_NAME 属性，则返回 StandardServletEnvironment.JNDI_PROPERTY
    if (ClassUtils.isPresent(SERVLET_ENVIRONMENT_CLASS, null)
            && sources.contains(StandardServletEnvironment.JNDI_PROPERTY_SOURCE_NAME)) {
        return StandardServletEnvironment.JNDI_PROPERTY_SOURCE_NAME;

    }
```

```
            // 否则，返回 SYSTEM_PROPERTIES_PROPERTY_SOURCE_NAME 属性
            return StandardEnvironment.SYSTEM_PROPERTIES_PROPERTY_SOURCE_NAME;
        }
```

当然，因为绝大多数情况下，我们并不会去使用 `spring.application.json` 或 `SPRING_APPLICATION_JSON` 去进行配置。所以呢，这块逻辑等胖友真的有需要，再来瞅瞅落。

## 4.4 ConfigFileApplicationListener

芮芮：真正的重头戏~

实现 `#addPropertySources(ConfigurableEnvironment environment,ResourceLoader resourceLoader)` 方法，代码如下：

```
// ConfigFileApplicationListener.java

protected void addPropertySources(ConfigurableEnvironment environment,ResourceLoader resourceLoader) {
    // <1> 添加 RandomValuePropertySource 到 environment 中
    RandomValuePropertySource.addToEnvironment(environment);
    // <2> 创建 Loader 对象，进行加载
    new Loader(environment, resourceLoader).load();
}
```

<1> 处，添加 RandomValuePropertySource 到 `environment` 中。详细解析，见 「5. RandomValuePropertySource」 中。
<2> 处，创建 Loader 对象，并调用 `Loader#load()` 方法，进行加载。详细解析，见 「4.4.1 Loader」 中。

### 4.4.1 Loader

芮芮：高能预警，关于这一块，内容会比较扎实（很多）！

Loader 是 ConfigFileApplicationListener 的内部类，负责加载指定的配置文件。构造方法如下：

```
// ConfigFileApplicationListener.java

private class Loader {

    private final Log logger = ConfigFileApplicationListener.this.logger;

    private final ConfigurableEnvironment environment;

    private final PropertySourcesPlaceholdersResolver placeholdersResolver;

    private final ResourceLoader resourceLoader;

    private final List<PropertySourceLoader> propertySourceLoaders;

    private Deque<Profile> profiles;

    private List<Profile> processedProfiles;

    private boolean activatedProfiles;
```

```
        private Map<Profile, MutablePropertySources> loaded;

        private Map<DocumentsCacheKey, List<Document>> loadDocumentsCache = new HashMap<>();

        Loader(ConfigurableEnvironment environment, ResourceLoader resourceLoader) {
            this.environment = environment;
            // <1> 创建 PropertySourcesPlaceholdersResolver 对象
            this.placeholdersResolver = new PropertySourcesPlaceholdersResolver(this.environment);
            // <2> 创建 DefaultResourceLoader 对象
            this.resourceLoader = (resourceLoader != null) ? resourceLoader : new DefaultResourceLoader();
            // <3> 加载指定类型 PropertySourceLoader 对应的，在 `META-INF/spring.factories` 里的类名的数组
            this.propertySourceLoaders = SpringFactoriesLoader.loadFactories(PropertySourceLoader.class, getClass().getCl
        }

        // ... 省略其它方法
    }
```

<1> 处，创建 PropertySourcesPlaceholdersResolver 对象。详细解析，胖友可以跳到 「6. PropertySourcesPlaceholdersResolver」 中，瞅一眼，然后继续回到此处。
<2> 处，创建 DefaultResourceLoader 对象。这个是 Spring Framework 中的类，用于资源的加载。当然，这不是本文的重点，暂时先忽略。
<3> 处，加载指定类型 PropertySourceLoader 对应的，在 `META-INF/spring.factories` 里的类名的数组。
- 默认情况下，返回的是 PropertiesPropertySourceLoader、YamlPropertySourceLoader 类。详细解析，见 「7. PropertySourceLoader」 。

#### 4.4.1.1 load

`#load()` 方法，加载配置。代码如下：

```
    // ConfigFileApplicationListener#Loader.java

    public void load() {
        // <1> 初始化变量
        this.profiles = new LinkedList<>(); // 未处理的 Profile 集合
        this.processedProfiles = new LinkedList<>(); // 已处理的 Profile 集合
        this.activatedProfiles = false;
        this.loaded = new LinkedHashMap<>();
        // <2> 初始化 Spring Profiles 相关
        initializeProfiles();
        // <3> 遍历 profiles 数组
        while (!this.profiles.isEmpty()) {
            // 获得 Profile 对象
            Profile profile = this.profiles.poll();
            // <3.1> 添加 Profile 到 environment 中
            if (profile != null && !profile.isDefaultProfile()) {
                addProfileToEnvironment(profile.getName());
            }
            // <3.2> 加载配置
            load(profile, this::getPositiveProfileFilter,
                    addToLoaded(MutablePropertySources::addLast, false));
            // <3.3> 添加到 processedProfiles 中，表示已处理
            this.processedProfiles.add(profile);
        }
        // <4> 获得真正加载的 Profile 们，添加到 environment 中。
```

```
        resetEnvironmentProfiles(this.processedProfiles);
        // <5> 加载配置
        load(null, this::getNegativeProfileFilter,
                addToLoaded(MutablePropertySources::addFirst, true));
        // <6> 将加载的配置对应的 MutablePropertySources 到 environment 中
        addLoadedPropertySources();
    }
```

<1> 处，初始化变量。每个变量的意思，看代码中的注释。
<2> 处，初始化 Spring Profiles 相关。详细解析，见 「4.4.1.1 initializeProfiles」 中。胖友可以先跳过去看一眼，然后回来。

<3> 处，遍历 profiles 数组，逐个加载对应的配置文件。

- <3.1> 处，调用 #addProfileToEnvironment(String profile) 方法，添加到 environment.activeProfiles 中。代码如下：

```
// ConfigFileApplicationListener#Loader.java

private void addProfileToEnvironment(String profile) {
    // 如果已经在 activeProfiles 中，则返回
    for (String activeProfile : this.environment.getActiveProfiles()) {
        if (activeProfile.equals(profile)) {
            return;
        }
    }
    // 如果不在，则添加到 environment 中
    this.environment.addActiveProfile(profile);
}
```

  - ~

- <3.2> 处，调用 #load(Profile profile, DocumentFilterFactory filterFactory, DocumentConsumer consumer) 方法，加载配置。这块会比较复杂，晚点再求看 「4.4.1.1.2 load」 。
- <3.3> 处，添加到 processedProfiles 中，表示已处理。

<4> 处，调用 #resetEnvironmentProfiles(List<Profile> processedProfiles) 方法，获得真正加载的 Profile 们，添加到 environment 中。代码如下：

```
// ConfigFileApplicationListener#Loader.java

private void resetEnvironmentProfiles(List<Profile> processedProfiles) {
    // 获得真正加载的 Profile 们，添加到 environment 中。
    String[] names = processedProfiles.stream()
            .filter((profile) -> profile != null && !profile.isDefaultProfile())
            .map(Profile::getName).toArray(String[]::new);
    this.environment.setActiveProfiles(names);
}
```

- 因为每个 Profile 可能不存在对应的配置文件，只有真正加载到配置文件的 Profile 们，才会设置到 environment.activeProfiles 属性中。

<5> 处，调用 #load(Profile profile, DocumentFilterFactory filterFactory, DocumentConsumer consumer) 方法

，加载配置。这块会比较复杂，晚点再求看 「4.4.1.1.2 load」 。

- 和 <3.2> 处，有一些不同，主要差别在于传入的方法参数。
- 在补充一点，有点不造怎么解释了。 <5> 处的作用是，将 profile=null 的情况中，将配置文件里有配置文件 spring.profiles 内容，且属于基于的 Profile ，也添加到 profile=null 在 Loader.loaded 的映射中。具体的，可以看 「666. 彩蛋」 提供的示例。真的是，好绕啊！！！！

<6> 处，调用 #addLoadedPropertySources() 方法，将加载的配置对应的 MutablePropertySources 到 environment 中。代码如下：

