
回到首页

芋道源码 —— 知识星球

我是一段不羁的公告！
记得给芳芳这 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
2021-02-13
Spring Boot

精尽 Spring Boot 源码分析 —— AutoConfigurationMetadataLoader

1. 概述

本文，我们来补充 《精尽 Spring Boot 源码分析 —— 自动配置》 文章，并未详细解析的 AutoConfigurationMetadataLoader 。在 SpringApplication 中，我们可以看到 `AutoConfigurationImportSelector.AutoConfigurationGroup#loadMetadata(ClassLoader classLoader, String path)` 方法中，加载自动配置类（AutoConfiguration）的元数据，是如下一段代码：

```
// AutoConfigurationImportSelector.AutoConfigurationGroup.java

/**
 * @return 获得 AutoConfigurationMetadata 对象
 */
private AutoConfigurationMetadata getAutoConfigurationMetadata() {
    // 不存在，则进行加载
    if (this.autoConfigurationMetadata == null) {
        this.autoConfigurationMetadata = AutoConfigurationMetadataLoader.loadMetadata(this.beanClassLoader);
    }
    // 存在，则直接返回
    return this.autoConfigurationMetadata;
}
```

在内部，会调用 `AutoConfigurationMetadataLoader#loadMetadata(ClassLoader classLoader)` 方法，加载 AutoConfigurationMetadata 对象。
而我们知道，后续会基于返回的 AutoConfigurationMetadata 对象，进行 AutoConfiguration 类的过滤，从而避免不符合条件的 AutoConfiguration 类的字节码，加载到 JVM 中。那么是怎么做到的呢？我们接着来看 「2. AutoConfigurationMetadataLoader」 。

2. AutoConfigurationMetadataLoader

`org.springframework.boot.autoconfigure.AutoConfigurationMetadataLoader` ，AutoConfigurationMetadata 加载器。其类上的注释如下：

```
/**
 * Internal utility used to load {@link AutoConfigurationMetadata}.
 */
```

2.1 loadMetadata

`#loadMetadata(ClassLoader classLoader)` 静态方法，加载 AutoConfigurationMetadata 。代码如下：

```
// AutoConfigurationMetadataLoader.java

protected static final String PATH = "META-INF/" + "spring-autoconfigure-metadata.properties";

public static AutoConfigurationMetadata loadMetadata(ClassLoader classLoader) {
    return loadMetadata(classLoader, PATH);
}

static AutoConfigurationMetadata loadMetadata(ClassLoader classLoader, String path) {
    try {
        // <1> 获得 PATH 对应的 URL 们
        Enumeration<URL> urls = (classLoader != null) ? classLoader.getResources(path) : ClassLoader.getSystemResourc
        // <2> 遍历 URL 数组，读取到 properties 中
        Properties properties = new Properties();
        while (urls.hasMoreElements()) {
            properties.putAll(PropertiesLoaderUtils.loadProperties(new UrlResource(urls.nextElement())));
        }
        // <3> 将 properties 转换成 PropertiesAutoConfigurationMetadata 对象
        return loadMetadata(properties);
    } catch (IOException ex) {
        throw new IllegalArgumentException("Unable to load @ConditionalOnClass location [" + path + "]", ex);
    }
}
```

<1> 处，获得 PATH 对应的 URL 们，而 PATH 就是 "META-INF/spring-autoconfigure-metadata.properties" 文件。这样，我们就可以避免去 AutoConfiguration 类上，读取其 Condition 条件了，从而避免将不符合条件的 AutoConfiguration 类的字节码，加载到 JVM 中。那么，此时就会有一个疑问，"META-INF/spring-autoconfigure-metadata.properties" 是怎么来的呢？答案我们在 「3. AutoConfigureAnnotationProcessor」 中说。
<2> 处，遍历 URL 数组，读取到 properties 中。

<3> 处，调用 `#loadMetadata(Properties properties)` 方法，将 properties 转换成 PropertiesAutoConfigurationMetadata 对象。代码如下：

```
// AutoConfigurationMetadataLoader.java

static AutoConfigurationMetadata loadMetadata(Properties properties) {
 return new PropertiesAutoConfigurationMetadata(properties);
}
```

