

无

“ 1. 概述在前面的文章，我们已经看过 Spring Boot 如何实现自动配置的功能，但是，实际场景下，这显然不够。

在前面的文章，我们已经看过 Spring Boot 如何实现自动配置的功能，但是，实际场景下，这显然不够。为什么呢？因为每个框架的配置，需要满足一定的条件，才应该进行自动配置。这时候，我们很自然就可以想到 Spring Boot 的 Condition 功能。不过呢，Condition 功能并不是 Spring Boot 所独有，而是在 Spring Framework 中就已经提供了。那么，究竟是什么样的关系呢，我们在 [「2. Condition 演进史」](#) 来瞅瞅。

2.1 Profile 的出场

在 Spring3.1 的版本，为了满足不同环境注册不同的 Bean，引入了 `@Profile` 注解。例如：

```
@Configuration
public class DataSourceConfiguration {

    @Bean
    @Profile("DEV")
    public DataSource devDataSource() {

    }

    @Bean
    @Profile("PROD")
    public DataSource prodDataSource() {

    }

}
```

- 在测试环境下，我们注册单机 MySQL 的 DataSource Bean。
- 在生产环境下，我们注册集群 MySQL 的 DataSource Bean。

`org.springframework.context.annotation.Profile`，代码如下：

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Conditional(ProfileCondition.class)
public @interface Profile {

    String[] value();

}
```

- 这是 Spring5 版本的 `@Profile` 注解的代码。它已经是经过 Condition 改造的实现。详细的，我们放在 [「2.2 Condition」](#)。

- 让我们在来看一眼 Spring3 版本的 `@Profile` 注解的代码。如下：

```
@Retention(RetentionPolicy.RUNTIME)
@Target({
    ANNOTATION_TYPE,
    TYPE
})
public @interface Profile {

    static final String CANDIDATE_PROFILES_ATTRIB_NAME = "value";

    String[] value();

}
```

- ```
String[] value());
}
```
- 可以大体猜出，此时并没有将 Profile 作为 Condition 的一种情况。

## 2.2 Condition 的出现

在 Spring4 的版本，正式出现 Condition 功能，体现在 `org.springframework.context.annotation.Condition` 接口，代码如下：

```
@FunctionalInterface
public interface Condition {

 boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata);

}
```

- 很简洁的一个接口，只有一个 `#matches(...)` 方法，用于判断是否匹配。从参数中就可以看出，它是和注解配合，而这个注解便是 `@Conditional` 。

`org.springframework.context.annotation.Conditional` 注解，也是在 Spring4 的版本，一起出现。代码如下：

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Conditional {

 Class<? extends Condition>[] value();

}
```

- 可以注解在方法、或者在类上，表示需要满足的条件（Condition）。
- 在 [\[2.1 Profile 的出现\]](#) 小节中，我们已经看到 `@Profile` 上，有 `@Conditional(ProfileCondition.class)` 的注解，表示使用 `org.springframework.context.annotation.ProfileCondition` 作为条件。
- 当然，我们也可以直接在 Configuration 类上使用。例如：

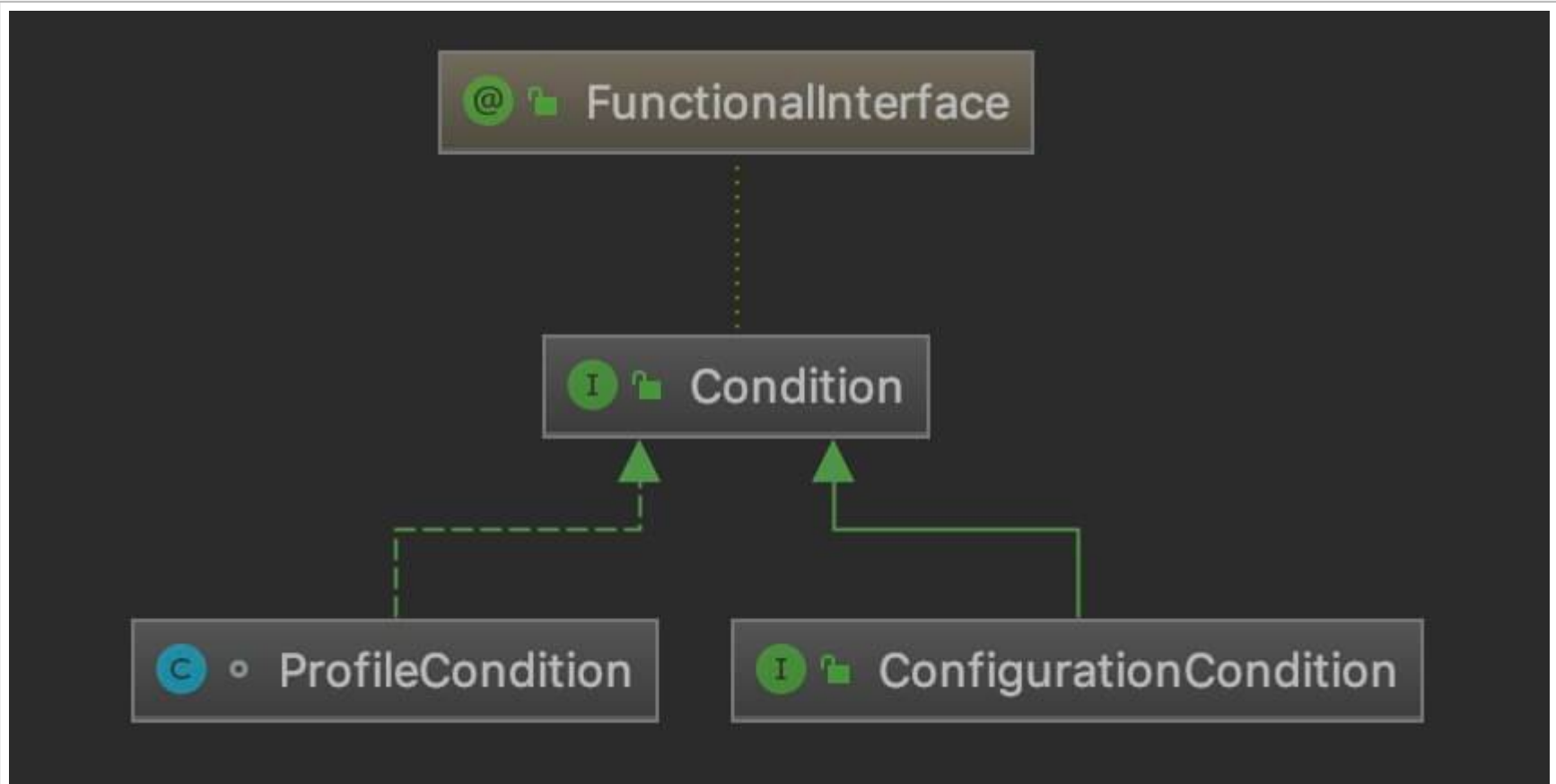
```
@Configuration
public class TestConfiguration {

 @Bean
 @Conditional(XXXCondition.class)
 public Object xxxObject() {
 return new Object();
 }

}
```