```
// ConfigFileApplicationListener.java

private static final String DEFAULT_PROPERTIES = "defaultProperties";

// ConfigFileApplicationListener#Loader.java

private void addLoadedPropertySources() {
    MutablePropertySources destination = this.environment.getPropertySources();
    // 获得当前加载的 MutablePropertySources 集合
    List<MutablePropertySources> loaded = new ArrayList<>(this.loaded.values());
    Collections.reverse(loaded); // <Z>
    // 声明变量
    String lastAdded = null; // 下面循环，最后找到的 MutablePropertySources 的名字
    Set<String> added = new HashSet<>(); // 已添加到 destination 中的 MutablePropertySources 的名字的集合
    // <X> 遍历 loaded 数组
    for (MutablePropertySources sources : loaded) {
        // <Y> 遍历 sources 数组
        for (PropertySource<?> source : sources) {
            // 添加到 destination 中
            if (added.add(source.getName())) {
                addLoadedPropertySource(destination, lastAdded, source);
                lastAdded = source.getName();
            }
        }
    }
}

private void addLoadedPropertySource(MutablePropertySources destination, String lastAdded, PropertySource<?> so
    if (lastAdded == null) {
        if (destination.contains(DEFAULT_PROPERTIES)) {
            destination.addBefore(DEFAULT_PROPERTIES, source);
        } else {
            destination.addLast(source);
        }
    } else {
        destination.addAfter(lastAdded, source);
    }
}
```

- <X> 和 <Y> 处，为什么是两层遍历呢？因为一个 Profile 可以对应多个配置文件。例如说，Profile 为 prod ，对应 applicaion-prod.properties 和 application-prod.yml 两个配置文件。
- 这样，我们就可以从 environment 中，读取加载到的配置文件。
- <Z> 处，为什么要反转一下呢？因为，配置在越后面的 Profile ，优先级越高，所以需要进行反转。举个例子 spring.profiles.active=prod,dev ，那么 Profile 的优先级是 dev > prod > null 。

下面，我们就可以跳到 「4.4.1.1.2 load」 小节，看看这个关键的逻辑。

> 芳芳：逻辑真的有点复杂！目的简单，过程中的逻辑细节比较多。和我一起，保持耐心！！！

#### 4.4.1.1.1 initializeProfiles

在 `#initializeProfiles()` 方法之前，我们先来看 Profile 类。它是 ConfigFileApplicationListener 的内部类，是 Spring Profiles 的封装对象。代码如下：

```
// ConfigFileApplicationListener#Profile.java

/**
 * A Spring Profile that can be loaded.
 */
private static class Profile {

    /**
     * Profile 名字
     */
 private final String name;
    /**
     * 是否为默认的 Profile
     */
 private final boolean defaultProfile;

    // ... 省略构造方法、getting 方法、toString 和 hashCode 方法

}
```

然后，继续来看 `#initializeProfiles()` 方法，初始化 Spring Profiles 相关。代码如下：

```
// ConfigFileApplicationListener#Loader.java

private void initializeProfiles() {
    // The default profile for these purposes is represented as null. We add it
    // first so that it is processed first and has lowest priority.
    // <1> 添加 null 到 profiles 中。用于加载默认的配置文件。优先添加到 profiles 中，因为希望默认的配置文件先被处理。
    this.profiles.add(null);
    // <2.1> 获得激活的 Profile 们（从配置中）
    Set<Profile> activatedViaProperty = getProfilesActivatedViaProperty();
    // <2.2> 先添加激活的 Profile 们（不在配置中）
    this.profiles.addAll(getOtherActiveProfiles(activatedViaProperty));
    // Any pre-existing active profiles set via property sources (e.g.
    // System properties) take precedence over those added in config files.
    // <2.3> 再添加激活的 Profile 们（在配置中）
    addActiveProfiles(activatedViaProperty);
    // <3> 如果没有激活的 Profile 们，则添加默认的 Profile
    if (this.profiles.size() == 1) { // only has null profile
        for (String defaultProfileName : this.environment.getDefaultProfiles()) {
            Profile defaultProfile = new Profile(defaultProfileName, true);
            this.profiles.add(defaultProfile);
        }
    }
}
```

<1>

处，添加 null 到 profiles 中。用于加载默认的配置文件。优先添加到 profiles 中，因为希望默认的配置文件先被处理。

<2.1> 处，调用 #getProfilesActivatedViaProperty() 方法，获得激活的 Profile 们（从配置中）。代码如下：

```
// ConfigFileApplicationListener#Loader.java

private Set<Profile> getProfilesActivatedViaProperty() {
 if (!this.environment.containsProperty(ACTIVE_PROFILES_PROPERTY)
            && !this.environment.containsProperty(INCLUDE_PROFILES_PROPERTY)) {
     return Collections.emptySet();
    }
    Binder binder = Binder.get(this.environment);
    Set<Profile> activeProfiles = new LinkedHashSet<>();
    activeProfiles.addAll(getProfiles(binder, INCLUDE_PROFILES_PROPERTY)); // spring.profiles.include
    activeProfiles.addAll(getProfiles(binder, ACTIVE_PROFILES_PROPERTY)); // spring.profiles.active
 return activeProfiles;
}
```

- 读取 "spring.profiles.include" 和 "spring.profiles.active" 对应的 Profile 们。

<2.2> 处，调用 #getOtherActiveProfiles(Set<Profile> activatedViaProperty) 方法，先添加激活的 Profile 们（不在配置中）到 profiles 中。代码如下：

```
// ConfigFileApplicationListener#Loader.java

private List<Profile> getOtherActiveProfiles(Set<Profile> activatedViaProperty) {
 return Arrays.stream(this.environment.getActiveProfiles()).map(Profile::new)
            .filter((profile) -> !activatedViaProperty.contains(profile))
            .collect(Collectors.toList());
}
```

- “不在配置中”，例如说：SpringApplication.additionalProfiles 。

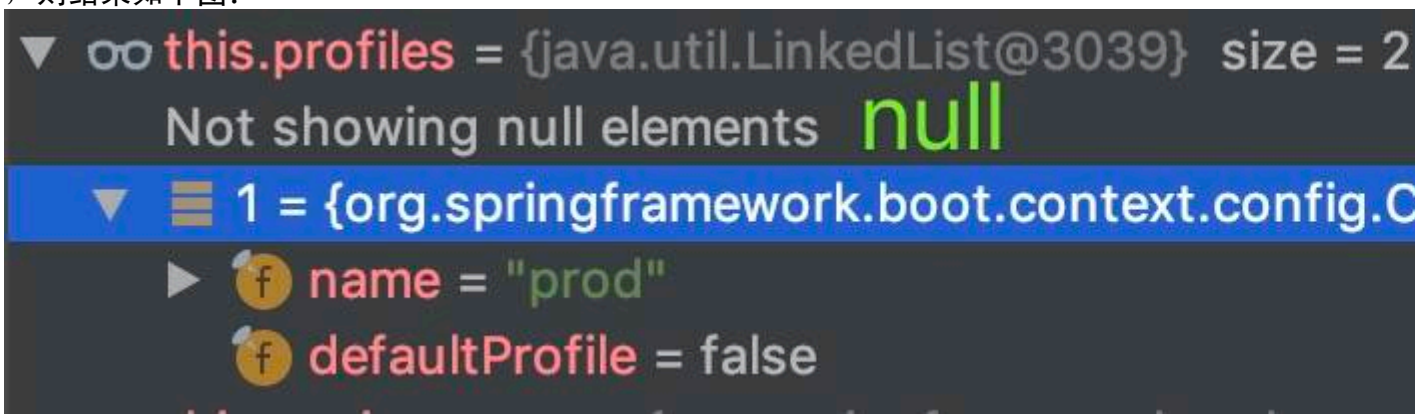
<2.3> 处，调用 #addActiveProfiles(Set<Profile> profiles) 方法，再添加激活的 Profile 们（在配置中）到 profiles 中。代码如下：

