
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
2021-01-10
Spring Boot

精尽 Spring Boot 源码分析 —— 自动配置

1. 概述

本文，我们来分享 Spring Boot 自动配置的实现源码。在故事的开始，我们先来说两个事情：

> 自动配置和自动装配的区别？
> Spring Boot 配置的原理

2. 自动配置 V.S 自动装配

在这篇文章的开始，芳芳是有点混淆自动配置和自动装配的概念，后来经过 Google 之后，发现两者是截然不如同的：

> 自动配置：是 Spring Boot 提供的，实现通过 jar 包的依赖，能够自动配置应用程序。例如说：我们引入 `spring-boot-starter-web` 之后，就自动引入了 Spring MVC 相关的 jar 包，从而自动配置 Spring MVC 。
> 自动装配：是 Spring 提供的 IoC 注入方式，具体看看 《Spring 教程 —— Beans 自动装配》 文档。

所以，不要和芳芳一样愚蠢的搞错落。

3. 自动装配原理

胖友可以直接看 《详解 Spring Boot 自动配置机制》 文章的 「二、Spring Boot 自动配置」 小节，芳芳觉得写的挺清晰的。

> 下面，我们即开始正式撸具体的代码实现了。

4. @SpringBootApplication

`org.springframework.boot.autoconfigure.@SpringBootApplication` 注解，基本我们的 Spring Boot 应用，一定会去有这样一个注解。并且，通过使用它，不仅仅能标记这是一个 Spring Boot 应用，而且能够开启自动配置的功能。这是为什么呢？

`@SpringBootApplication` 注解，它在 `spring-boot-autoconfigure` 模块中。所以，我们使用 Spring Boot 项目时，如果不想使用自动配置功能，就不用引入它。当然，我们貌似不太会存在这样的需求，是吧~

`@SpringBootApplication` 是一个组合注解。代码如下：

```
// SpringBootApplication.java

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
     @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
     @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {

 /**
     * Exclude specific auto-configuration classes such that they will never be applied.
     * @return the classes to exclude
     */
 @AliasFor(annotation = EnableAutoConfiguration.class)
    Class<?>[] exclude() default {};

 /**
     * Exclude specific auto-configuration class names such that they will never be
     * applied.
     * @return the class names to exclude
     * @since 1.3.0
     */
 @AliasFor(annotation = EnableAutoConfiguration.class)
    String[] excludeName() default {};

 /**
     * Base packages to scan for annotated components. Use {@link #scanBasePackageClasses}
     * for a type-safe alternative to String-based package names.
     * @return base packages to scan
     * @since 1.3.0
     */
 @AliasFor(annotation = ComponentScan.class, attribute = "basePackages")
    String[] scanBasePackages() default {};

 /**
     * Type-safe alternative to {@link #scanBasePackages} for specifying the packages to
     * scan for annotated components. The package of each class specified will be scanned.
     * <p>
     * Consider creating a special no-op marker class or interface in each package that
     * serves no purpose other than being referenced by this attribute.
     * @return base packages to scan
     * @since 1.3.0
     */
 @AliasFor(annotation = ComponentScan.class, attribute = "basePackageClasses")
    Class<?>[] scanBasePackageClasses() default {};

}
```

下面，我们来逐个看 `@SpringBootApplication` 上的每个注解。

4.1 @Inherited

Java 自带的注解。

java.lang.annotation.@Inherited 注解，使用此注解声明出来的自定义注解，在使用此自定义注解时，如果注解在类上面时，子类会自动继承此注解，否则的话，子类不会继承此注解。

这里一定要记住，使用 @Inherited 声明出来的注解，只有在类上使用时才会有效，对方法，属性等其他无效。

不了解的胖友，可以看看 《关于 Java 注解中元注解 Inherited 的使用详解》 文章。

4.2 @SpringBootConfiguration

Spring Boot 自定义的注解

org.springframework.boot.@SpringBootConfiguration 注解，标记这是一个 Spring Boot 配置类。代码如下：

```
// SpringBootConfiguration.java

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration
public @interface SpringBootConfiguration {

}
```

可以看到，它上面继承自 @Configuration 注解，所以两者功能也一致，可以将当前类内声明的一个或多个以 @Bean 注解标记的方法的实例纳入到 Srping 容器中，并且实例名就是方法名。

4.3 @ComponentScan

Spring 自定义的注解

org.springframework.context.annotation.@ComponentScan 注解，扫描指定路径下的 Component（@Componment、@Configuration、@Service 等等）。

不了解的胖友，可以看看 《Spring：@ComponentScan 使用》 文章。

4.4 @EnableAutoConfiguration

Spring Boot 自定义的注解

org.springframework.boot.autoconfigure.@EnableAutoConfiguration 注解，用于开启自动配置功能，是 spring-boot-autoconfigure 项目最核心的注解。代码如下：

```
// EnableAutoConfiguration.java

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
```

```
@Documented
@Inherited
@AutoConfigurationPackage
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {

    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";

 /**
     * Exclude specific auto-configuration classes such that they will never be applied.
     * @return the classes to exclude
     */
    Class<?>[] exclude() default {};

 /**
     * Exclude specific auto-configuration class names such that they will never be
     * applied.
     * @return the class names to exclude
     * @since 1.3.0
     */
    String[] excludeName() default {};

}
```

org.springframework.boot.autoconfigure.@AutoConfigurationPackage 注解，主要功能自动配置包，它会获取主程序类所在的包路径，并将包路径（包括子包）下的所有组件注册到 Spring IOC 容器中。代码如下：

```
// AutoConfigurationPackage.java

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Import(AutoConfigurationPackages.Registrar.class)
public @interface AutoConfigurationPackage {

}
```

- org.springframework.context.annotation.@Import 注解，可用于资源的导入。情况比较多，可以看看 《6、@Import 注解——导入资源》 文章。
- AutoConfigurationPackages.Registrar ，有点神奇，这里先不说。胖友最后去看 「6. AutoConfigurationPackages」 小节。

@Import(AutoConfigurationImportSelector.class) 注解部分，是重头戏的开始。

- org.springframework.context.annotation.@Import 注解，可用于资源的导入。情况比较多，可以看看 《6、@Import 注解——导入资源》 文章。
- AutoConfigurationImportSelector ，导入自动配置相关的资源。详细解析，见 「5. AutoConfigurationImportSelector」 小节。

5. AutoConfigurationImportSelector

org.springframework.boot.autoconfigure.AutoConfigurationImportSelector ，实现 DeferredImportSelector、

BeanClassLoaderAware、ResourceLoaderAware、BeanFactoryAware、EnvironmentAware、Ordered 接口，处理 `@EnableAutoConfiguration` 注解的资源导入。