- 即，创建 `#xxxObject()` 方法对应的 Bean 对象，需要满足 XXXCondition 条件。

在 Spring5 中，芳芳整理了下目前提供的 Condition 实现类，如下图：

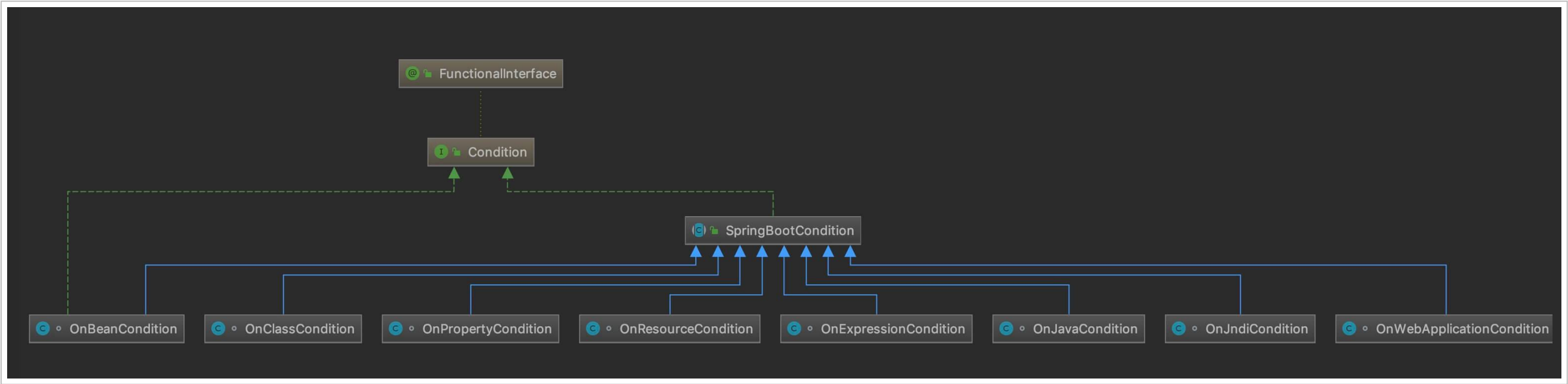


Condition 实现类

- 显然，默认提供的 Condition 实现类非常少。

2.3 SpringBootCondition 的进击

为了满足更加丰富的 Condition（条件）的需要，Spring Boot 进一步拓展了更多的实现类，如下图所示：



Spring Boot Condition 实现类

- org.springframework.boot.autoconfigure.condition.SpringBootCondition ，是 Spring Boot 实现 Condition 的抽象类，且是 Spring Boot 所有 Condition 实现类的基类。
- 分别对应如下注解：
  - @ConditionalOnBean ：当容器里有指定 Bean 的条件下。
  - @ConditionalOnMissingBean ：当容器里没有指定 Bean 的情况下。
  - @ConditionalOnSingleCandidate ：当指定 Bean 在容器中只有一个，或者虽然有多个但是指定首选 Bean 。
  - @ConditionalOnClass ：当类路径下有指定类的条件下。
  - @ConditionalOnMissingClass ：当类路径下没有指定类的条件下。
  - @ConditionalOnProperty ：指定的属性是否有指定的值
  - @ConditionalOnResource ：类路径是否有指定的值
  - @ConditionalOnExpression ：基于 SpEL 表达式作为判断条件。
  - @ConditionalOnJava ：基于 Java 版本作为判断条件
  - @ConditionalOnJndi ：在 JNDI 存在的条件下差在指定的位置
  - @ConditionalOnNotWebApplication ：当前项目不是 Web 项目的条件下
  - @ConditionalOnWebApplication ：当前项目是 Web 项目的条件下。

2.4 小结

到了此处，我们基本算是理清了 Condition 的整个演进构成：

- @Profile 注解，在 Spring3.1 提出，可以作为是 Condition 的雏形。
- Condition 接口，在 Spring4 提出，是 Condition 的正式出现。
- SpringCondition 抽象类，在 Spring Boot 实现，是对 Condition 进一步拓展。