```
// ConfigFileApplicationListener#Loader.java

void addActiveProfiles(Set<Profile> profiles) {
    if (profiles.isEmpty()) {
        return;
    }
    // 如果已经标记 activatedProfiles 为 true ，则直接返回
    if (this.activatedProfiles) {
        if (this.logger.isDebugEnabled()) {
            this.logger.debug("Profiles already activated, '" + profiles + "' will not be applied");
        }
        return;
    }
    // 添加到 profiles 中
    this.profiles.addAll(profiles);
    if (this.logger.isDebugEnabled()) {
        this.logger.debug("Activated activeProfiles " + StringUtils.collectionToCommaDelimitedString(profiles))
```

```
        }
        // 标记 activatedProfiles 为 true 。
        this.activatedProfiles = true;
        // 移除 profiles 中，默认的配置们。
        removeUnprocessedDefaultProfiles();
    }

    private void removeUnprocessedDefaultProfiles() {
        this.profiles.removeIf(
                (profile) -> (profile != null && profile.isDefaultProfile()));
    }
```

<3> 处，如果没有激活的 Profile 们，则添加默认的 Profile 。此处的“默认”是，指的是配置文件中的 "spring.profiles.default" 对应的值。

这块比较绕，胖友最好自己调试下。例如说，我们在 JVM 启动增加 --spring.profiles.active=prod ，则结果如下图：




#### 4.4.1.1.2 load

#load(Profile profile, DocumentFilterFactory filterFactory, DocumentConsumer consumer) 方法，加载指定 Profile 的配置文件。代码如下：

```
// ConfigFileApplicationListener#Loader.java

private void load(Profile profile, DocumentFilterFactory filterFactory, DocumentConsumer consumer) {
    // <1> 获得要检索配置的路径
    getSearchLocations().forEach((location) -> {
        // 判断是否为文件夹
        boolean isFolder = location.endsWith("/");
        // <2> 获得要检索配置的文件名集合
        Set<String> names = isFolder ? getSearchNames() : NO_SEARCH_NAMES;
        // <3> 遍历文件名集合，逐个加载配置文件
        names.forEach(
                (name) -> load(location, name, profile, filterFactory, consumer));
    });
}
```

<1> 处，调用 #getSearchLocations() 方法，获得要检索配置的路径们。代码如下：

```
    // ConfigFileApplicationListener.java
```

```
// Note the order is from least to most specific (last one wins)
private static final String DEFAULT_SEARCH_LOCATIONS = "classpath:/,classpath:/config/,file:./,file:./config/"

/**
 * The "config location" property name.
 */
public static final String CONFIG_LOCATION_PROPERTY = "spring.config.location";
/**
 * The "config additional location" property name.
 */
public static final String CONFIG_ADDITIONAL_LOCATION_PROPERTY = "spring.config.additional-location";

// ConfigFileApplicationListener#Loader.java

private Set<String> getSearchLocations() {
    // 获得 `"spring.config.location"` 对应的配置的值
    if (this.environment.containsProperty(CONFIG_LOCATION_PROPERTY)) {
        return getSearchLocations(CONFIG_LOCATION_PROPERTY);
    }
    // 获得 `"spring.config.additional-location"` 对应的配置的值
    Set<String> locations = getSearchLocations(CONFIG_ADDITIONAL_LOCATION_PROPERTY);
    // 添加 searchLocations 到 locations 中
    locations.addAll(asResolvedSet(ConfigFileApplicationListener.this.searchLocations, DEFAULT_SEARCH_LOCATIONS
    return locations;
}

private Set<String> getSearchLocations(String propertyName) {
    Set<String> locations = new LinkedHashSet<>();
    // 如果 environment 中存在 propertyName 对应的值
    if (this.environment.containsProperty(propertyName)) {
        // 读取属性值，进行分割后，然后遍历
        for (String path : asResolvedSet(this.environment.getProperty(propertyName), null)) {
            // 处理 path
            if (!path.contains("$")) {
                path = StringUtils.cleanPath(path);
                if (!ResourceUtils.isUrl(path)) {
                    path = ResourceUtils.FILE_URL_PREFIX + path;
                }
            }
            // 添加到 path 中
            locations.add(path);
        }
    }
    return locations;
}

// 优先使用 value 。如果 value 为空，则使用 fallback
private Set<String> asResolvedSet(String value, String fallback) {
    List<String> list = Arrays.asList(StringUtils.trimArrayElements(
            StringUtils.commaDelimitedListToStringArray((value != null)
                    ? this.environment.resolvePlaceholders(value) : fallback)));
    Collections.reverse(list);
    return new LinkedHashSet<>(list);
}
```

- 看似比较长，逻辑并不复杂。
- 如果配置了 `"spring.config.location"` ，则使用它的值，作为要检索配置的路径。
- 如果配置了 `"spring.config.additional-location"` ，则使用它作为附加要检索配置的路径。当然，还是会添加默认的 `DEFAULT_SEARCH_LOCATIONS` 路径。

- 默认情况下，返回的值是 DEFAULT_SEARCH_LOCATIONS 。因为，绝大数情况，我们并不会做相应的配置。 所以，胖友如果懒的看逻辑，就记得这个结论就可以了。

<2> 处，调用 #getSearchNames() 方法，获得要检索配置的文件名集合。代码如下：

```
// ConfigFileApplicationListener.java

private static final String DEFAULT_NAMES = "application";

private static final Set<String> NO_SEARCH_NAMES = Collections.singleton(null);

public static final String CONFIG_NAME_PROPERTY = "spring.config.name";

// ConfigFileApplicationListener#Loader.java

private Set<String> getSearchNames() {
    // 获得 `"spring.config.name"` 对应的配置的值
    if (this.environment.containsProperty(CONFIG_NAME_PROPERTY)) {
        String property = this.environment.getProperty(CONFIG_NAME_PROPERTY);
        return asResolvedSet(property, null);
    }
    // 添加 names or DEFAULT_NAMES 到 locations 中
    return asResolvedSet(ConfigFileApplicationListener.this.names, DEFAULT_NAMES);
}
```

- 通过 DEFAULT_NAMES 静态属性，我们可以知道，默认都去的配置文件名（不考虑后缀）为 "application" 。
- 剩余的逻辑，胖友简单看看即可。
- 默认情况下，返回的值是 DEFAULT_NAMES 。因为，绝大数情况，我们并不会做相应的配置。 所以，胖友如果懒的看逻辑，就记得这个结论就可以了。

<3> 处，遍历 names 数组，逐个调用 #load(String location, String name, Profile profile, DocumentFilterFactory filterFactory, DocumentConsumer consumer) 方法，逐个加载 Profile 指定的配置文件。详细解析，见 「4.4.1.1.3 load」 。

#### 4.4.1.1.3 load

#load(String location, String name, Profile profile, DocumentFilterFactory filterFactory, DocumentConsumer consumer) 方法，逐个加载 Profile 指定的配置文件。代码如下：