5.1 getCandidateConfigurations

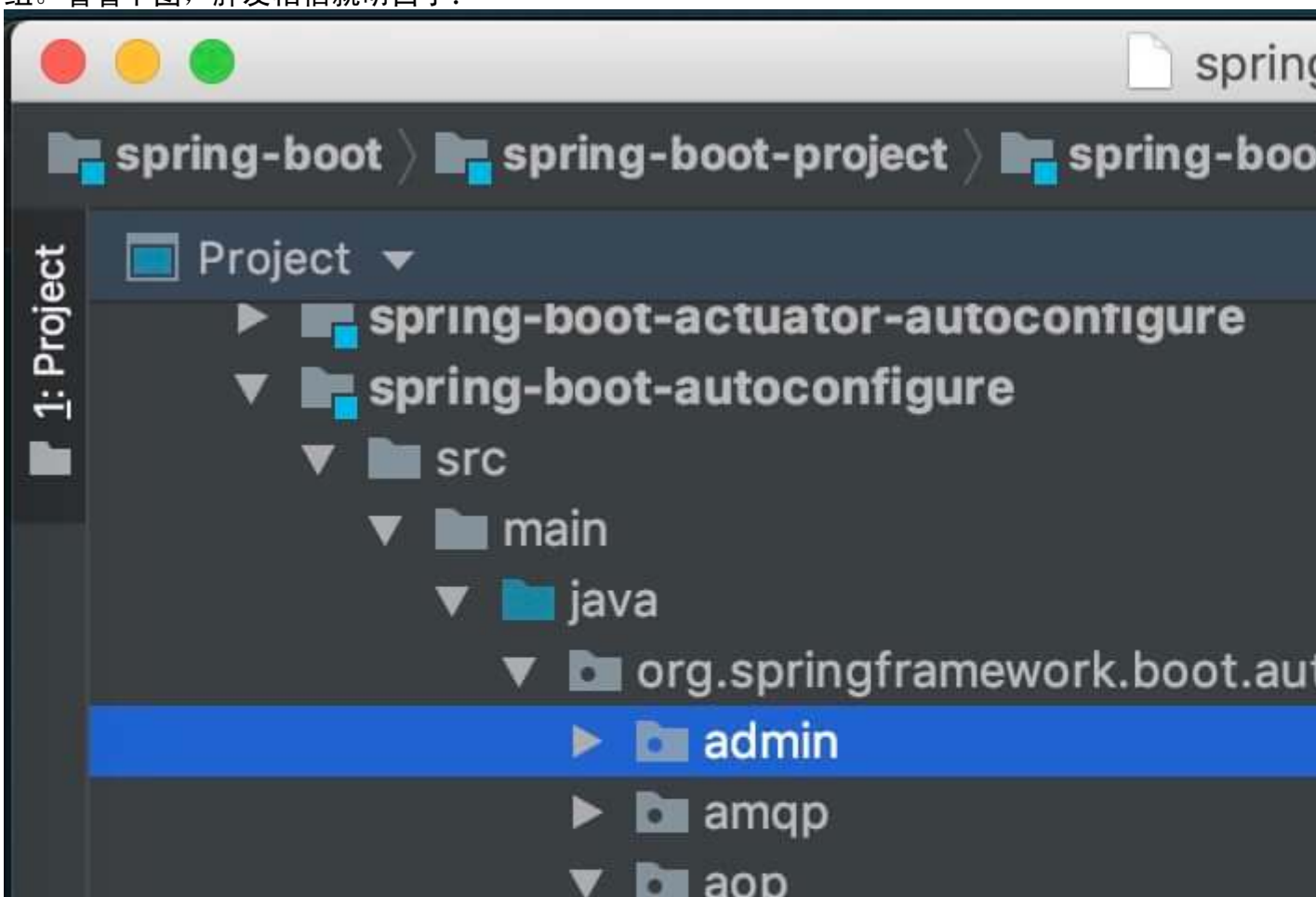
`#getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes)` 方法，获得符合条件的配置类的数组。代码如下：

```
// AutoConfigurationImportSelector.java

protected List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes) {
    // <1> 加载指定类型 EnableAutoConfiguration 对应的，在 `META-INF/spring.factories` 里的类名的数组
    List<String> configurations = SpringFactoriesLoader.loadFactoryNames(getSpringFactoriesLoaderFactoryClass(), getB
 // 断言，非空
    Assert.notEmpty(configurations, "No auto configuration classes found in META-INF/spring.factories. If you " + "ar
 return configurations;
}
```

<1> 处，调用 `#getSpringFactoriesLoaderFactoryClass()` 方法，获得要从 `META-INF/spring.factories` 加载的指定类型为 EnableAutoConfiguration 类。代码如下：

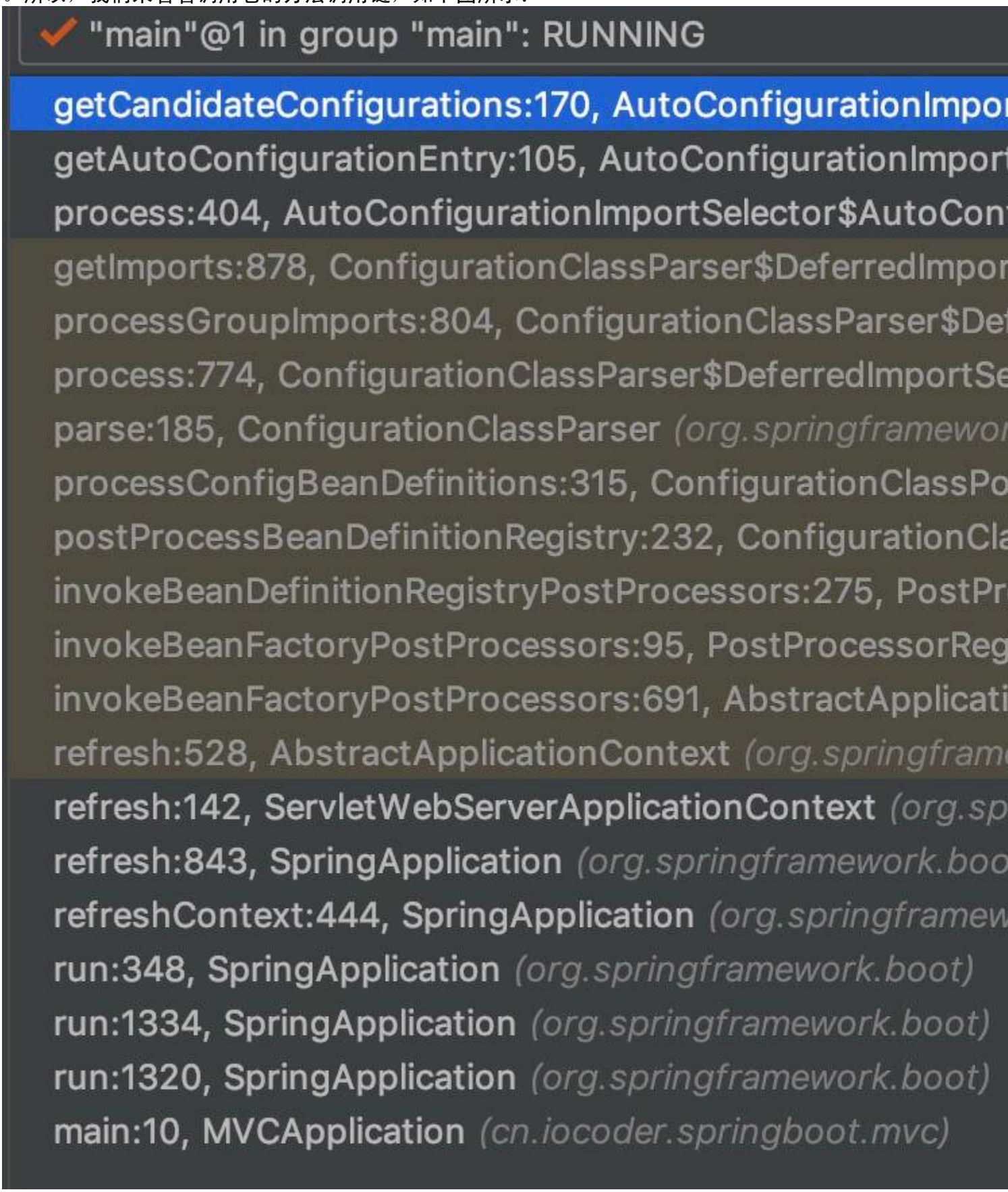
```
// AutoConfigurationImportSelector.java

protected Class<?> getSpringFactoriesLoaderFactoryClass() {
 return EnableAutoConfiguration.class;
}
```