- 关于 PropertiesAutoConfigurationMetadata 类，在 「2.2 」 中看。

2.2 PropertiesAutoConfigurationMetadata

PropertiesAutoConfigurationMetadata ，是 AutoConfigurationMetadataLoader 的内部静态类，实现 AutoConfigurationMetadata 接口，代码如下：

```
// AutoConfigurationMetadataLoader#PropertiesAutoConfigurationMetadata.java

/**
 * {@link AutoConfigurationMetadata} implementation backed by a properties file.
 */
private static class PropertiesAutoConfigurationMetadata implements AutoConfigurationMetadata {

    /**
     * Properties 对象
     */
    private final Properties properties;

    PropertiesAutoConfigurationMetadata(Properties properties) {
        this.properties = properties;
    }

    @Override
    public boolean wasProcessed(String className) {
        return this.properties.containsKey(className);
    }

    @Override
    public Integer getInteger(String className, String key) {
        return getInteger(className, key, null);
    }

    @Override
    public Integer getInteger(String className, String key, Integer defaultValue) {
        String value = get(className, key);
        return (value != null) ? Integer.valueOf(value) : defaultValue;
    }

    @Override
    public Set<String> getSet(String className, String key) {
        return getSet(className, key, null);
    }

    @Override
    public Set<String> getSet(String className, String key, Set<String> defaultValue) {
        String value = get(className, key);
        return (value != null) ? StringUtils.commaDelimitedListToSet(value) : defaultValue;
    }

    @Override
    public String get(String className, String key) {
        return get(className, key, null);
    }

    @Override
    public String get(String className, String key, String defaultValue) {
        String value = this.properties.getProperty(className + "." + key);
        return (value != null) ? value : defaultValue;
    }

}
```

3. AutoConfigureAnnotationProcessor

在 Spring Boot 的源码中，我们如果去检索 "spring-autoconfigure-metadata.properties" 文件，然而并找不到。是不是感觉很奇怪。于是乎，芳芳在搜索了一些网络的上的资料，原来是需要引入 spring-boot-autoconfigure-processor 依赖。这样，它的 AutoConfigureAnnotationProcessor 类，就会自动根据 AutoConfiguration 类的条件，生成 "META-INF/spring-autoconfigure-metadata.properties" 文件。

那么，此时又会有一个疑惑，那是什么时候生成呢？AutoConfigureAnnotationProcessor 继承自 javax.annotation.processing.AbstractProcessor 类，它可以在编译时，扫描和处理注解（Annotation），从而生成 "META-INF/spring-autoconfigure-metadata.properties" 文件。 很有意思。

> FROM 《Java 注解处理器》
>
> 注解处理器（Annotation Processor）是javac的一个工具，它用来在编译时扫描和处理注解（Annotation）。你可以对自定义注解，并注册相应的注解处理器。到这里，我假设你已经知道什么是注解，并且知道怎么申明的一个注解。如果你不熟悉注解，你可以在这官方文档中得到更多信息。注解处理器在Java 5开始就有了，但是从Java 6（2006年12月发布）开始才有可用的API。过了一些时间，Java世界才意识到注解处理器的强大作用，所以它到最近几年才流行起来。

那么，我们开始撸撸 AutoConfigureAnnotationProcessor 的源码吧。

org.springframework.boot.autoconfigureprocessor.AutoConfigureAnnotationProcessor ，继承 AbstractProcessor 抽象类，根据 AutoConfiguration 类的条件，生成 "META-INF/spring-autoconfigure-metadata.properties" 文件。其类上注释如下：

```
// AutoConfigureAnnotationProcessor.java

/**
 * Annotation processor to store certain annotations from auto-configuration classes in a
 * property file.
 */
```