```
// ConfigFileApplicationListener#Loader.java

private void load(String location, String name, Profile profile, DocumentFilterFactory filterFactory, DocumentConsume
    // <1> 这块逻辑先无视，因为我们不会配置 name 为空。
    // 默认情况下，name 为 DEFAULT_NAMES=application
    if (!StringUtils.hasText(name)) {
        for (PropertySourceLoader loader : this.propertySourceLoaders) {
            if (canLoadFileExtension(loader, location)) {
                load(loader, location, profile, filterFactory.getDocumentFilter(profile), consumer);
                return;
            }
        }
    }
    Set<String> processed = new HashSet<>(); // 已处理的文件后缀集合
    // <2> 遍历 propertySourceLoaders 数组，逐个使用 PropertySourceLoader 读取配置
```

```
        for (PropertySourceLoader loader : this.propertySourceLoaders) {
            // <3> 遍历每个 PropertySourceLoader 可处理的文件后缀集合
            for (String fileExtension : loader.getFileExtensions()) {
                // <4> 添加到 processed 中，一个文件后缀，有且仅能被一个 PropertySourceLoader 所处理
                if (processed.add(fileExtension)) {
                    // 加载 Profile 指定的配置文件（带后缀）
                    loadForFileExtension(loader, location + name, "." + fileExtension,
                            profile, filterFactory, consumer);
                }
            }
        }
    }
```

<1> 处的逻辑，可以无视。具体的原因，见我添加的注释。
<2> 处，遍历 propertySourceLoaders 数组，逐个使用 PropertySourceLoader 读取配置。
<3> 处，遍历每个 PropertySourceLoader 可处理的文件后缀集合。例如说，PropertiesPropertySourceLoader 可处理 .properties 和 .xml 后缀，YamlPropertySourceLoader 可处理 .yml 和 .yaml 后置。
<4> 处，添加到 processed 中。一个文件后缀，有且仅能被一个 PropertySourceLoader 所处理。
<5> 处，调用 #loadForFileExtension(PropertySourceLoader loader, String prefix, String fileExtension, Profile profile, DocumentFilterFactory filterFactory, DocumentConsumer consumer) 方法，加载 Profile 指定的配置文件（带后缀）。详细解析，见 「4.4.1.1.7 loadForFileExtension」 。

#### 4.4.1.1.4 Document

因为稍后在讲解 #loadForFileExtension(...) 方法需要用到，所以插播下。

Document ，是 ConfigFileApplicationListener 的内部类，用于封装 PropertySourceLoader 加载配置文件后。代码如下：

```
    // ConfigFileApplicationListener#Document.java

    private static class Document {

        private final PropertySource<?> propertySource;

        /**
         * 对应 `spring.profiles` 属性值
         */
        private String[] profiles;

        /**
         * 对应 `spring.profiles.active` 属性值
         */
        private final Set<Profile> activeProfiles;

        /**
         * 对应 `spring.profiles.include` 属性值
         */
        private final Set<Profile> includeProfiles;
    }
```

##### 4.4.1.1.4.1 DocumentsCacheKey

DocumentsCacheKey ，是 ConfigFileApplicationListener 的内部类，用于表示加载 Documents 的缓存 KEY 。代码如下：

```
// ConfigFileApplicationListener#DocumentsCacheKey.java

/**
 * Cache key used to save loading the same document multiple times.
 */
private static class DocumentsCacheKey {

 private final PropertySourceLoader loader;

 private final Resource resource;

    DocumentsCacheKey(PropertySourceLoader loader, Resource resource) {
     this.loader = loader;
     this.resource = resource;
    }

 @Override
 public boolean equals(Object obj) {
     if (this == obj) {
         return true;
        }
     if (obj == null || getClass() != obj.getClass()) {
         return false;
        }
        DocumentsCacheKey other = (DocumentsCacheKey) obj;
     return this.loader.equals(other.loader)
                && this.resource.equals(other.resource);
    }

 @Override
 public int hashCode() {
     return this.loader.hashCode() * 31 + this.resource.hashCode();
    }

}
```

因为一个配置文件在 「4.4.1.1.7 loadForFileExtension」 方法中，我们会看到可能存在重复加载的情况，所以通过缓存，避免重新读取~

#### 4.4.1.1.5 DocumentFilterFactory

因为稍后在讲解 `#loadForFileExtension(...)` 方法需要用到，所以插播下。

DocumentFilterFactory ，是 ConfigFileApplicationListener 的内部接口，用于创建 DocumentFilter 对象。代码如下：

```
// ConfigFileApplicationListener#DocumentFilterFactory.java

@FunctionalInterface
private interface DocumentFilterFactory {

 DocumentFilter getDocumentFilter(Profile profile);
```

```
}
```

在 「4.4.1.1 load」 中，我们在该方法中，已经看到它的两个匿名实现类，如下：

```
// ConfigFileApplicationListener#Loader.java

第一个：this::getPositiveProfileFilter
第二个：this::getNegativeProfileFilter
```

它们分别调用对应的方法，创建 DocumentFilter 对象。

#### 4.4.1.1.5.1 DocumentFilter

DocumentFilter ，是 ConfigFileApplicationListener 的内部接口，用于匹配配置加载后的 Document 对象。代码如下：

```
// ConfigFileApplicationListener#DocumentFilter.java

@FunctionalInterface
private interface DocumentFilter {

 boolean match(Document document);

}
```

在下面的文章中，我们会看到，加载的配置文件后，返回的是 Document 对象。但是，返回的 Document 对象，需要和 Profile 进行匹配。

#### 4.4.1.1.5.2 getPositiveProfileFilter

#getPositiveProfileFilter(Profile profile) 方法，代码如下：

```
// ConfigFileApplicationListener#Loader.java

private DocumentFilter getPositiveProfileFilter(Profile profile) {
    return (Document document) -> {
        // <1> 当 profile 为空时，document.profiles 也要为空
        if (profile == null) {
            return ObjectUtils.isEmpty(document.getProfiles());
        }
        // <2> 要求 document.profiles 包含 profile
        // 并且，environment.activeProfiles 包含 document.profiles
        // 总结来说，environment.activeProfiles 包含 document.profiles 包含 profile
        return ObjectUtils.containsElement(document.getProfiles(), profile.getName())
                && this.environment.acceptsProfiles(Profiles.of(document.getProfiles()));
    };
}
```

<1> 处，当传入的 profile 为空时，要求 document.profiles 也要为空。
<2> 处，当传入的 profile 非空时，要求 environment.activeProfiles 包含 document.profiles 包含 profile 。

可能比较绕，胖友先看 《Spring-boot动态profiles的实践》 文章的 「Spring Boot的Profiles属性」 部分。

什么意思呢？假设一个 application-prod.properties 的配置文件，一般我们的理解是对应 Profile 为 prod 的情况，对吧？！但是，如果说我们在配置文件中增加了 spring.profiles=dev ，那它实际是属于 Profile 为 dev 的情况。

芮芮：第一次知道还有这样的设定！！！

当然，我们绝大都数情况，并不会去定义 spring.profiles 属性。所以呢，分成两种情况：

芮芮：仔细理解，我也懵逼了好多小时！！！！

- profile 为 null 的情况，处理默认情况，即我们未定义 spring.profiles 属性。
- profile 非 null 的情况，处理配置文件中定义了 spring.profiles 属性，则需要使用 profile 和 spring.profiles 匹配，并且它要属于 environment.activeProfiles 中已经激活的。

所以呢，我们在 「4.4.1.1 load」 中，看到它是在加载指定 Profile 的配置文件所使用。

#### 4.4.1.1.5.3 getPositiveProfileFilter

#getPositiveProfileFilter(Profile profile) 方法，代码如下：