下面，我们就正式开始撸 Condition 相关的源码落。

在上面的文章中，我们已经看到， `@Conditional` 注解，可以添加：

- 类级别上
- 方法级别上

添加到注解上，相当于添加到类级别或者方法级别上。

并且，一般情况下我们和配置类（Configuration）一起使用，但是实际上，我们也可以添加到普通的 Bean 类上。例如：

```
@Controller
@RequestMapping("/demo")
@Conditional(TestCondition.class)
public class DemoController {

 @ResponseBody
 @RequestMapping("/hello")
 public String hello() {
 return "world";
 }
}
```

那么，究竟 Condition 是如何生效的呢？分成两种情况：

- 方式一，配置类。添加到配置类（Configuration）上面。
- 方式二，创建 Bean 对象。添加到配置类（Configuration）、或者 Bean Class 的上面。

本质上，方式二上的两种，都是创建 Bean 对象，所以统一处理方式即可。

假设，我们在 TestConfiguration 这个示例下进行测试，看看具体的调用链。代码如下：

```
@Configuration
@Conditional(TestCondition.class)
public class TestConfiguration {

 @Bean
 @Conditional(TestCondition.class)
 public Object testObject() {
 return new Object();
 }
}

public class TestCondition implements Condition {

 @Override
 public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
 return true;
 }
}
```