<1> 处，调用 `SpringFactoriesLoader#loadFactoryNames(Class<?> factoryClass, ClassLoader classLoader)` 方法，加载指定类型 EnableAutoConfiguration 对应的，在 `META-INF/spring.factories` 里的类名的数组。看看下图，胖友相信就明白了：



一般来说，和网络上 Spring Boot 敢于这块的源码解析，我们就可以结束了。如果单纯是为了了解原理 Spring Boot 自动配置的原理，这里结束也是没问题的。因为，拿到 Configuration 配置类后，后面的就是 Spring Java Config 的事情了。不了解的胖友，可以看看 《Spring 教程 —— 基于 Java 的配置》 文章。

但是（“但是”同学，你赶紧坐下），具有倒腾精神的芮芮，觉得还是继续瞅瞅 `#getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes)` 方法是怎么被调用的。所以，我们来看看调用它的方法调用链，如下图所示：



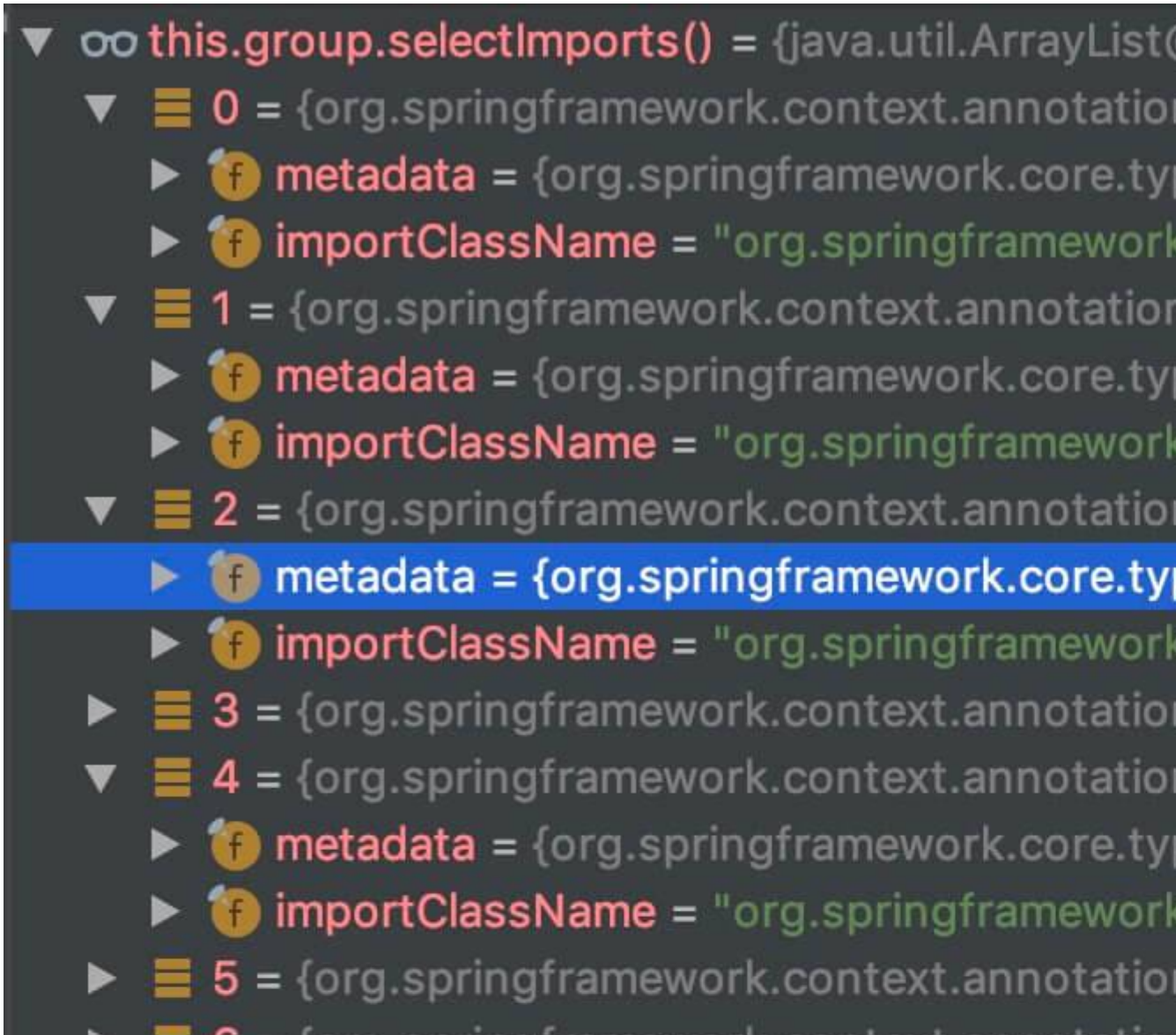
① 处，refresh 方法的调用，我们在 《精尽 Spring Boot 源码分析 —— SpringApplication》 中，SpringApplication 启动时，会调用到该方法。
② 处，#getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes) 方法被调用。

③ 处，那么此处，就是问题的关键。代码如下：

> 芮芮：因为我还没特别完整的撸完 Spring Java Annotations 相关的源码，所以下面的部分，我们更多是看整个调用过程。　　恰好，胖友也没看过，哈哈哈哈。

```
// ConfigurationClassParser#DeferredImportSelectorGroupingHandler.java

private final DeferredImportSelector.Group group;

public Iterable<Group.Entry> getImports() {
 for (DeferredImportSelectorHolder deferredImport : this.deferredImports) {
     this.group.process(deferredImport.getConfigurationClass().getMetadata(),
                deferredImport.getImportSelector()); // <1>
    }
 return this.group.selectImports(); // <2>
}
```

- <1> 处，调用 DeferredImportSelector.Group#process(AnnotationMetadata metadata, DeferredImportSelector selector) 方法，处理被 @Import 注解的注解。
- <2> 处，调用 DeferredImportSelector.Group#this.group.selectImports() 方法，选择需要导入的。例如：



    - 这里，我们可以看到需要导入的 Configuration 配置类。
  - 具体 <1> 和 <2> 处，在 「5.3 AutoConfigurationGroup」 详细解析。

5.2 getImportGroup

#getImportGroup() 方法，获得对应的 Group 实现类。代码如下：

```
// AutoConfigurationImportSelector.java

@Override // 实现自 DeferredImportSelector 接口
public Class<? extends Group> getImportGroup() {
 return AutoConfigurationGroup.class;
}
```

关于 AutoConfigurationGroup 类，在 「5.3 AutoConfigurationGroup」 详细解析。

5.3 AutoConfigurationGroup

芮芮：注意，从这里开始后，东西会比较难。因为，涉及的东西会比较多。

AutoConfigurationGroup ，是 AutoConfigurationImportSelector 的内部类，实现 DeferredImportSelector.Group、BeanClassLoaderAware、BeanFactoryAware、ResourceLoaderAware 接口，自动配置的 Group 实现类。

5.3.1 属性

```
// AutoConfigurationImportSelector#AutoConfigurationGroup.java

/**
 * AnnotationMetadata 的映射
 *
 * KEY：配置类的全类名
 */
private final Map<String, AnnotationMetadata> entries = new LinkedHashMap<>();
/**
 * AutoConfigurationEntry 的数组
 */
private final List<AutoConfigurationEntry> autoConfigurationEntries = new ArrayList<>();

private ClassLoader beanClassLoader;
private BeanFactory beanFactory;
private ResourceLoader resourceLoader;

/**
 * 自动配置的元数据
 */
private AutoConfigurationMetadata autoConfigurationMetadata;
```