```
// ConfigFileApplicationListener#Loader.java

private DocumentFilter getNegativeProfileFilter(Profile profile) {
    return (Document document) -> (profile == null // <1>
            && !ObjectUtils.isEmpty(document.getProfiles()) // <2> 非空
            // environment.activeProfiles 包含 document.profiles
            && this.environment.acceptsProfiles(Profiles.of(document.getProfiles()))); // <3>
}
```

<1> 处，要求传入的 profile 为空。因为呢，「4.4.1.1 load」 中，看到它是在加载无 Profile 的配置文件所使用。
<2> 处，要求 document.profiles 非空。一般情况下，我们在 application.properties 中，也并不会填写 spring.profiles 属性值。这就是说，这个方法默认基本返回 false 。
<3> 处，environment.activeProfiles 包含 document.profiles 。

#### 4.4.1.1.6 DocumentConsumer

因为稍后在讲解 #loadForFileExtension(...) 方法需要用到，所以插播下。

DocumentConsumer ，是 ConfigFileApplicationListener 的内部接口，用于处理传入的 Document 。代码如下：

```
// ConfigFileApplicationListener#DocumentConsumer.java

/**
 * Consumer used to handle a loaded {@link Document}.
 */
@FunctionalInterface
private interface DocumentConsumer {
```

```
        void accept(Profile profile, Document document);

    }
```

在 「4.4.1.1 load」 中，我们在该方法中，已经看到它的一个匿名实现类，如下：

```
// ConfigFileApplicationListener#Loader.java

第一个：addToLoaded(MutablePropertySources::addLast, false))
第二个：addToLoaded(MutablePropertySources::addFirst, true))
```

差别在于传入的 #addToLoaded(BiConsumer<MutablePropertySources, PropertySource<?>> addMethod, boolean checkForExisting) 方法的 addMethod 参数不同。为什么呢？

- 对于有 Profile 的情况，使用前者 MutablePropertySources::addLast ，将 Document 的 PropertySource 添加到尾部。
- 对于无 Profile 的情况，使用后者 MutablePropertySources::addFirst ，将 Document 的 PropertySource 添加到头部。
- 最终，我们看到 #addLoadedPropertySources() 方法中，会执行 Collections.reverse(loaded) 代码段，进行颠倒。为什么呢？这样，就能很巧妙的实现 application-prod.properties 的优先级，高于 application.properties ，从而实现相同属性时，覆盖读取~，即读取的是 application-prod.properties 的属性配置。

#### 4.4.1.1.6.1 addToLoaded

#addToLoaded(BiConsumer<MutablePropertySources, PropertySource<?>> addMethod, boolean checkForExisting) 方法，将 Document 的 PropertySource ，添加到 Loader.loaded 中。代码如下：

```
// ConfigFileApplicationListener#Loader.java

private DocumentConsumer addToLoaded(BiConsumer<MutablePropertySources, PropertySource<?>> addMethod, boolean checkFo
    // 创建 DocumentConsumer 对象
    return (profile, document) -> {
        // 如果要校验已经存在的情况，则如果已经存在，则直接 return
        if (checkForExisting) {
            for (MutablePropertySources merged : this.loaded.values()) {
                if (merged.contains(document.getPropertySource().getName())) {
                    return;
                }
            }
        }
        // 获得 profile 对应的 MutablePropertySources 对象
        MutablePropertySources merged = this.loaded.computeIfAbsent(profile,
                (k) -> new MutablePropertySources());
        // 将加载的 document 合并到 merged 中
        addMethod.accept(merged, document.getPropertySource());
    };
}
```

创建了一个 DocumentConsumer 对象。而其内部的逻辑，很简单，胖友自己瞅瞅即可。

#### 4.4.1.1.7 loadForFileExtension

#loadForFileExtension(PropertySourceLoader loader, String prefix, String fileExtension, Profile profile, DocumentFilterFactory filterFactory, DocumentConsumer consumer) 方法，加载 Profile 指定的配置文件（带后缀）。代码如下：

```
// ConfigFileApplicationListener#Loader.java

private void loadForFileExtension(PropertySourceLoader loader, String prefix,
        String fileExtension, Profile profile,
        DocumentFilterFactory filterFactory, DocumentConsumer consumer) {
    // <1> 获得 DocumentFilter 对象
    DocumentFilter defaultFilter = filterFactory.getDocumentFilter(null);
    DocumentFilter profileFilter = filterFactory.getDocumentFilter(profile);
    // <2> 加载 Profile 指定的配置文件（带后缀）。
    if (profile != null) {
        // Try profile-specific file & profile section in profile file (gh-340)
        // 加载 Profile 指定的配置文件（带后缀）。
        String profileSpecificFile = prefix + "-" + profile + fileExtension;
        load(loader, profileSpecificFile, profile, defaultFilter, consumer); // <2.1> 情况一，未配置 spring.profile 属
        load(loader, profileSpecificFile, profile, profileFilter, consumer); // <2.2> 情况二，有配置 spring.profile 属
        // Try profile specific sections in files we've already processed
        // <2.3》 特殊情况，之前读取 Profile 对应的配置文件，也可被当前 Profile 所读取。
        // 举个例子，假设之前读取了 Profile 为 common 对应配置文件是 application-common.properties ，里面配置了 sprin
        //         那么，此时如果读取的 Profile 为 dev 时，也能读取 application-common.properties 这个配置文件
        for (Profile processedProfile : this.processedProfiles) {
            if (processedProfile != null) {
                String previouslyLoaded = prefix + "-" + processedProfile + fileExtension; // 拼接之前的配置文件名
                load(loader, previouslyLoaded, profile, profileFilter, consumer); // 注意噢，传入的 profile 是当前的
            }
        }
    }
    // Also try the profile-specific section (if any) of the normal file
    // <3> 加载（无需带 Profile）指定的配置文件（带后缀）。
    load(loader, prefix + fileExtension, profile, profileFilter, consumer);
}
```

<1> 处，获得两个 DocumentFilter 对象。为什么是两个呢？胖友思考下。实际上，我们在「4.4.1.1.5.2 getPositiveProfileFilter」 中，已经说明了答案。
<2> 处，加载 Profile 指定的配置文件（带后缀）。有三种情况，胖友认真看 <2.1>、<2.2>、<2.3> 处的代码注释。
<3> 处，加载（无需带 Profile）指定的配置文件（带后缀）。
关于 #load(PropertySourceLoader loader, String location, Profile profile, DocumentFilter filter, DocumentConsumer consumer) 方法，真正加载 Profile 指定的配置文件（带后缀）。详细解析，见「4.4.1.1.8 load」 。

#### 4.4.1.1.8 load

#load(PropertySourceLoader loader, String location, Profile profile, DocumentFilter filter, DocumentConsumer consumer) 方法，真正加载 Profile 指定的配置文件（带后缀）。代码如下：

```
// ConfigFileApplicationListener#Loader.java

private void load(PropertySourceLoader loader, String location, Profile profile, DocumentFilter filter, DocumentConsu
    try {
        // <1.1> 判断指定的配置文件是否存在。若不存在，则直接返回
        Resource resource = this.resourceLoader.getResource(location);
```