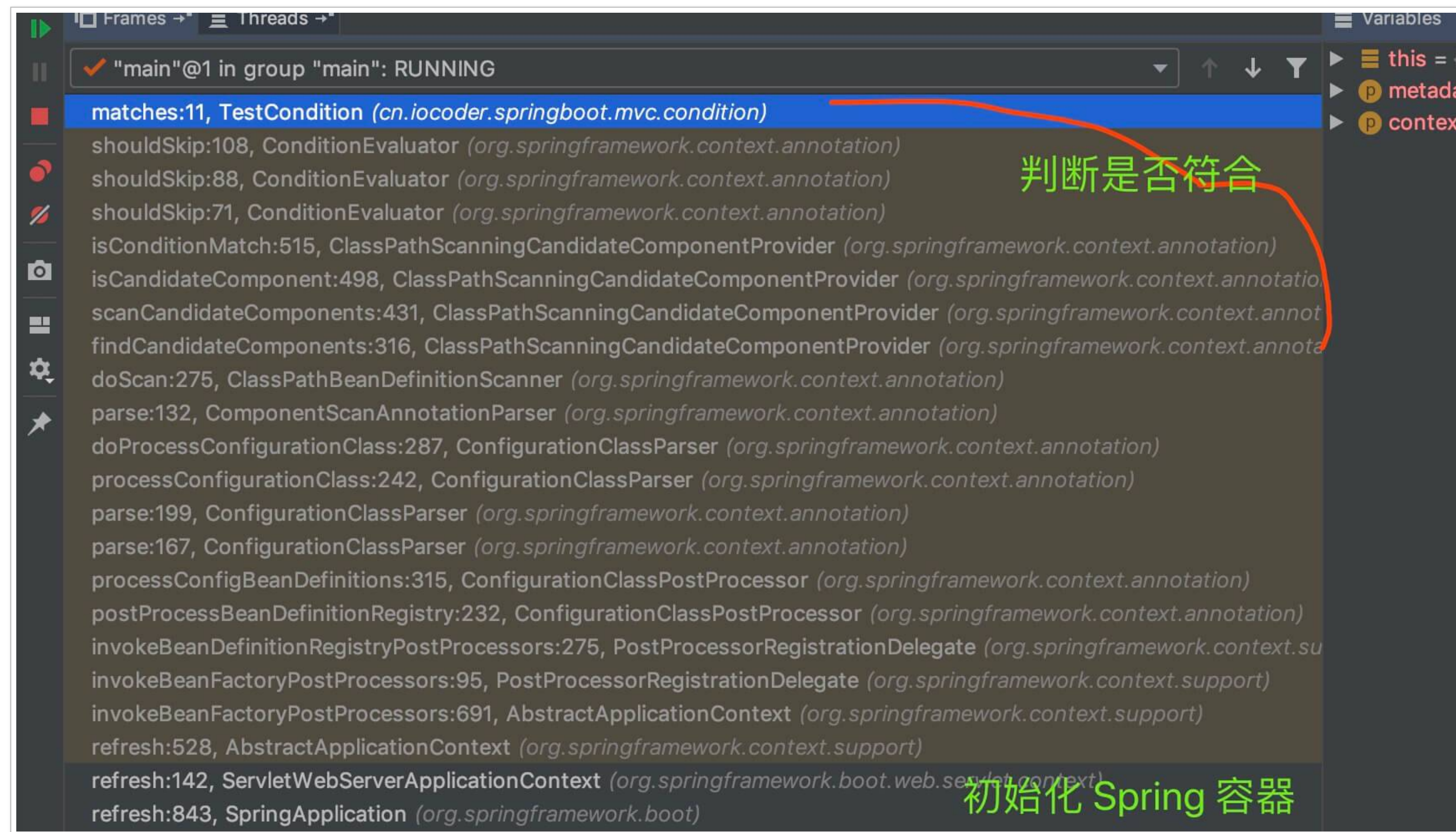
本小节，不会讲特别细的源码。

### 3.1 方式一：配置类

在 TestCondition 的 `matches()` 方法中，打个断点，看看方式一情况下的具体的表现，如下图所示：



在 TestCondition 的 #matches(...) 方法中，打个断点。看看方式一情况下的具体的表现。如下图所示：



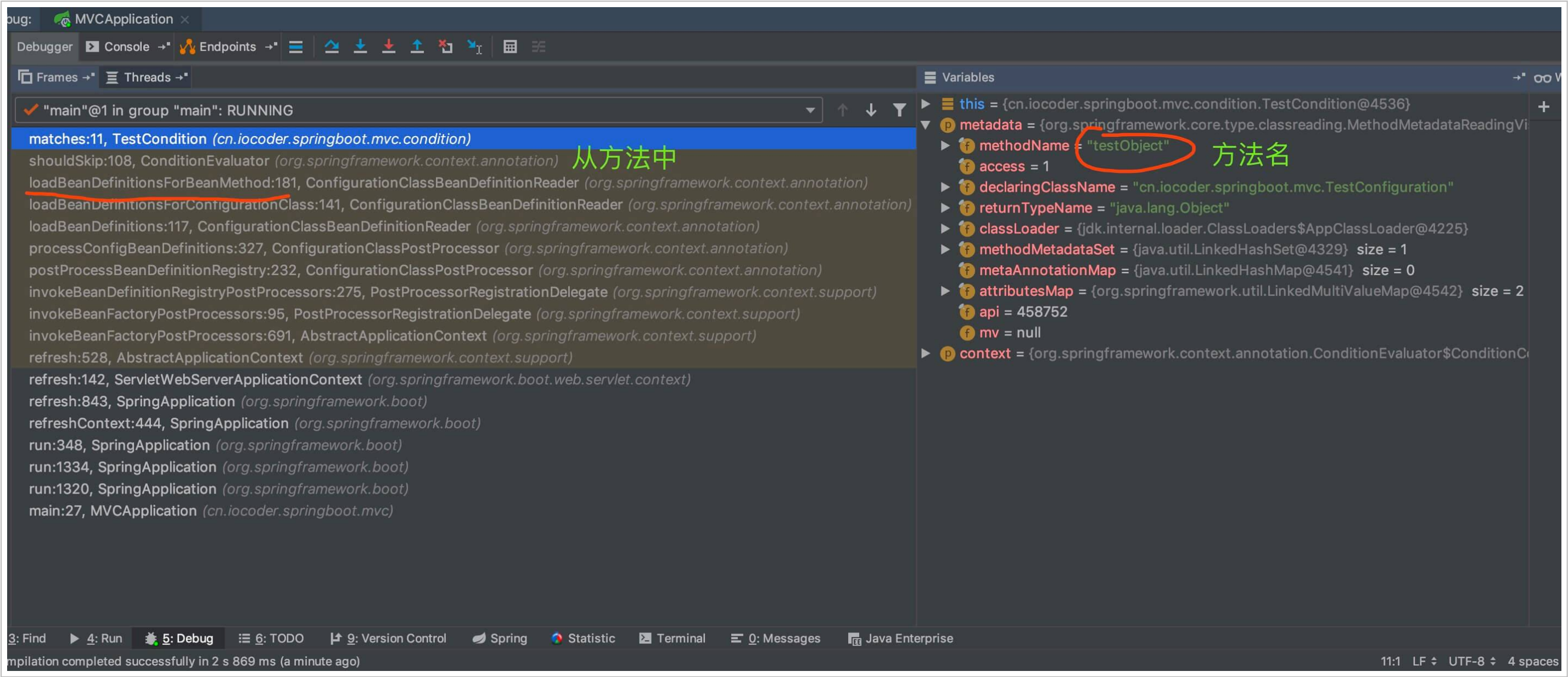
调用栈

- 通过调用 Condition#matches(...) 方法，判断该是否匹配。如果不匹配，内部所有方法，都无法创建 Bean 对象。

## 3.2 方式二：创建 Bean 对象

在 TestCondition 的 #matches(...) 方法中，打个断点。看看方式二情况下的具体的表现。如下图所示：





调用栈

- 通过调用 `Condition#matches(...)` 方法，判断是否匹配。如果吧匹配，则不从该方法加载 `BeanDefinition` 。这样，就不会创建对应的 `Bean` 对象了。

### 3.3 小结

至此，我们已经看到 `Condition` 如何生效。还是相对比较简单。

下面，我们一起来看看 `SpringBootCondition` 如何实现它的进击。

芳芳：先插播下 `ProfileCondition` 的实现代码。

`org.springframework.context.annotation.ProfileCondition` ，实现 `Condition` 接口，给 `@Profile` 使用的 `Condition` 实现类。代码如下：

```
class ProfileCondition implements Condition {

 @Override
 public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {

 MultiValueMap<String, Object> attrs = metadata.getAllAnnotationAttributes(Profile.class.getName());

 if (attrs != null) {

 for (Object value : attrs.get("value")) {
```

```
 if (context.getEnvironment().acceptsProfiles(Profiles.of((String[]) value))) {
 return true;
 }

 return false;
 }

 return true;
}

}
```

- 核心逻辑，获得 @Profile 的 value 属性，和 environment 是否有匹配的。如果有，则表示匹配。

org.springframework.boot.autoconfigure.condition.SpringBootCondition ，实现 Condition 接口，Spring Boot Condition 的抽象基类，主要用于提供相应的日志，帮助开发者判断哪些被进行加载。如下是其上的类注释：

## 5.1 matches

实现 #matches(ConditionContext context, AnnotatedTypeMetadata metadata) 方法，实现匹配逻辑。代码如下：

```
@Override
public final boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {

 String classOrMethodName = getClassOrMethodName(metadata);
 try {

 ConditionOutcome outcome = getMatchOutcome(context, metadata);

 logOutcome(classOrMethodName, outcome);

 recordEvaluation(context, classOrMethodName, outcome);

 return outcome.isMatch();
 } catch (NoClassDefFoundError ex) {
 throw new IllegalStateException(
 "Could not evaluate condition on " + classOrMethodName + " due to "
 + ex.getMessage() + " not "
 + "found. Make sure your own configuration does not rely on "
 + "that class. This can also happen if you are "
 + "@ComponentScanning a springframework package (e.g. if you "
 + "put a @ComponentScan in the default package by mistake)",
 ex);
 } catch (RuntimeException ex) {
 throw new IllegalStateException("Error processing condition on " + getName(metadata), ex);
 }
}
```