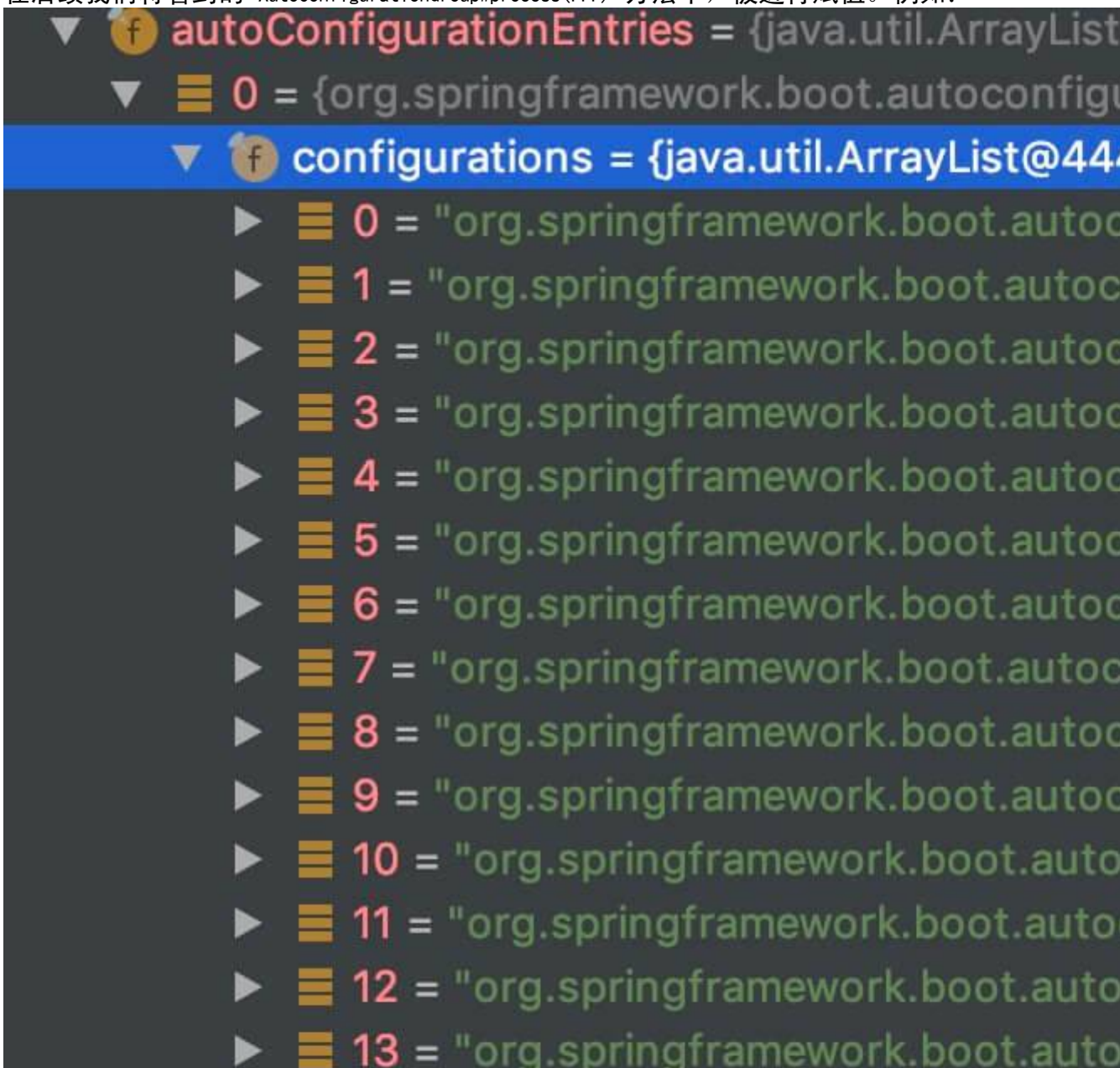
entries 属性，AnnotationMetadata 的映射。其中，KEY 为 配置类的全类名。在后续我们将看到的 AutoConfigurationGroup#process(...) 方法中，被进行赋值。例如：

autoConfigurationEntries 属性，AutoConfigurationEntry 的数组。

- 其中，AutoConfigurationEntry 是 AutoConfigurationImportSelector 的内部类，自动配置的条目。代码如下：

```
// AutoConfigurationImportSelector#AutoConfigurationEntry.java

protected static class AutoConfigurationEntry {

    /**
     * 配置类的全类名的数组
     */
    private final List<String> configurations;
    /**
     * 排除的配置类的全类名的数组
     */
    private final Set<String> exclusions;

    // 省略构造方法和 setting/getting 方法

}
```

  - 属性比较简单。
- 在后续我们将看到的 AutoConfigurationGroup#process(...) 方法中，被进行赋值。例如：

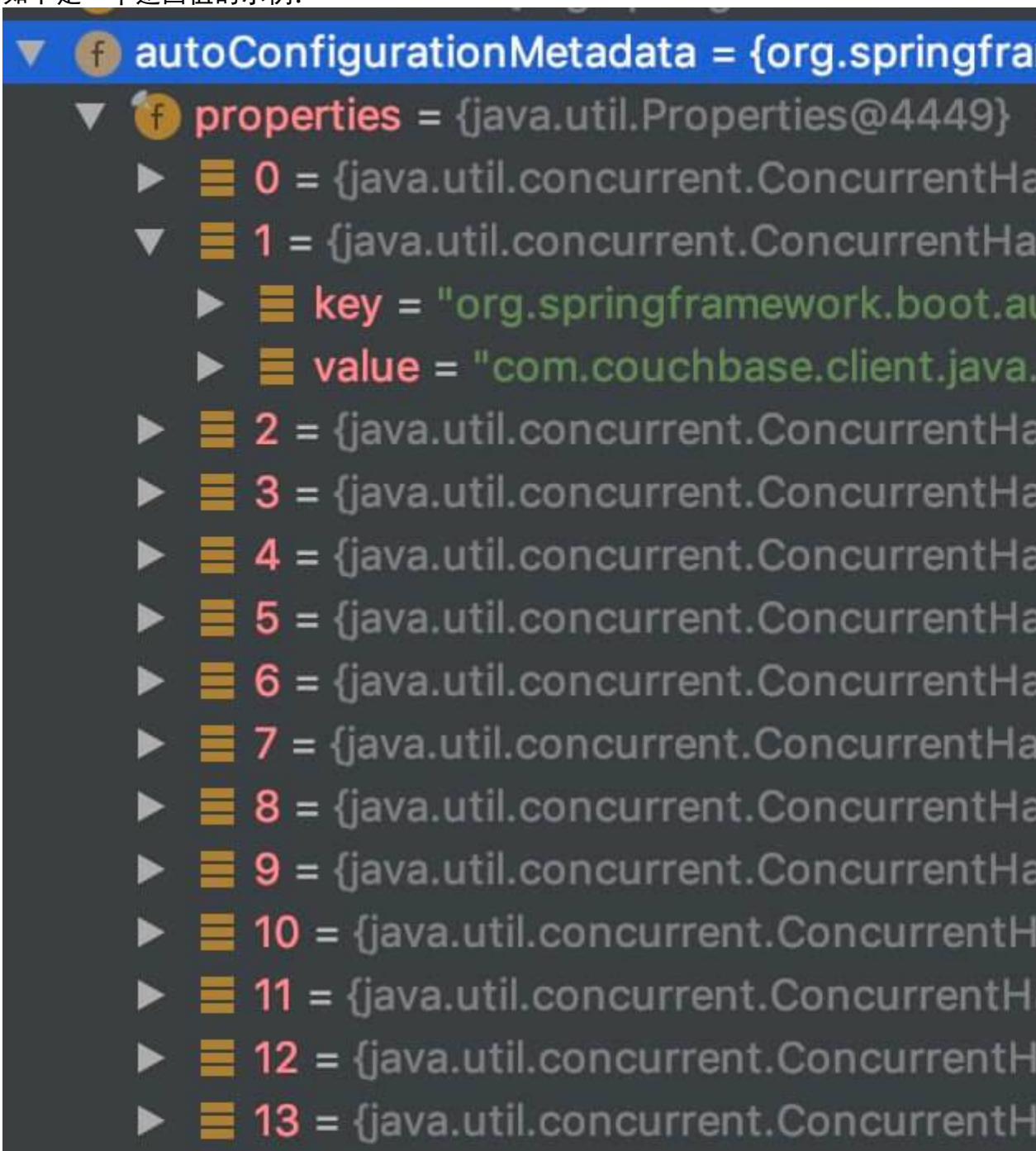


autoConfigurationMetadata 属性，自动配置的元数据（Metadata）。

- 通过 #getAutoConfigurationMetadata() 方法，会初始化该属性。代码如下：

```
// AutoConfigurationImportSelector#AutoConfigurationGroup.java

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;
}
```