```
            if (!resource.exists()) {
                if (this.logger.isTraceEnabled()) {
                    StringBuilder description = getDescription("Skipped missing config ", location, resource, profile);
                    this.logger.trace(description);
                }
                return;
            }
            // <1.2> 如果没有文件后缀的配置文件，则忽略，不进行读取
            if (!StringUtils.hasText(StringUtils.getFilenameExtension(resource.getFilename()))) {
                if (this.logger.isTraceEnabled()) {
                    StringBuilder description = getDescription("Skipped empty config extension ", location, resource, pro
                    this.logger.trace(description);
                }
                return;
            }
            // <1.3> 加载配置文件，返回 Document 数组
            String name = "applicationConfig: [" + location + "]";
            List<Document> documents = loadDocuments(loader, name, resource);
            // <1.4> 如果没加载到，则直接返回
            if (CollectionUtils.isEmpty(documents)) {
                if (this.logger.isTraceEnabled()) {
                    StringBuilder description = getDescription(
                            "Skipped unloaded config ", location, resource, profile);
                    this.logger.trace(description);
                }
                return;
            }
            // <2> 使用 DocumentFilter 过滤匹配的 Document ，添加到 loaded 数组中。
            List<Document> loaded = new ArrayList<>();
            for (Document document : documents) {
                if (filter.match(document)) { // 匹配
                    addActiveProfiles(document.getActiveProfiles()); // <2.1>
                    addIncludedProfiles(document.getIncludeProfiles()); // <2.2>
                    loaded.add(document);
                }
            }
            Collections.reverse(loaded);
            // <3> 使用 DocumentConsumer 进行消费 Document ，添加到本地的 loaded 中。
            if (!loaded.isEmpty()) {
                loaded.forEach((document) -> consumer.accept(profile, document));
                if (this.logger.isDebugEnabled()) {
                    StringBuilder description = getDescription("Loaded config file ", location, resource, profile);
                    this.logger.debug(description);
                }
            }
        } catch (Exception ex) {
            throw new IllegalStateException("Failed to load property "
                    + "source from location '" + location + "'", ex);
        }
    }
```

<1.1> 处，判断指定的配置文件是否存在。若不存在，则直接返回。
<1.2> 处，如果没有文件后缀的配置文件，则忽略，不进行读取。
<1.3> 处，调用 `#loadDocuments(PropertySourceLoader loader, String name, Resource resource)` 方法，加载配置文件，并返回 Document 数组。详细解析，见 「4.4.1.1.8.1 loadDocuments」 小节。
<1.4> 处，如果没加载到，则直接返回。

<2> 处，遍历加载到 documents 数组，逐个调用 `DocumentFilter#match(Document document)` 方法，进行匹配。若匹配成功，则添加到 loaded 中。

- <2.1> 处，#addActiveProfiles(Set<Profile> profiles) 方法，已经在 「4.4.1.1.1 initializeProfiles」 中，详细解析。
- <2.2> 处，#addIncludedProfiles(Set<Profile> profiles) 方法，代码如下：

```
// ConfigFileApplicationListener#Loader.java

private void addIncludedProfiles(Set<Profile> includeProfiles) {
    // 整体逻辑是 includeProfiles - processedProfiles + profiles
    LinkedList<Profile> existingProfiles = new LinkedList<>(this.profiles);
 this.profiles.clear();
 this.profiles.addAll(includeProfiles);
 this.profiles.removeAll(this.processedProfiles);
 this.profiles.addAll(existingProfiles);
}
```

~

- <3> 处，遍历 loaded 数组，调用 DocumentConsumer#accept(Profile profile, Document document) 方法，添加到本地的 Loader.loaded 中。此处，在结合 「4.4.1.1.6 DocumentConsumer」 一起看。

至此，Spring Boot 加载配置的功能，基本是完成了。总的来说，理解大体的流程，还是相对比较容易的。但是，想要扣懂这个过程的每一个细节，需要多多的调试。

> 艿艿：可能有些细节写的不到位，或者解释的不到位，又或者讲的不正确。所以，有任何疑惑，请立刻马上赶紧给我星球留言提问哈。

#### 4.4.1.1.8.1 loadDocuments

#loadDocuments(PropertySourceLoader loader, String name, Resource resource) 方法，加载配置文件，并返回 Document 数组。代码如下：

```
// ConfigFileApplicationListener#Loader.java

private List<Document> loadDocuments(PropertySourceLoader loader, String name, Resource resource) throws IOException
    // <1> 创建 DocumentsCacheKey 对象，从 loadDocumentsCache 缓存中加载 Document 数组
    DocumentsCacheKey cacheKey = new DocumentsCacheKey(loader, resource);
    List<Document> documents = this.loadDocumentsCache.get(cacheKey);
    // <2.1> 如果不存在，则使用 PropertySourceLoader 加载指定配置文件
    if (documents == null) {
        List<PropertySource<?>> loaded = loader.load(name, resource);
        // <2.2> 将返回的 PropertySource 数组，封装成 Document 数组
        documents = asDocuments(loaded);
        // <2.3> 添加到 loadDocumentsCache 缓存中
        this.loadDocumentsCache.put(cacheKey, documents);
    }
    return documents;
}
```

<1> 处，创建 DocumentsCacheKey 对象，从 loadDocumentsCache 缓存中加载 Document 数组。

<2.1> 处，如果不存在，则调用 PropertySourceLoader#load(String name, Resource resource) 方法，加载指定配置文件。详细的解析，见 「7. PropertySourceLoader」 中。

<2.2> 处，调用 #asDocuments(List<PropertySource<?>> loaded) 方法，将返回的 PropertySource 数组

，封装成 Document 数组。代码如下：

```
// ConfigFileApplicationListener#Loader.java

private List<Document> asDocuments(List<PropertySource<?>> loaded) {
    if (loaded == null) {
        return Collections.emptyList();
    }
    return loaded.stream().map((propertySource) -> {
        // 创建 Binder 对象
        Binder binder = new Binder(
                ConfigurationPropertySources.from(propertySource),
                this.placeholdersResolver);
        return new Document(propertySource,
                binder.bind("spring.profiles", STRING_ARRAY).orElse(null), // 读取 "spring.profiles" 配置
                getProfiles(binder, ACTIVE_PROFILES_PROPERTY), // 读取 "spring.profiles.active" 配置
                getProfiles(binder, INCLUDE_PROFILES_PROPERTY)); // 读取 "spring.profiles.include" 配置
    }).collect(Collectors.toList());
}

private Set<Profile> getProfiles(Binder binder, String name) {
 return binder.bind(name, STRING_ARRAY).map(this::asProfileSet).orElse(Collections.emptySet());
}

private Set<Profile> asProfileSet(String[] profileNames) {
    List<Profile> profiles = new ArrayList<>();
 for (String profileName : profileNames) {
        profiles.add(new Profile(profileName));
    }
 return new LinkedHashSet<>(profiles);
}
```

<2.3> 处，添加到 loadDocumentsCache 缓存中。

# 5. RandomValuePropertySource

org.springframework.boot.env.RandomValuePropertySource ，继承 PropertySource 类，提供随机值的 PropertySource 实现类。

不了解 Spring Boot 从配置文件中获取随机数，可以看看 《Spring Boot学习 - 从配置文件中获取随机数》 文章。

## 5.1 addToEnvironment

#addToEnvironment(ConfigurableEnvironment environment) 静态方法，创建 RandomValuePropertySource 对象，添加到 environment 中。代码如下：

```
// RandomValuePropertySource.java

/**
 * Name of the random {@link PropertySource}.
 */
public static final String RANDOM_PROPERTY_SOURCE_NAME = "random";
```

```
public static void addToEnvironment(ConfigurableEnvironment environment) {
    environment.getPropertySources().addAfter(StandardEnvironment.SYSTEM_ENVIRONMENT_PROPERTY_SOURCE_NAME, new Random
    logger.trace("RandomValuePropertySource add to Environment");
}
```

## 5.2 getProperty

实现 `#getProperty(String name)` 方法，获得 `name` 对应的随机值。代码如下：

```
// RandomValuePropertySource.java

private static final String PREFIX = "random.";

@Override
public Object getProperty(String name) {
    // <1> 必须以 random. 前缀
    if (!name.startsWith(PREFIX)) {
        return null;
    }
    if (logger.isTraceEnabled()) {
        logger.trace("Generating random property for '" + name + "'");
    }
    // <2> 根据类型，获得随机值
    return getRandomValue(name.substring(PREFIX.length()));
}
```

<1> 处，必须以 `"random."` 前缀。在获取属性值，`name` 前后的 `${}` 已经被去掉。

<2> 处，调用 `#getRandomValue(String type)` 方法，根据类型，获得随机值。代码如下：