- <1> 处，调用 #getClassOrMethodName(AnnotatedTypeMetadata metadata) 方法，获得注解的是方法名还是类名。代码如下：

```
private static String getClassOrMethodName(AnnotatedTypeMetadata metadata) {

 if (metadata instanceof ClassMetadata) {
 ClassMetadata classMetadata = (ClassMetadata) metadata;
 return classMetadata.getClassName();
 }

 MethodMetadata methodMetadata = (MethodMetadata) metadata;
 return methodMetadata.getDeclaringClassName() + "#" + methodMetadata.getMethodName();
}
```

- <2> 处，调用 #getMatchOutcome(ConditionContext context, AnnotatedTypeMetadata metadata) 抽象方法，执行匹配，返回匹配结果。这是一个抽象方法，由子类进行实现。
  - org.springframework.boot.autoconfigure.condition.ConditionOutcome ，匹配结果。
  - org.springframework.boot.autoconfigure.condition.ConditionMessage ，匹配消息。
  - 以上的类，自己瞅瞅。简单~

- <3> 处，调用 #logOutcome(String classOrMethodName, ConditionOutcome outcome) 方法，打印结果日志。代码如下：

```
protected final void logOutcome(String classOrMethodName, ConditionOutcome outcome) {
```

```
protected final void log(String classOrMethodName, ConditionOutcome outcome) {
 if (this.logger.isTraceEnabled()) {
 this.logger.trace(getLogMessage(classOrMethodName, outcome));
 }
}

private StringBuilder getLogMessage(String classOrMethodName, ConditionOutcome outcome) {
 StringBuilder message = new StringBuilder();
 message.append("Condition ");
 message.append(ClassUtils.getShortName(getClass()));
 message.append(" on ");
 message.append(classOrMethodName);
 message.append(outcome.isMatch() ? " matched" : " did not match");
 if (StringUtils.hasLength(outcome.getMessage())) {
 message.append(" due to ");
 message.append(outcome.getMessage());
 }
 return message;
}
```

- <4> 处，调用 #recordEvaluation(ConditionContext context, String classOrMethodName, ConditionOutcome outcome) 方法，记录到 ConditionEvaluationReport 。代码如下：

```
private void recordEvaluation(ConditionContext context, String classOrMethodName, ConditionOutcome outcome) {
 if (context.getBeanFactory() != null) {
 ConditionEvaluationReport.get(context.getBeanFactory()).recordConditionEvaluation(classOrMethodName, this, outcome);
 }
}
```

- 关于 org.springframework.boot.autoconfigure.condition.ConditionEvaluationReport 类，先不详细看，避免研究过深。

- <5> 处，返回是否匹配。

## 5.2 anyMatches

#anyMatches(ConditionContext context, AnnotatedTypeMetadata metadata, Condition... conditions) 方法，判断是否匹配指定的 Condition 们中的任——一个。代码如下：

芬芳：总感觉这个方法，应该是个静态方法才合适。所以，胖友即酱油看看即可。

```
protected final boolean anyMatches(ConditionContext context, AnnotatedTypeMetadata metadata, Condition... conditions) {

 for (Condition condition : conditions) {

 if (matches(context, metadata, condition)) {
 return true;
 }
 }
 return false;
}
```

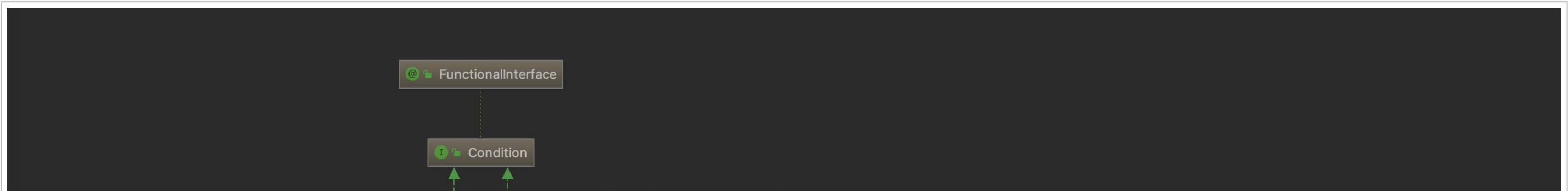
- 遍历 conditions 数组，调用 #matches(ConditionContext context, AnnotatedTypeMetadata metadata, Condition condition) 方法，执行匹配。代码如下：

```
protected final boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata, Condition condition) {

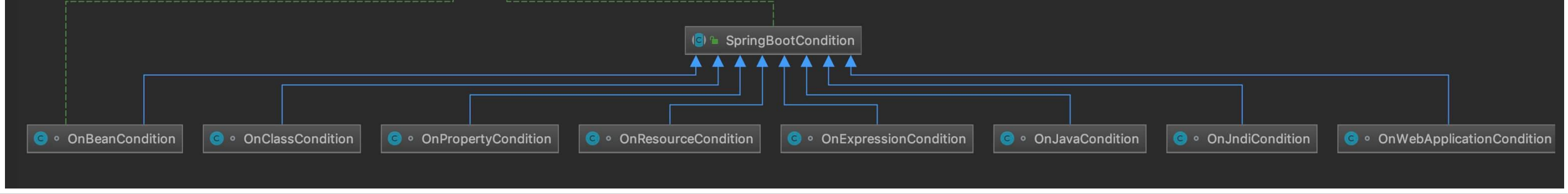
 if (condition instanceof SpringBootCondition) {
 return ((SpringBootCondition) condition).getMatchOutcome(context, metadata).isMatch();
 }
 return condition.matches(context, metadata);
}
```