- 关于 AutoConfigurationMetadataLoader 类，我们先不去愁。避免，我们调试的太过深入。TODO 后续在补充下。
- 返回的类型是 PropertiesAutoConfigurationMetadata ，比较简单，胖友点击 传送门 瞅一眼即可。
- 如下是一个返回值的示例：



        - 可能胖友会有点懵逼，这么多，并且 KEY / VALUE 结果看不懂？不要方，我们简单来说下 CouchbaseReactiveRepositoriesAutoConfiguration 配置类。如果它生效，需要 classpath 下有 Bucket、ReactiveCouchbaseRepository、Flux 三个类，所以红线那个条目，对应的就是 CouchbaseReactiveRepositoriesAutoConfiguration 类上的 `@ConditionalOnClass({ Bucket.class, ReactiveCouchbaseRepository.class, Flux.class })` 注解部分。
    - 所以，`autoConfigurationMetadata` 属性，用途就是制定配置类（Configuration）的生效条件（Condition）。

5.3.2 process

`#process(AnnotationMetadata annotationMetadata, DeferredImportSelector deferredImportSelector)` 方法，进行处理。代码如下：

```
// AutoConfigurationImportSelector#AutoConfigurationGroup.java

@Override
public void process(AnnotationMetadata annotationMetadata, DeferredImportSelector deferredImportSelector) {
    // 断言
    Assert.state(
            deferredImportSelector instanceof AutoConfigurationImportSelector,
            () -> String.format("Only %s implementations are supported, got %s",
                    AutoConfigurationImportSelector.class.getSimpleName(),
                    deferredImportSelector.getClass().getName()));
    // <1> 获得 AutoConfigurationEntry 对象
    AutoConfigurationEntry autoConfigurationEntry = ((AutoConfigurationImportSelector) deferredImportSelector)
            .getAutoConfigurationEntry(getAutoConfigurationMetadata(), annotationMetadata);
    // <2> 添加到 autoConfigurationEntries 中
    this.autoConfigurationEntries.add(autoConfigurationEntry);
    // <3> 添加到 entries 中
    for (String importClassName : autoConfigurationEntry.getConfigurations()) {
        this.entries.putIfAbsent(importClassName, annotationMetadata);
    }
}
```

`annotationMetadata` 参数，一般来说是被 `@SpringBootApplication` 注解的元数据。因为，`@SpringBootApplication` 组合了 `@EnableAutoConfiguration` 注解。
`deferredImportSelector` 参数，`@EnableAutoConfiguration` 注解的定义的 `@Import` 的类，即 AutoConfigurationImportSelector 对象。
<1> 处，调用 `AutoConfigurationImportSelector#getAutoConfigurationEntry(AutoConfigurationMetadata autoConfigurationMetadata, AnnotationMetadata annotationMetadata)` 方法，获得 AutoConfigurationEntry 对象。详细解析，见 「5.4 AutoConfigurationEntry」 。因为这块比较重要，所以先跳过去瞅瞅。
<2> 处，添加到 `autoConfigurationEntries` 中。
<3> 处，添加到 `entries` 中。

5.3.3 selectImports

`#selectImports()` 方法，获得要引入的配置类。代码如下：

```
// AutoConfigurationImportSelector#AutoConfigurationGroup.java
```

```
@Override
public Iterable<Entry> selectImports() {
    // <1> 如果为空，则返回空数组
    if (this.autoConfigurationEntries.isEmpty()) {
        return Collections.emptyList();
    }
    // <2.1> 获得 allExclusions
    Set<String> allExclusions = this.autoConfigurationEntries.stream()
            .map(AutoConfigurationEntry::getExclusions)
            .flatMap(Collection::stream).collect(Collectors.toSet());
    // <2.2> 获得 processedConfigurations
    Set<String> processedConfigurations = this.autoConfigurationEntries.stream()
            .map(AutoConfigurationEntry::getConfigurations)
            .flatMap(Collection::stream)
            .collect(Collectors.toCollection(LinkedHashSet::new));
    // <2.3> 从 processedConfigurations 中，移除排除的
    processedConfigurations.removeAll(allExclusions);
    // <3> 处理，返回结果
    return sortAutoConfigurations(processedConfigurations, getAutoConfigurationMetadata()) // <3.1> 排序
                .stream()
                .map((importClassName) -> new Entry(this.entries.get(importClassName), importClassName)) // <3.2> 创建
                .collect(Collectors.toList()); // <3.3> 转换成 List
}
```

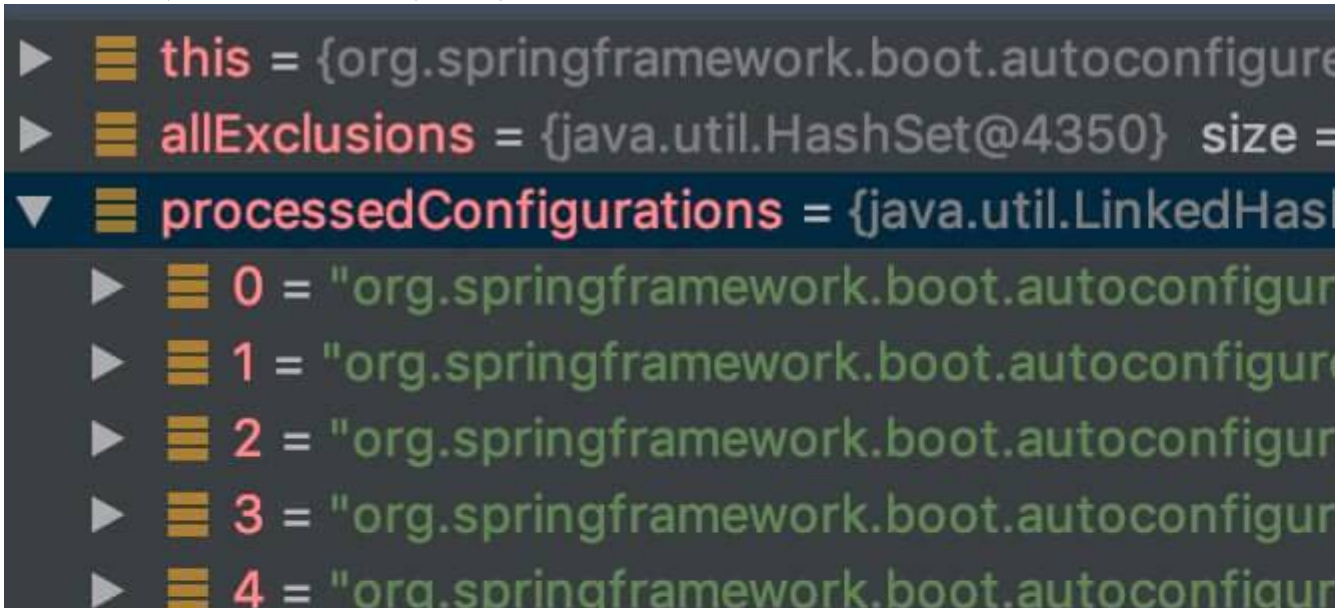
<1> 处，如果为空，则返回空数组。
<2.1>、<2.2>、<2.3> 处，获得要引入的配置类集合。　　比较奇怪的是，上面已经做过一次移除的处理，这里又做一次。不过，没多大关系，可以先无视。