3.1 构造方法

```
// AutoConfigureAnnotationProcessor.java

/**
 * 注解名和全类名的映射
 *
 * KEY：注解名
 * VALUE：全类名
 */
private final Map<String, String> annotations;
/**
 * 注解名和 ValueExtractor 的映射
 *
 * KEY：注解名
 */
private final Map<String, ValueExtractor> valueExtractors;

/**
```

```
     * 扫描和处理注解（Annotation），生成的 Properties 对象
     */
    private final Properties properties = new Properties();

    public AutoConfigureAnnotationProcessor() {
        // <1> 初始化 annotations 属性
        Map<String, String> annotations = new LinkedHashMap<>();
        addAnnotations(annotations);
        this.annotations = Collections.unmodifiableMap(annotations);
        // <2> 初始化 valueExtractors 属性
        Map<String, ValueExtractor> valueExtractors = new LinkedHashMap<>();
        addValueExtractors(valueExtractors);
        this.valueExtractors = Collections.unmodifiableMap(valueExtractors);
    }
```

`annotations` 属性，注解名和全类名的映射。在 `<1>` 处，调用 `#addAnnotations(Map<String, String> annotations)` 方法，进行初始化。代码如下：

```
    // AutoConfigureAnnotationProcessor.java

    protected void addAnnotations(Map<String, String> annotations) {
     // 条件
        annotations.put("Configuration", "org.springframework.context.annotation.Configuration");
        annotations.put("ConditionalOnClass", "org.springframework.boot.autoconfigure.condition.ConditionalOnClass");
        annotations.put("ConditionalOnBean", "org.springframework.boot.autoconfigure.condition.ConditionalOnBean");
        annotations.put("ConditionalOnSingleCandidate", "org.springframework.boot.autoconfigure.condition.Condition
        annotations.put("ConditionalOnWebApplication", "org.springframework.boot.autoconfigure.condition.Conditiona
     // 排序
        annotations.put("AutoConfigureBefore", "org.springframework.boot.autoconfigure.AutoConfigureBefore");
        annotations.put("AutoConfigureAfter", "org.springframework.boot.autoconfigure.AutoConfigureAfter");
        annotations.put("AutoConfigureOrder", "org.springframework.boot.autoconfigure.AutoConfigureOrder");
    }
```

`valueExtractors` 属性，注解名和 ValueExtractor 的映射。在 `<2>` 处，调用 `#addValueExtractors(Map<String, ValueExtractor> attributes)` 方法，进行初始化。代码如下：

```
    // AutoConfigureAnnotationProcessor.java

    private void addValueExtractors(Map<String, ValueExtractor> attributes) {
        attributes.put("Configuration", ValueExtractor.allFrom("value"));
        attributes.put("ConditionalOnClass", new OnClassConditionValueExtractor());
        attributes.put("ConditionalOnBean", new OnBeanConditionValueExtractor());
        attributes.put("ConditionalOnSingleCandidate", new OnBeanConditionValueExtractor());
        attributes.put("ConditionalOnWebApplication", ValueExtractor.allFrom("type"));
        attributes.put("AutoConfigureBefore", ValueExtractor.allFrom("value", "name"));
        attributes.put("AutoConfigureAfter", ValueExtractor.allFrom("value", "name"));
        attributes.put("AutoConfigureOrder", ValueExtractor.allFrom("value"));
    }
```

`properties` 属性，扫描和处理注解（Annotation），生成的 Properties 对象。

3.2 ValueExtractor

ValueExtractor ，是 AutoConfigureAnnotationProcessor 的内部接口，值提取器接口。代码如下

:

```
// AutoConfigureAnnotationProcessor#ValueExtractor.java

@FunctionalInterface
private interface ValueExtractor {

    /**
     * 从注解中，获得对应的值的数组
     *
     * @param annotation 注解
     * @return 值的数组
     */
    List<Object> getValues(AnnotationMirror annotation);

    /**
     * 创建 NamedValuesExtractor 对象
     *
     * @param names 从注解的指定 names 中，提取值们
     * @return NamedValuesExtractor 对象
     */
    static ValueExtractor allFrom(String... names) {
        return new NamedValuesExtractor(names);
    }

}
```