总的来说，SpringBootCondition 这个类，没啥好说，重点还是在子类。

我们在回忆下，SpringBootCondition 的实现类，主要如下图：







### Spring Boot Condition 实现类

显然，我们不会去看每一个类的 SpringBootCondition 的实现类。所以呢，芴芴也不会每个类都写。

旁白君：偷懒都偷的如此猥琐，哈哈哈哈。

## 6.1 OnPropertyCondition

芴芴：来来来，先看一个容易的（捏个软柿子）。

`org.springframework.boot.autoconfigure.condition.OnPropertyCondition`，继承 `SpringBootCondition` 抽象类，给 `@ConditionalOnProperty` 使用的 Condition 实现类。

如果胖友不熟悉 `@ConditionalOnProperty` 注解，赶紧打开 [《@ConditionalOnProperty 来控制 Configuration 是否生效》](#) 学习 3 分钟~ 不能再多了。

### 6.1.1 getMatchOutcome

`#getMatchOutcome(ConditionContext context, AnnotatedTypeMetadata metadata)` 方法，获得匹配结果。代码如下：

```
@Override
public ConditionOutcome getMatchOutcome(ConditionContext context, AnnotatedTypeMetadata metadata) {

 List<AnnotationAttributes> allAnnotationAttributes = annotationAttributesFromMultiValueMap(
 metadata.getAllAnnotationAttributes(ConditionalOnProperty.class.getName()));

 List<ConditionMessage> noMatch = new ArrayList<>();
 List<ConditionMessage> match = new ArrayList<>();

 for (AnnotationAttributes annotationAttributes : allAnnotationAttributes) {
 ConditionOutcome outcome = determineOutcome(annotationAttributes, context.getEnvironment());
 (outcome.isMatch() ? match : noMatch).add(outcome.getConditionMessage());
 }

 if (!noMatch.isEmpty()) {
 return ConditionOutcome.noMatch(ConditionMessage.of(noMatch));
 }

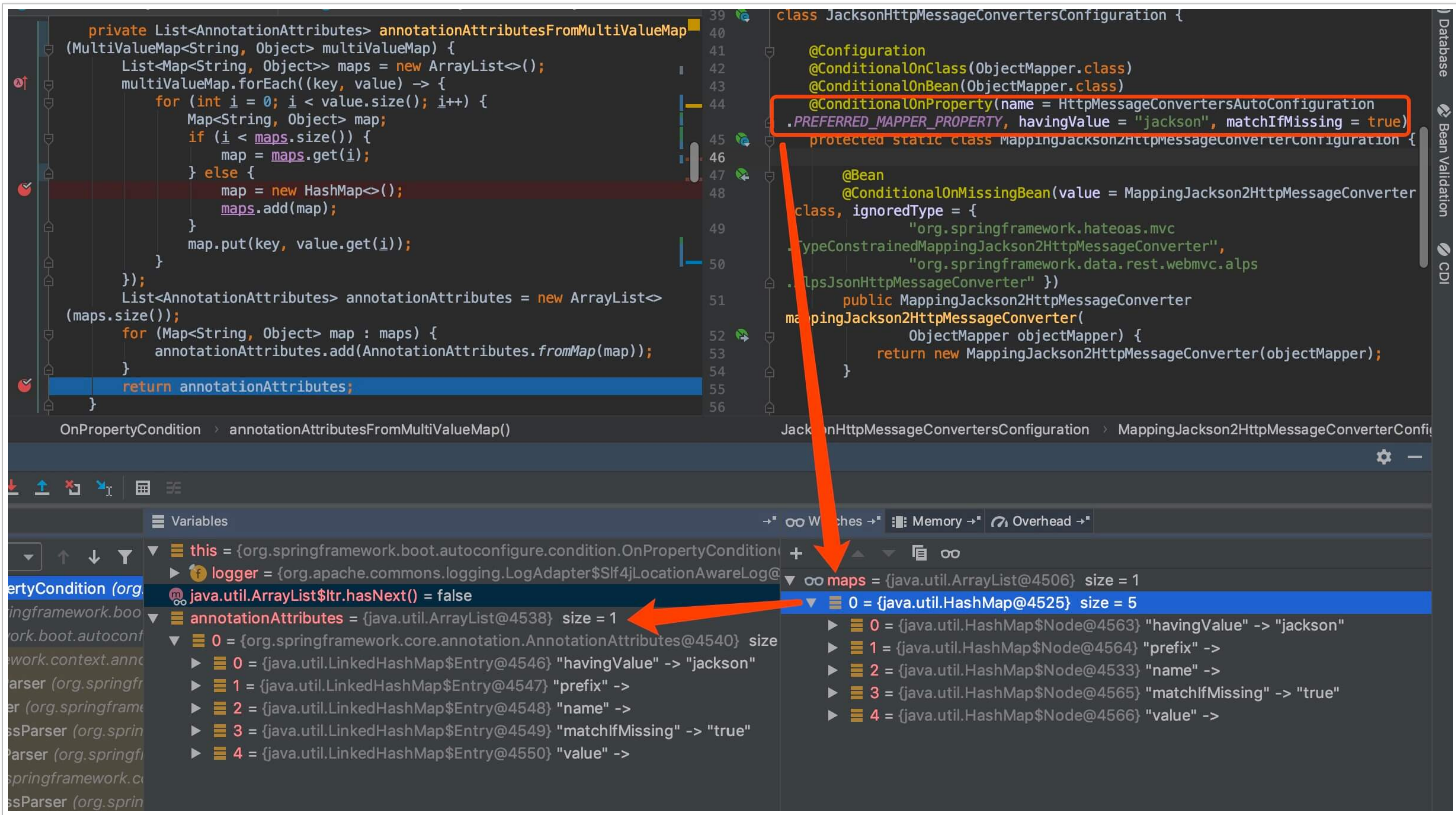
 return ConditionOutcome.match(ConditionMessage.of(match));
}
```