<3> 处，处理，返回结果。

- <3.1> 处，调用 #sortAutoConfigurations(Set<String> configurations, AutoConfigurationMetadata autoConfigurationMetadata) 方法，排序。代码如下：

```
// AutoConfigurationImportSelector#AutoConfigurationGroup.java

private List<String> sortAutoConfigurations(Set<String> configurations, AutoConfigurationMetadata autoCo
 return new AutoConfigurationSorter(getMetadataReaderFactory(), autoConfigurationMetadata).getInPriority
}
```

  - 具体的排序逻辑，胖友自己看。实际上，还是涉及哪些，例如说 @Order 注解。
- <3.2> 处，创建 Entry 对象。
- <3.3> 处，转换成 List 。结果如下图：



芳芳：略微有点艰难的过程。不过回过头来，其实也没啥特别复杂的逻辑。是不，胖友~

5.4 getAutoConfigurationEntry

芳芳：这是一个关键方法。因为会调用到，我们会在 「5.1 getCandidateConfigurations」 的方法。

#getAutoConfigurationEntry(AutoConfigurationMetadata autoConfigurationMetadata, AnnotationMetadata annotationMetadata) 方法，获得 AutoConfigurationEntry 对象。代码如下：

```
// AutoConfigurationImportSelector.java

protected AutoConfigurationEntry getAutoConfigurationEntry(AutoConfigurationMetadata autoConfigurationMetadata, Annot
    // <1> 判断是否开启。如未开启，返回空数组。
    if (!isEnabled(annotationMetadata)) {
        return EMPTY_ENTRY;
    }
    // <2> 获得注解的属性
    AnnotationAttributes attributes = getAttributes(annotationMetadata);
    // <3> 获得符合条件的配置类的数组
    List<String> configurations = getCandidateConfigurations(annotationMetadata, attributes);
    // <3.1> 移除重复的配置类
    configurations = removeDuplicates(configurations);
    // <4> 获得需要排除的配置类
    Set<String> exclusions = getExclusions(annotationMetadata, attributes);
    // <4.1> 校验排除的配置类是否合法
    checkExcludedClasses(configurations, exclusions);
    // <4.2> 从 configurations 中，移除需要排除的配置类
    configurations.removeAll(exclusions);
    // <5> 根据条件（Condition），过滤掉不符合条件的配置类
    configurations = filter(configurations, autoConfigurationMetadata);
    // <6> 触发自动配置类引入完成的事件
    fireAutoConfigurationImportEvents(configurations, exclusions);
    // <7> 创建 AutoConfigurationEntry 对象
    return new AutoConfigurationEntry(configurations, exclusions);
}
```

这里每一步都是细节的方法，所以会每一个方法，都会是引导到对应的小节的方法。

虽然有点长，但是很不复杂。简单的来说，加载符合条件的配置类们，然后移除需要排除（exclusion）的。

<1> 处，调用 #isEnabled(AnnotationMetadata metadata) 方法，判断是否开启。如未开启，返回空数组。详细解析，见 「5.4.1 isEnabled」 。
<2> 处，调用 #getAttributes(AnnotationMetadata metadata) 方法，获得注解的属性。详细解析，见 「5.4.2 getAttributes」 。

【重要】<3> 处，调用 #getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes) 方法，获得符合条件的配置类的数组。

嘻嘻，到达此书之后，整个细节是不是就串起来了！

- <3.1> 处，调用 #removeDuplicates(List<T> list) 方法，移除重复的配置类。代码如下：

```
// AutoConfigurationImportSelector.java

protected final <T> List<T> removeDuplicates(List<T> list) {
 return new ArrayList<>(new LinkedHashSet<>(list));
}
```

- 简单粗暴

<4> 处，调用 `#getExclusions(AnnotationMetadata metadata, AnnotationAttributes attributes)` 方法，获得需要排除的配置类。详细解析，见 「5.4.3 getExclusions」 。

- <4.1> 处，调用 `#checkExcludedClasses(List<String> configurations, Set<String> exclusions)` 方法，校验排除的配置类是否合法。详细解析，见 「5.4.4 checkExcludedClasses」 。
- <4.2> 处，从 `configurations` 中，移除需要排除的配置类。

<5> 处，调用 `#filter(List<String> configurations, AutoConfigurationMetadata autoConfigurationMetadata)` 方法，根据条件（Condition），过滤掉不符合条件的配置类。详细解析，见 《精尽 Spring Boot 源码分析 —— Condition》 文章。

<6> 处，调用 `#fireAutoConfigurationImportEvents(List<String> configurations, Set<String> exclusions)` 方法，触发自动配置类引入完成的事件。详细解析，见 「5.4.5 fireAutoConfigurationImportEvents」 。
<7> 处，创建 AutoConfigurationEntry 对象。

整个 「5.4 getAutoConfigurationEntry」 看完后，胖友请跳回 「5.3.3 selectImports」 。

5.4.1 isEnabled

`#isEnabled(AnnotationMetadata metadata)` 方法，判断是否开启自动配置。代码如下：

```
// AutoConfigurationImportSelector.java

protected boolean isEnabled(AnnotationMetadata metadata) {
    // 判断 "spring.boot.enableautoconfiguration" 配置判断，是否开启自动配置。
    // 默认情况下（未配置），开启自动配置。
 if (getClass() == AutoConfigurationImportSelector.class) {
     return getEnvironment().getProperty(EnableAutoConfiguration.ENABLED_OVERRIDE_PROPERTY, Boolean.class, true);
    }
 return true;
}
```

5.4.2 getAttributes

`#getAttributes(AnnotationMetadata metadata)` 方法，获得注解的属性。代码如下：

```
// AutoConfigurationImportSelector.java

protected AnnotationAttributes getAttributes(AnnotationMetadata metadata) {
    String name = getAnnotationClass().getName();
 // 获得注解的属性
    AnnotationAttributes attributes = AnnotationAttributes.fromMap(metadata.getAnnotationAttributes(name, true));
```

```
 // 断言
    Assert.notNull(attributes,
            () -> "No auto-configuration attributes found. Is "
                    + metadata.getClassName() + " annotated with "
                    + ClassUtils.getShortName(name) + "?");
 return attributes;
}
```

注意，此处 `getAnnotationClass().getName()` 返回的是 `@EnableAutoConfiguration` 注解，所以这里返回的注解属性，只能是 `exclude` 和 `excludeName` 这两个。

举个例子，假设 Spring 应用上的注解如下：

```
@SpringBootApplication(exclude = {SpringApplicationAdminJmxAutoConfiguration.class},
    scanBasePackages = "cn.iocoder")
```

- 返回的结果，如下图：

5.4.3 getExclusions

`#getExclusions(AnnotationMetadata metadata, AnnotationAttributes attributes)` 方法，获得需要排除的配置类。代码如下：

```
// AutoConfigurationImportSelector.java

protected Set<String> getExclusions(AnnotationMetadata metadata, AnnotationAttributes attributes) {
    Set<String> excluded = new LinkedHashSet<>();
 // 注解上的 exclude 属性
    excluded.addAll(asList(attributes, "exclude"));
 // 注解上的 excludeName 属性
    excluded.addAll(Arrays.asList(attributes.getStringArray("excludeName")));
 // 配置文件的 spring.autoconfigure.exclude 属性
    excluded.addAll(getExcludeAutoConfigurationsProperty());
```

```
 return excluded;
}
```