3.2.1 AbstractValueExtractor

AbstractValueExtractor ，是 AutoConfigureAnnotationProcessor 的内部类，实现 ValueExtractor 接口，ValueExtractor 抽象实现类。代码如下：

```
// AutoConfigureAnnotationProcessor#AbstractValueExtractor.java

private abstract static class AbstractValueExtractor implements ValueExtractor {

    @SuppressWarnings("unchecked")
    protected Stream<Object> extractValues(AnnotationValue annotationValue) {
        // 注解值为空，返回空
        if (annotationValue == null) {
            return Stream.empty();
        }
        Object value = annotationValue.getValue();
        // 注解值为数组，则遍历数组，逐个提取值
        if (value instanceof List) {
            return ((List<AnnotationValue>) value).stream()
                    .map((annotation) -> extractValue(annotation.getValue()));
        }
        // 注解值非数组，直接提取值
        return Stream.of(extractValue(value));
    }

    private Object extractValue(Object value) {
        if (value instanceof DeclaredType) {
            return Elements.getQualifiedName(((DeclaredType) value).asElement());
        }
        return value;
```

```
    }

}
```

提供了从 AnnotationValue 读取值的公用方法。

3.2.2 NamedValuesExtractor

NamedValuesExtractor ，是 AutoConfigureAnnotationProcessor 的内部类，继承 AbstractValueExtractor 抽象类，读取 names 的 ValueExtractor 实现类。代码如下：

```
// AutoConfigureAnnotationProcessor#NamedValuesExtractor.java

private static class NamedValuesExtractor extends AbstractValueExtractor {

    private final Set<String> names;

    NamedValuesExtractor(String... names) {
        this.names = new HashSet<>(Arrays.asList(names));
    }

    @Override
    public List<Object> getValues(AnnotationMirror annotation) {
        List<Object> result = new ArrayList<>();
        // 遍历 names 数组，读取 name 对应的值，添加到 result 中
        annotation.getElementValues().forEach((key, value) -> {
            if (this.names.contains(key.getSimpleName().toString())) {
                extractValues(value).forEach(result::add);
            }
        });
        return result;
    }

}
```

3.2.3 OnBeanConditionValueExtractor

OnBeanConditionValueExtractor ，是 AutoConfigureAnnotationProcessor 的内部类，继承 AbstractValueExtractor 抽象类，读取 @ConditionalOnBean 和 @ConditionalOnSingleCandidate 注解的 ValueExtractor 实现类。代码如下：

```
// AutoConfigureAnnotationProcessor#OnBeanConditionValueExtractor.java

private static class OnBeanConditionValueExtractor extends AbstractValueExtractor {

    @Override
    public List<Object> getValues(AnnotationMirror annotation) {
        // 遍历注解的元素们，读取到 attributes 中
        Map<String, AnnotationValue> attributes = new LinkedHashMap<>();
        annotation.getElementValues().forEach((key, value) -> attributes.put(key.getSimpleName().toString(), value));
        // 如果 "name" 对应的值为空，返回空
        if (attributes.containsKey("name")) {
            return Collections.emptyList();
        }
```

```
            // 读取 "value"、`"type"` 对应的值，添加到 result 中
            List<Object> result = new ArrayList<>();
            extractValues(attributes.get("value")).forEach(result::add);
            extractValues(attributes.get("type")).forEach(result::add);
            return result;
        }

    }
```

3.2.4 OnClassConditionValueExtractor

OnClassConditionValueExtractor ，是 AutoConfigureAnnotationProcessor 的内部类，继承 NamedValuesExtractor 类，读取 `@OnClassConditionValueExtractor` 注解的 ValueExtractor 实现类。代码如下：

```
    // AutoConfigureAnnotationProcessor#OnClassConditionValueExtractor.java

    private static class OnClassConditionValueExtractor extends NamedValuesExtractor {

        OnClassConditionValueExtractor() {
            super("value", "name");
        }

        @Override
        public List<Object> getValues(AnnotationMirror annotation) {
            // 读取 "value", "name" 的值
            List<Object> values = super.getValues(annotation);
            // 排序
            values.sort(this::compare);
            return values;
        }

        private int compare(Object o1, Object o2) {
            return Comparator.comparing(this::isSpringClass)
                    .thenComparing(String.CASE_INSENSITIVE_ORDER)
                    .compare(o1.toString(), o2.toString());
        }

        private boolean isSpringClass(String type) {
            return type.startsWith("org.springframework");
        }

    }
```