- <1> 处，调用 `#annotationAttributesFromMultiValueMap(MultiValueMap<String, Object> multiValueMap)` 方法，获得 `@ConditionalOnProperty` 注解的属性。代码如下：

```
private List<AnnotationAttributes> annotationAttributesFromMultiValueMap(MultiValueMap<String, Object> multiValueMap) {
 List<Map<String, Object>> maps = new ArrayList<>();
 multiValueMap.forEach((key, value) -> {
 for (int i = 0; i < value.size(); i++) {
 Map<String, Object> map;
 if (i < maps.size()) {
 map = maps.get(i);
 } else {
 map = new HashMap<>();
 maps.add(map);
 }
 map.put(key, value.get(i));
 }
 });
 List<AnnotationAttributes> annotationAttributes = new ArrayList<>(maps.size());
 for (Map<String, Object> map : maps) {
 annotationAttributes.add(AnnotationAttributes.fromMap(map));
 }
}
```

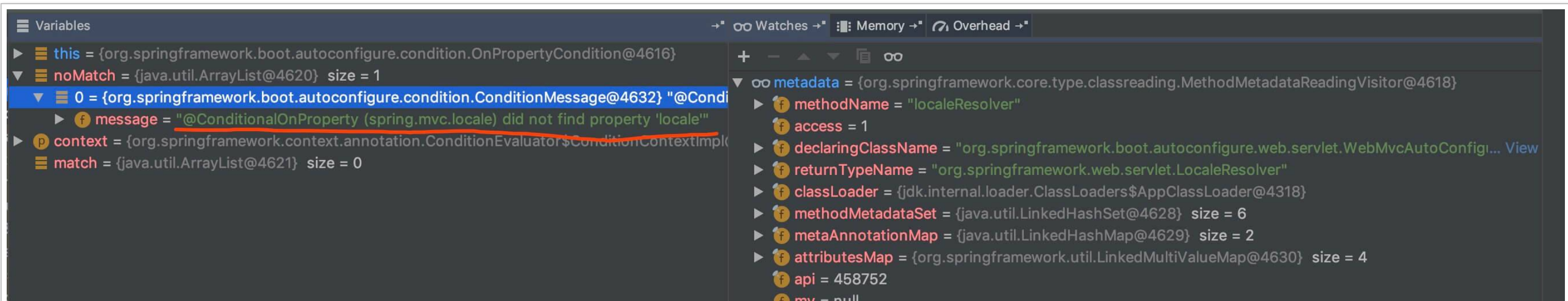


- ```
    }
    annotationAttributes.add(AnnotationAttributes.fromMap(map));
}
return annotationAttributes;
}
```
- 懒的看整个代码实现的过程，可以直接看最终执行的结果图：



`annotationAttributes`

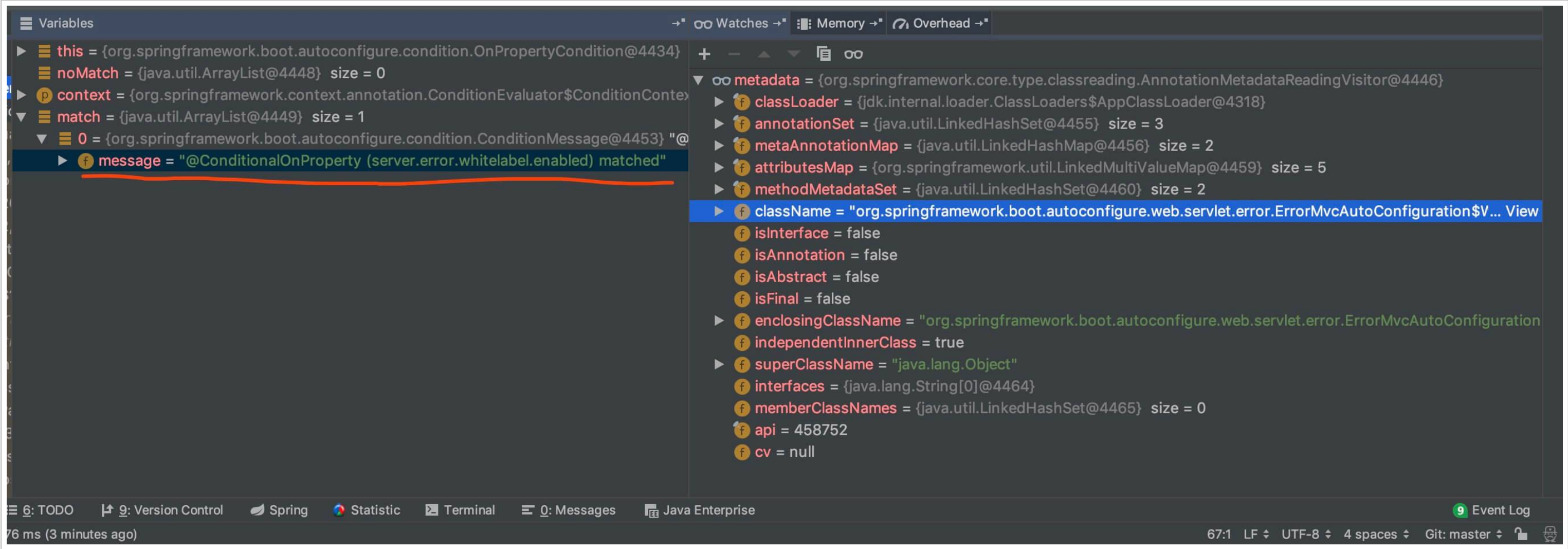
- <2> 处，存储匹配和不匹配的结果消息结果。
- <3> 处，遍历 annotationAttributes 属性数组，逐个调用 #determineOutcome(AnnotationAttributes annotationAttributes, PropertyResolver resolver) 方法，判断是否匹配，并添加到结果。详细解析，见「6.1.2 determineOutcome」。
- <4.1> 处，如果有不匹配的，则返回不匹配。返回结果示例如下：