一共有三种方式，配置排除属性。

该方法会调用如下的方法，比较简单，胖友自己瞅瞅。

```
// AutoConfigurationImportSelector.java

private List<String> getExcludeAutoConfigurationsProperty() {
    // 一般来说，会走这块的逻辑
 if (getEnvironment() instanceof ConfigurableEnvironment) {
        Binder binder = Binder.get(getEnvironment());
     return binder.bind(PROPERTY_NAME_AUTOCONFIGURE_EXCLUDE, String[].class).map(Arrays::asList).orElse(Collect
    }
    String[] excludes = getEnvironment().getProperty(PROPERTY_NAME_AUTOCONFIGURE_EXCLUDE, String[].class);
 return (excludes != null) ? Arrays.asList(excludes) : Collections.emptyList();
}

protected final List<String> asList(AnnotationAttributes attributes, String name) {
    String[] value = attributes.getStringArray(name);
 return Arrays.asList(value);
}
```

5.4.4 checkExcludedClasses

#checkExcludedClasses(List<String> configurations, Set<String> exclusions) 方法，校验排除的配置类是否合法。代码如下：

```
// AutoConfigurationImportSelector.java

private void checkExcludedClasses(List<String> configurations, Set<String> exclusions) {
    // 获得 exclusions 不在 invalidExcludes 的集合，添加到 invalidExcludes 中
    List<String> invalidExcludes = new ArrayList<>(exclusions.size());
    for (String exclusion : exclusions) {
        if (ClassUtils.isPresent(exclusion, getClass().getClassLoader()) // classpath 存在该类
                && !configurations.contains(exclusion)) { // configurations 不存在该类
            invalidExcludes.add(exclusion);
        }
    }
    // 如果 invalidExcludes 非空，抛出 IllegalStateException 异常
    if (!invalidExcludes.isEmpty()) {
        handleInvalidExcludes(invalidExcludes);
    }
}

/**
 * Handle any invalid excludes that have been specified.
 * @param invalidExcludes the list of invalid excludes (will always have at least one
 * element)
 */
protected void handleInvalidExcludes(List<String> invalidExcludes) {
    StringBuilder message = new StringBuilder();
    for (String exclude : invalidExcludes) {
        message.append("\t- ").append(exclude).append(String.format("%n"));
```

```
        }
        throw new IllegalStateException(String.format("The following classes could not be excluded because they are"
                        + " not auto-configuration classes:%n%s", message));
    }
```

不合法的定义，`exclusions` 存在于 classpath 中，但是不存在 `configurations` 。这样做的目的是，如果不存在的，就不要去排除啦！
代码比较简单，胖友自己瞅瞅即可。

5.4.5 fireAutoConfigurationImportEvents

`#fireAutoConfigurationImportEvents(List<String> configurations, Set<String> exclusions)` 方法，触发自动配置类引入完成的事件。代码如下：

```
// AutoConfigurationImportSelector.java

private void fireAutoConfigurationImportEvents(List<String> configurations, Set<String> exclusions) {
    // <1> 加载指定类型 AutoConfigurationImportListener 对应的，在 `META-INF/spring.factories` 里的类名的数组。
    List<AutoConfigurationImportListener> listeners = getAutoConfigurationImportListeners();
 if (!listeners.isEmpty()) {
        // <2> 创建 AutoConfigurationImportEvent 事件
        AutoConfigurationImportEvent event = new AutoConfigurationImportEvent(this, configurations, exclusions);
     // <3> 遍历 AutoConfigurationImportListener 监听器们，逐个通知
     for (AutoConfigurationImportListener listener : listeners) {
            // <3.1> 设置 AutoConfigurationImportListener 的属性
            invokeAwareMethods(listener);
         // <3.2> 通知
            listener.onAutoConfigurationImportEvent(event);
        }
    }
}
```

<1> 处，调用 `#getAutoConfigurationImportListeners()` 方法，加载指定类型 AutoConfigurationImportListener 对应的，在 `META-INF/spring.factories` 里的类名的数组。例如：

<2> 处，创建 AutoConfigurationImportEvent 事件。

<3> 处，遍历 AutoConfigurationImportListener 监听器们，逐个通知。

- <3.1> 处，调用 `#invokeAwareMethods(Object instance)` 方法，设置 AutoConfigurationImportListener 的属性。代码如下：

```
// AutoConfigurationImportSelector.java

private void invokeAwareMethods(Object instance) {
    // 各种 Aware 属性的注入
```

```
            if (instance instanceof Aware) {
                if (instance instanceof BeanClassLoaderAware) {
                        ((BeanClassLoaderAware) instance).setBeanClassLoader(this.beanClassLoader);
                    }
                if (instance instanceof BeanFactoryAware) {
                        ((BeanFactoryAware) instance).setBeanFactory(this.beanFactory);
                    }
                if (instance instanceof EnvironmentAware) {
                        ((EnvironmentAware) instance).setEnvironment(this.environment);
                    }
                if (instance instanceof ResourceLoaderAware) {
                        ((ResourceLoaderAware) instance).setResourceLoader(this.resourceLoader);
                    }
                }
            }
```

    - 各种 Aware 属性的注入。

- <3.2> 处，调用 AutoConfigurationImportListener#onAutoConfigurationImportEvent(event) 方法，通知监听器。目前只有一个 ConditionEvaluationReportAutoConfigurationImportListener 监听器，没啥逻辑，有兴趣自己看哈。

6. AutoConfigurationPackages

org.springframework.boot.autoconfigure.AutoConfigurationPackages ，自动配置所在的包名。可能这么解释有点怪怪的，我们来看下官方注释：

```
Class for storing auto-configuration packages for reference later (e.g. by JPA entity scanner).
```

简单来说，就是将使用 @AutoConfigurationPackage 注解的类所在的包（package），注册成一个 Spring IoC 容器中的 Bean 。酱紫，后续有其它模块需要使用，就可以通过获得该 Bean ，从而获得所在的包。例如说，JPA 模块，需要使用到。

是不是有点神奇，芮芮也觉得。

6.1 Registrar

Registrar ，是 AutoConfigurationPackages 的内部类，实现 ImportBeanDefinitionRegistrar、DeterminableImports 接口，注册器，用于处理 @AutoConfigurationPackage 注解。代码如下：