```
// RandomValuePropertySource.java

private Object getRandomValue(String type) {
    // int
    if (type.equals("int")) {
        return getSource().nextInt();
    }
    // long
    if (type.equals("long")) {
        return getSource().nextLong();
    }
    // int 范围
    String range = getRange(type, "int");
    if (range != null) {
        return getNextIntInRange(range);
    }
    // long 范围
    range = getRange(type, "long");
    if (range != null) {
        return getNextLongInRange(range);
    }
    // uuid
    if (type.equals("uuid")) {
        return UUID.randomUUID().toString();
    }
```

```
        // md5
        return getRandomBytes();
    }

    private String getRange(String type, String prefix) {
        if (type.startsWith(prefix)) {
            int startIndex = prefix.length() + 1;
            if (type.length() > startIndex) {
                return type.substring(startIndex, type.length() - 1);
            }
        }
        return null;
    }

    private int getNextIntInRange(String range) {
        String[] tokens = StringUtils.commaDelimitedListToStringArray(range);
        int start = Integer.parseInt(tokens[0]);
        if (tokens.length == 1) {
            return getSource().nextInt(start);
        }
        return start + getSource().nextInt(Integer.parseInt(tokens[1]) - start);
    }

    private long getNextLongInRange(String range) {
        String[] tokens = StringUtils.commaDelimitedListToStringArray(range);
        if (tokens.length == 1) {
            return Math.abs(getSource().nextLong() % Long.parseLong(tokens[0]));
        }
        long lowerBound = Long.parseLong(tokens[0]);
        long upperBound = Long.parseLong(tokens[1]) - lowerBound;
        return lowerBound + Math.abs(getSource().nextLong() % upperBound);
    }

    private Object getRandomBytes() {
        byte[] bytes = new byte[32];
        getSource().nextBytes(bytes);
        return DigestUtils.md5DigestAsHex(bytes);
    }
```

- 比较简单，瞅瞅即明白。

这个谜题，是不是被揭晓了~

# 6. PropertySourcesPlaceholdersResolver

org.springframework.boot.context.properties.bind.PropertySourcesPlaceholdersResolver ，实现 PropertySource 对应的值是占位符的解析器 。

## 6.1 构造方法

```
// PropertySourcesPlaceholdersResolver.java

private final Iterable<PropertySource<?>> sources;
```

```
private final PropertyPlaceholderHelper helper;

public PropertySourcesPlaceholdersResolver(Environment environment) {
 this(getSources(environment), null);
}

public PropertySourcesPlaceholdersResolver(Iterable<PropertySource<?>> sources) {
 this(sources, null);
}

public PropertySourcesPlaceholdersResolver(Iterable<PropertySource<?>> sources,
        PropertyPlaceholderHelper helper) {
 this.sources = sources;
 this.helper = (helper != null) ? helper
            : new PropertyPlaceholderHelper(SystemPropertyUtils.PLACEHOLDER_PREFIX, // ${
                    SystemPropertyUtils.PLACEHOLDER_SUFFIX, // }
                    SystemPropertyUtils.VALUE_SEPARATOR, true);
}

// 获得 PropertySources 们
private static PropertySources getSources(Environment environment) {
    Assert.notNull(environment, "Environment must not be null");
    Assert.isInstanceOf(ConfigurableEnvironment.class, environment, "Environment must be a ConfigurableEnvironment");
 return ((ConfigurableEnvironment) environment).getPropertySources();
}
```

其中，创建的 `helper` 属性，为 PropertyPlaceholderHelper 对象。其中 `SystemPropertyUtils.PLACEHOLDER_PREFIX` 为 `${` ， `SystemPropertyUtils.PLACEHOLDER_SUFFIX` 为 `}` 。这样，例如说 RandomValuePropertySource 的 `${random.int}` 等等，就可以被 PropertySourcesPlaceholdersResolver 所处理。

## 6.2 resolvePlaceholders

实现 `#resolvePlaceholders(Object value)` 方法，解析占位符。代码如下：

```
// PropertySourcesPlaceholdersResolver.java

@Override
public Object resolvePlaceholders(Object value) {
    // 如果 value 是 String 类型，才是占位符
    if (value instanceof String) {
        return this.helper.replacePlaceholders((String) value, this::resolvePlaceholder);
    }
    return value;
}
```

如果 `value` 是 String 类型，才可能是占位符。满足时，调用 `PropertyPlaceholderHelper#replacePlaceholders(String value, PlaceholderResolver placeholderResolver)` 方法，解析出占位符里面的内容。

- 例如说：PropertySourcesPlaceholdersResolver 中，占位符是 `${}` ，那么 `${random.int}` 被解析后的内容是 `random.int` 。
- 解析到占位符后，则回调 `#resolvePlaceholder(String placeholder)` 方法，获得占位符对应的值。代码如下：

```
// PropertySourcesPlaceholdersResolver.java

protected String resolvePlaceholder(String placeholder) {
    if (this.sources != null) {
        // 遍历 sources 数组，逐个获得属性值。若获取到，则进行返回
        for (PropertySource<?> source : this.sources) {
            Object value = source.getProperty(placeholder);
            if (value != null) {
                return String.valueOf(value);
            }
        }
    }
    return null;
}
```

- 这样， RandomValuePropertySource 是不是就被串起来喽。

# 7. PropertySourceLoader

org.springframework.boot.env.PropertySourceLoader 接口，加载指定配置文件，返回 PropertySource 数组。代码如下：

```
// PropertySourceLoader.java

public interface PropertySourceLoader {

 /**
     * 获得可以处理的配置文件的后缀
     *
     * Returns the file extensions that the loader supports (excluding the '.').
     * @return the file extensions
     */
    String[] getFileExtensions();

 /**
     * 加载指定配置文件，返回 PropertySource 数组
     *
     * Load the resource into one or more property sources. Implementations may either
     * return a list containing a single source, or in the case of a multi-document format
     * such as yaml a source for each document in the resource.
     * @param name the root name of the property source. If multiple documents are loaded
     * an additional suffix should be added to the name for each source loaded.
     *             PropertySource 的名字
     * @param resource the resource to load
     *             配置文件的 Resource 对象
     * @return a list property sources
     * @throws IOException if the source cannot be loaded
     */
    List<PropertySource<?>> load(String name, Resource resource) throws IOException;

}
```

可能胖友，和我开始一样，为什么一个配置文件，加载后会存在多个 PropertySource 对象呢？下面，我们来见分晓~

# 7.1 PropertiesPropertySourceLoader

org.springframework.boot.env.PropertiesPropertySourceLoader ，实现 PropertySourceLoader 接口，加载 .xml 和 .properties 类型的配置文件。代码如下：

```
// PropertiesPropertySourceLoader.java

public class PropertiesPropertySourceLoader implements PropertySourceLoader {

    private static final String XML_FILE_EXTENSION = ".xml";

    @Override
    public String[] getFileExtensions() {
        return new String[] { "properties", "xml" }; // <1>
    }

    @Override
    public List<PropertySource<?>> load(String name, Resource resource)
            throws IOException {
        // <2.1> 读取指定配置文件，返回 Map 对象
        Map<String, ?> properties = loadProperties(resource);
        // <2.2> 如果 Map 为空，返回空数组
        if (properties.isEmpty()) {
            return Collections.emptyList();
        }
        // <2.3> 将 Map 封装成 OriginTrackedMapPropertySource 对象，然后返回单元素的数组
        return Collections.singletonList(new OriginTrackedMapPropertySource(name, properties));
    }