3.3 process

实现 `#process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv)` 方法，进行处理。代码如下：

```
    // AutoConfigureAnnotationProcessor.java

    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {
        // <1> 遍历 annotations 集合，逐个处理
```

```
    for (Map.Entry<String, String> entry : this.annotations.entrySet()) {
        process(roundEnv, entry.getKey(), entry.getValue());
    }
 // <2> 处理完成，写到文件 PROPERTIES_PATH 中
 if (roundEnv.processingOver()) {
     try {
            writeProperties();
        } catch (Exception ex) {
         throw new IllegalStateException("Failed to write metadata", ex);
        }
    }
 return false;
}
```

<1> 处，调用 #process(RoundEnvironment roundEnv, String propertyKey, String annotationName) 方法，遍历 annotations 集合，逐个处理，添加到 properties 中。代码如下：

```
// AutoConfigureAnnotationProcessor.java

private void process(RoundEnvironment roundEnv, String propertyKey, String annotationName) {
    TypeElement annotationType = this.processingEnv.getElementUtils().getTypeElement(annotationName);
    if (annotationType != null) {
        for (Element element : roundEnv.getElementsAnnotatedWith(annotationType)) {
            Element enclosingElement = element.getEnclosingElement();
            if (enclosingElement != null
                    && enclosingElement.getKind() == ElementKind.PACKAGE) {
                processElement(element, propertyKey, annotationName);
            }
        }
    }
}

// propertyKey=注解名
// annotationName=全类名
private void processElement(Element element, String propertyKey, String annotationName) {
    try {
        // 获得自动配置类的全类名。例如说：org.springframework.boot.autoconfigure.transaction.TransactionAutoCo
        String qualifiedName = Elements.getQualifiedName(element);
        // 获得 AnnotationMirror 对象
        AnnotationMirror annotation = getAnnotation(element, annotationName);
        if (qualifiedName != null && annotation != null) {
            // 获得值
            List<Object> values = getValues(propertyKey, annotation);
            // 添加到 properties 中
            this.properties.put(qualifiedName + "." + propertyKey, toCommaDelimitedString(values));
            // 添加到 properties 中
            this.properties.put(qualifiedName, "");
        }
    } catch (Exception ex) {
        throw new IllegalStateException("Error processing configuration meta-data on " + element, ex);
    }
}

// 获得 AnnotationMirror 对象
private AnnotationMirror getAnnotation(Element element, String type) {
    if (element != null) {
        for (AnnotationMirror annotation : element.getAnnotationMirrors()) {
            if (type.equals(annotation.getAnnotationType().toString())) {
```

```
                    return annotation;
                }
            }
        }
        return null;
    }

    // 获得值
    private List<Object> getValues(String propertyKey, AnnotationMirror annotation) {
        ValueExtractor extractor = this.valueExtractors.get(propertyKey);
        if (extractor == null) {
            return Collections.emptyList();
        }
        return extractor.getValues(annotation);
    }

    // 拼接值
    private String toCommaDelimitedString(List<Object> list) {
        StringBuilder result = new StringBuilder();
        for (Object item : list) {
            result.append((result.length() != 0) ? "," : "");
            result.append(item);
        }
        return result.toString();
    }
```

- 胖友简单瞅两眼即可，不是很重要哈~

<2> 处，调用 `#writeProperties()` 方法，处理完成，写到文件 `PROPERTIES_PATH` 中。代码如下：

```
    // AutoConfigureAnnotationProcessor.java

    protected static final String PROPERTIES_PATH = "META-INF/" + "spring-autoconfigure-metadata.properties";

    private void writeProperties() throws IOException {
        if (!this.properties.isEmpty()) {
            // 创建 FileObject 对象
            FileObject file = this.processingEnv.getFiler().createResource(StandardLocation.CLASS_OUTPUT, "", PROPE
            // 写入 properties 到文件
            try (OutputStream outputStream = file.openOutputStream()) {
                this.properties.store(outputStream, null);
            }
        }
    }
```

- 这不，和 「2. AutoConfigurationMetadataLoader」 就对上列。

666. 彩蛋

简单小文一篇~在 `spring-boot-autoconfigure-processor` 的寻找上，花了一些些时间。HOHO ~

文章目录

回到首页