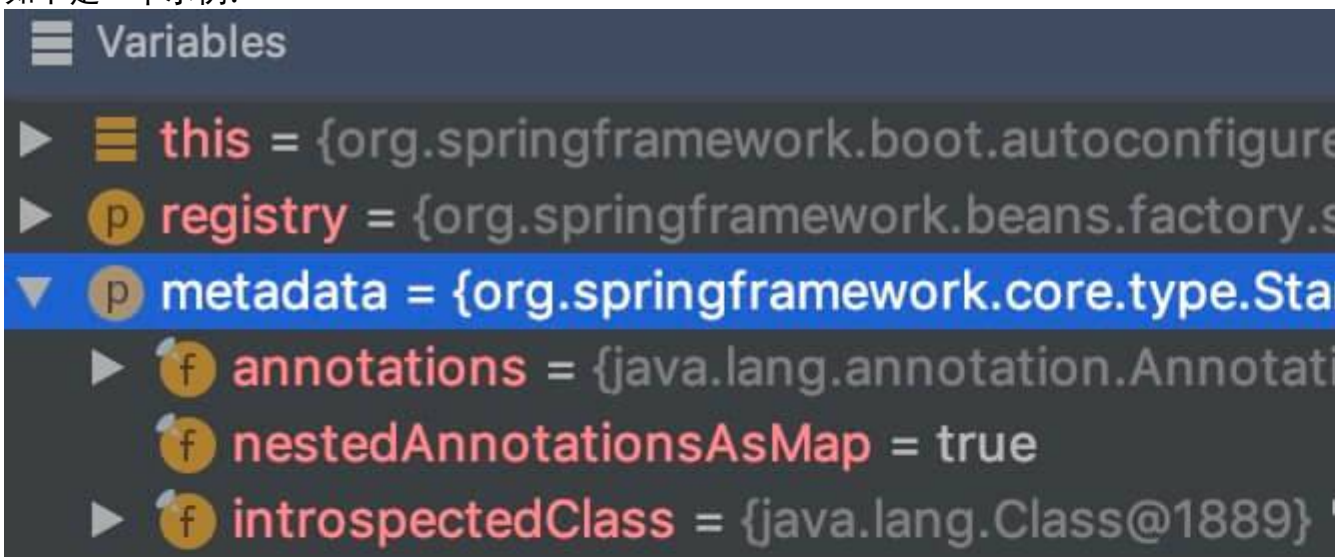
```
// AutoConfigurationPackages#Registrar.java

static class Registrar implements ImportBeanDefinitionRegistrar, DeterminableImports {

 @Override
 public void registerBeanDefinitions(AnnotationMetadata metadata, BeanDefinitionRegistry registry) {
        register(registry, new PackageImport(metadata).getPackageName()); // <X>
    }

 @Override
 public Set<Object> determineImports(AnnotationMetadata metadata) {
```

```
        return Collections.singleton(new PackageImport(metadata));
    }

}
```

PackageImport 是 AutoConfigurationPackages 的内部类，用于获得包名。代码如下：

```
    // AutoConfigurationPackages#Registrar.java

    private static final class PackageImport {

        /**
         * 包名
         */
        private final String packageName;

        PackageImport(AnnotationMetadata metadata) {
            this.packageName = ClassUtils.getPackageName(metadata.getClassName());
        }

        public String getPackageName() {
            return this.packageName;
        }

        @Override
        public boolean equals(Object obj) {
            if (obj == null || getClass() != obj.getClass()) {
                return false;
            }
            return this.packageName.equals(((PackageImport) obj).packageName);
        }

        @Override
        public int hashCode() {
            return this.packageName.hashCode();
        }

        @Override
        public String toString() {
            return "Package Import " + this.packageName;
        }

    }
```

- 如下是一个示例：



<X> 处，调用 #register(BeanDefinitionRegistry registry, String... packageNames) 方法，注册一个用于存储报名（package）的 Bean 到 Spring IoC 容器中。详细解析，见 「6.2 register」 。

6.2 register

#register(BeanDefinitionRegistry registry, String... packageNames) 方法，注册一个用于存储报名（package）的 Bean 到 Spring IoC 容器中。代码如下：

```
// AutoConfigurationPackages.java

private static final String BEAN = AutoConfigurationPackages.class.getName();

public static void register(BeanDefinitionRegistry registry, String... packageNames) {
    // <1> 如果已经存在该 BEAN ，则修改其包（package）属性
 if (registry.containsBeanDefinition(BEAN)) {
        BeanDefinition beanDefinition = registry.getBeanDefinition(BEAN);
        ConstructorArgumentValues constructorArguments = beanDefinition.getConstructorArgumentValues();
        constructorArguments.addIndexedArgumentValue(0, addBasePackages(constructorArguments, packageNames));
    // <2> 如果不存在该 BEAN ，则创建一个 Bean ，并进行注册
    } else { GenericBeanDefinition beanDefinition = new GenericBeanDefinition();
        beanDefinition.setBeanClass(BasePackages.class);
        beanDefinition.getConstructorArgumentValues().addIndexedArgumentValue(0, packageNames);
        beanDefinition.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
        registry.registerBeanDefinition(BEAN, beanDefinition);
    }
}
```

注册的 BEAN 的类型，为 BasePackages 类型。它是 AutoConfigurationPackages 的内部类。代码如下：

```
// AutoConfigurationPackages#BasePackages.java

static final class BasePackages {

 private final List<String> packages;

 private boolean loggedBasePackageInfo;

    BasePackages(String... names) {
        List<String> packages = new ArrayList<>();
     for (String name : names) {
         if (StringUtils.hasText(name)) {
                packages.add(name);
            }
        }
     this.packages = packages;
    }

 public List<String> get() {
     if (!this.loggedBasePackageInfo) {
         if (this.packages.isEmpty()) {
             if (logger.isWarnEnabled()) {
                    logger.warn("@EnableAutoConfiguration was declared on a class "
                            + "in the default package. Automatic @Repository and "
                            + "@Entity scanning is not enabled.");
                }
```

```
                } else {
                 if (logger.isDebugEnabled()) {
                        String packageNames = StringUtils
                                .collectionToCommaDelimitedString(this.packages);
                        logger.debug("@EnableAutoConfiguration was declared on a class "
                                + "in the package '" + packageNames
                                + "'. Automatic @Repository and @Entity scanning is "
                                + "enabled.");
                    }
                }
                this.loggedBasePackageInfo = true;
               }
             return this.packages;
            }

        }
```

- 就是一个有 packages 属性的封装类。

<1> 处，如果已经存在该 BEAN ，则修改其包（package）属性。而合并 package 的逻辑，通过 #addBasePackages(ConstructorArgumentValues constructorArguments, String[] packageNames) 方法，进行实现。代码如下：

```
// AutoConfigurationPackages.java

private static String[] addBasePackages(ConstructorArgumentValues constructorArguments, String[] packageNames) {
 // 获得已存在的
    String[] existing = (String[]) constructorArguments.getIndexedArgumentValue(0, String[].class).getValue();
 // 进行合并
    Set<String> merged = new LinkedHashSet<>();
    merged.addAll(Arrays.asList(existing));
    merged.addAll(Arrays.asList(packageNames));
 return StringUtils.toStringArray(merged);
}
```

<2> 处，如果不存在该 BEAN ，则创建一个 Bean ，并进行注册。

6.3 has

#has(BeanFactory beanFactory) 方法，判断是否存在该 BEAN 在传入的容器中。代码如下：

```
// AutoConfigurationPackages.java

public static boolean has(BeanFactory beanFactory) {
 return beanFactory.containsBean(BEAN) && !get(beanFactory).isEmpty();
}
```

6.4 get

#get(BeanFactory beanFactory) 方法，获得 BEAN 。代码如下：

```
// AutoConfigurationPackages.java
```

```
public static List<String> get(BeanFactory beanFactory) {
 try {
     return beanFactory.getBean(BEAN, BasePackages.class).get();
    } catch (NoSuchBeanDefinitionException ex) {
     throw new IllegalStateException("Unable to retrieve @EnableAutoConfiguration base packages");
    }
}
```

666. 彩蛋

比想象中长的一篇文章。虽然中间有些地方复杂了一点，但是觉得还是蛮有趣的。

撸完有点不清晰的胖友，再调试两遍。还有疑惑，星球留言走一波哟。

参考和推荐如下文章：

> 快乐崇拜 《Spring Boot 源码深入分析》
>
> > 有木发现，芳芳写的比他详细很多很多。
>
> 老田 《Spring Boot 2.0 系列文章(六)：Spring Boot 2.0 中SpringBootApplication注解详解》
>
> dm_vincent 《[Spring Boot] 4. Spring Boot实现自动配置的原理》

文章目录

回到首页