}
```

<1> 处，返回可处理的文件类型，为 properties 和 xml 。

<2.1> 处，调用 #loadProperties(Resource resource) 方法，读取指定配置文件，返回 Map 对象。代码如下：

```
// PropertiesPropertySourceLoader.java

private Map<String, ?> loadProperties(Resource resource) throws IOException {
    String filename = resource.getFilename();
    // 读取 XML 后缀的配置文件
    if (filename != null && filename.endsWith(XML_FILE_EXTENSION)) {
        return (Map) PropertiesLoaderUtils.loadProperties(resource);
    }
    // 读取 Properties 后缀的配置文件
    return new OriginTrackedPropertiesLoader(resource).load();
}
```

- 根据 .xml 和 .properties 后缀，使用不同的读取方法。至于读取配置的逻辑，暂时不在本文的范畴，hoho 。感兴趣的胖友，自己去瞅瞅。

<2.2> 处，如果 Map 为空，返回空数组。

<2.3> 处，将 Map 封装成 OriginTrackedMapPropertySource 对象，然后返回单元素的数组。org.springframework.boot.env.OriginTrackedMapPropertySource ，继承 MapPropertySource 类，实现

OriginLookup 接口，代码如下：

```
// OriginTrackedMapPropertySource.java

public final class OriginTrackedMapPropertySource extends MapPropertySource
     implements OriginLookup<String> {

 @SuppressWarnings({ "unchecked", "rawtypes" })
 public OriginTrackedMapPropertySource(String name, Map source) {
     super(name, source);
    }

 @Override
 public Object getProperty(String name) {
        // 获得属性值
        Object value = super.getProperty(name);
     // 如果是 OriginTrackedValue 封装类型，则返回其真实的值
     if (value instanceof OriginTrackedValue) {
         return ((OriginTrackedValue) value).getValue();
        }
     return value;
    }

 @Override
 public Origin getOrigin(String name) {
        // 获得属性值
        Object value = super.getProperty(name);
        // 如果是 OriginTrackedValue 封装类型，则返回其 Origin
     if (value instanceof OriginTrackedValue) {
         return ((OriginTrackedValue) value).getOrigin();
        }
     return null;
    }

}
```

# 7.2 YamlPropertySourceLoader

org.springframework.boot.env.YamlPropertySourceLoader ，实现 PropertySourceLoader 接口，加载 .yaml 和 .yml 类型的配置文件。代码如下：

```
// YamlPropertySourceLoader.java

public class YamlPropertySourceLoader implements PropertySourceLoader {

    @Override
    public String[] getFileExtensions() {
        return new String[] { "yml", "yaml" }; // <1>
    }

    @Override
    public List<PropertySource<?>> load(String name, Resource resource) throws IOException {
        // <2> 如果不存在 org.yaml.snakeyaml.Yaml 类，说明没有引入 snakeyaml 依赖
        if (!ClassUtils.isPresent("org.yaml.snakeyaml.Yaml", null)) {
            throw new IllegalStateException("Attempted to load " + name
                    + " but snakeyaml was not found on the classpath");
```

```
        }
        // <3.1> 加载配置，返回 Map 数组
        List<Map<String, Object>> loaded = new OriginTrackedYamlLoader(resource).load();
        // <3.2> 如果数组为空，返回空数组
        if (loaded.isEmpty()) {
            return Collections.emptyList();
        }
        // <3.3> 将 Map 数组，封装成 OriginTrackedMapPropertySource 数组，返回
        List<PropertySource<?>> propertySources = new ArrayList<>(loaded.size());
        for (int i = 0; i < loaded.size(); i++) {
            String documentNumber = (loaded.size() != 1) ? " (document #" + i + ")" : "";
            propertySources.add(new OriginTrackedMapPropertySource(name + documentNumber, loaded.get(i)));
        }
        return propertySources;
    }

}
```

<1> 处，返回可处理的文件类型，为 `yaml` 和 `yml` 。
<2> 处，如果不存在 `org.yaml.snakeyaml.Yaml` 类，说明没有引入 snakeyaml 依赖，则抛出 IllegalStateException 异常。
<3.1> 处，调用 `OriginTrackedYamlLoader#load()` 方法，加载配置，返回 Map 数组。 哎哟，此处就是我们的好奇了，返回的是 Map 数组列！我们先来看看 《Spring Boot 使用 YML 文件配置多环境》 的 「1 一个yml文件」 ，每个 `----` 分割线，会解析成一个对应的 `Map<String, Object>` 对象。
<3.2> 处，如果 Map 数组为空，返回空数组。
<3.3> 处，将 Map 数组，封装成 OriginTrackedMapPropertySource 数组，然后返回。

# 666. 彩蛋

卧槽，真心是内心无比卧槽。比我预先的，长太太太多了。

如果有解释不到位的地方，麻烦胖友在星球提出下哟。hoho ， 春节期间初二，在老家写完~

参考和推荐如下文章：

oldflame-Jm 《Spring boot源码分析-profiles环境（4）》
oldflame-Jm 《Spring boot源码分析-ApplicationListener应用环境（5）》
youzhibing2904 《spring-boot-2.0.3不一样系列之源码篇 - run方法（二）之prepareEnvironment，绝对有值得你看的地方》

---

如下内容，是芳芳的草稿，胖友可以先不去了解。

假设 Profile 为 `prod,dev` 。读取结果如下：

如下过程，省略读取不到配置文件的情况。

getPositiveProfileFilter 的情况
- profile=null 部分
  - 读取 `classpath:/application.properties` ， 匹配成功。
  - 读取 `classpath:/application.yaml` ， 匹配成功。
- profile=prod
  - 读取 `classpath:/application-prod.properties` ， 匹配成功（基于 `defaultFilter`）。

        - 读取 classpath:/application-prod.properties ，匹配失败（基于 profileFilter）。
        - 读取 classpath:/application-prod.properties ，匹配失败（基于 profileFilter）。
        - 读取 classpath:/application.yaml ，匹配失败（基于 profileFilter）。【匹配上 prod 那部分】
    - profile=dev
        - 读取 classpath:/application-dev.properties ，匹配成功（基于 defaultFilter）。
        - 读取 classpath:/application-dev.properties ，匹配失败（基于 profileFilter）。
        - 读取 classpath:/application-dev.properties ，匹配失败（基于 profileFilter）。
        - 读取 classpath:/application.yaml ，匹配失败（基于 profileFilter）。【匹配上 dev 那部分】

getNegativeProfileFilter 的情况

- profile=null 部分
    - 读取 classpath:/application.properties ，匹配失败。
    - 读取 classpath:/application.yaml ，匹配成功【匹配上 prod、dev 那部分】。比较有意思，用于解决 profile=null 的情况，可以把激活的 Profile ，也匹配上。但是在目前这个情况下，即使匹配上，在 #addToLoaded(...) 方法中的 DocumentConsumer 的逻辑时，会在 checkForExisting 那块的逻辑，如果要校验已经存在的情况，则如果已经存在，则直接 return 被排除掉。因为在 profile=prod 和 profile=dev 部分，已经匹配上该部分的配置。
        - = =~当然，实在搞不懂这个逻辑，也不用纠结这个。重点是搞懂 Spring Boot 加载配置的核心逻辑。也就是，ConfigFileApplicationListener 整体的逻辑。
        - 芳芳又思考了下，貌似突然也有点想不通，什么情况下，这块逻辑会成功加载到 Profile 不匹配的配置文件，即在在 #addToLoaded(...) 方法中的 DocumentConsumer 的逻辑时，会在 checkForExisting 那块的逻辑，如果要校验已经存在的情况，则如果不存在。

文章目录

2014 - 2023 芋道源码 |
总访客数 次 && 总访问量 次


回到首页