不匹配

- <4.2> 处，如果都匹配，则返回匹配。返回结果示例如下：



匹配

6.1.2 determineOutcome

#determineOutcome(AnnotationAttributes annotationAttributes, PropertyResolver resolver) 方法，判断是否匹配。代码如下：

```
private ConditionOutcome determineOutcome(AnnotationAttributes annotationAttributes, PropertyResolver resolver) {  
  
    Spec spec = new Spec(annotationAttributes);  
  
    List<String> missingProperties = new ArrayList<>();  
    List<String> nonMatchingProperties = new ArrayList<>();  
  
    spec.collectProperties(resolver, missingProperties, nonMatchingProperties);  
  
    if (!missingProperties.isEmpty()) {  
        return ConditionOutcome.noMatch(ConditionMessage.forCondition(ConditionalOnProperty.class, spec)  
            .didNotFind("property", "properties").items(Style.QUOTE, missingProperties));  
    }  
  
    if (!nonMatchingProperties.isEmpty()) {  
        return ConditionOutcome.noMatch(ConditionMessage.forCondition(ConditionalOnProperty.class, spec)  
            .found("different value in property", "different value in properties").items(Style.QUOTE, nonMatchingProperties));  
    }  
  
    return ConditionOutcome.match(ConditionMessage.forCondition(ConditionalOnProperty.class, spec).because("matched"));  
}
```

- <1> 处，解析成 Spec 对象。Spec 是 OnPropertyCondition 的内部静态类。代码如下：

```
private static class Spec {
```



```
private final String prefix;

private final String havingValue;

private final String[] names;

private final boolean matchIfMissing;

Spec(AnnotationAttributes annotationAttributes) {
    String prefix = annotationAttributes.getString("prefix").trim();
    if (StringUtils.hasText(prefix) && !prefix.endsWith(".")) {
        prefix = prefix + ".";
    }
    this.prefix = prefix;
    this.havingValue = annotationAttributes.getString("havingValue");
    this.names = getNames(annotationAttributes);
    this.matchIfMissing = annotationAttributes.getBoolean("matchIfMissing");
}

private String[] getNames(Map<String, Object> annotationAttributes) {
    String[] value = (String[]) annotationAttributes.get("value");
    String[] name = (String[]) annotationAttributes.get("name");
    Assert.state(value.length > 0 || name.length > 0, "The name or value attribute of @ConditionalOnProperty must be specified");
    Assert.state(value.length == 0 || name.length == 0, "The name and value attributes of @ConditionalOnProperty are exclusive");
    return (value.length > 0) ? value : name;
}

}
```

- <2> 处，创建结果数组。
- <3> 处，调用 `Spec#collectProperties(PropertyResolver resolver, List<String> missing, List<String> nonMatching)` 方法，收集是否不匹配的信息，到 `missingProperties` 、 `nonMatchingProperties` 中。代码如下：

```
private void collectProperties(PropertyResolver resolver, List<String> missing, List<String> nonMatching) {

    for (String name : this.names) {

        String key = this.prefix + name;

        if (resolver.containsProperty(key)) {

            if (!isMatch(resolver.getProperty(key), this.havingValue)) {
                nonMatching.add(name);
            }

        } else {

            if (!this.matchIfMissing) {
                missing.add(name);
            }

        }

    }

}

private boolean isMatch(String value, String requiredValue) {

    if (StringUtils.hasLength(requiredValue)) {
        return requiredValue.equalsIgnoreCase(value);
    }

    return !"false".equalsIgnoreCase(value);

}
```

- 匹配的逻辑，胖友自己瞅瞅。可能比较绕的逻辑是， `matchIfMissing` 那块，也就看两眼就明白。
- <41> 处，如果有属性缺失，则返回不匹配。

- `<4.2>` 处，如果有属性不匹配，则返回不匹配。
- `<4.3>` 处，返回匹配。

6.2 其它实现类

SpringBootCondition 的其它实现类，胖友可以自己去瞅瞅啦。当然，有部分实现类，我们会在 [「8. FilteringSpringBootCondition」](#) 分享。

在 [《精尽 Spring Boot 源码分析 —— 自动配置》](#) 一文中，我们埋了一个 `AutoConfigurationImportSelector#filter(List<String> configurations, AutoConfigurationMetadata autoConfigurationMetadata)` 方法的坑，没有进行详细解析。所以呢，这一节我们将填掉这个坑。

`org.springframework.boot.autoconfigure.AutoConfigurationImportFilter` 接口，用于过滤掉无需自动引入的自动配置类（Configuration）。正如其类上的注释：

- 重点是 “fast removal of auto-configuration classes before their bytecode is even read” 。因为自动配置类可能会很多，如果无需使用，而将字节码读取到内存中，这个是一种浪费。

AutoConfigurationImportFilter 的代码如下：

```
@FunctionalInterface
public interface AutoConfigurationImportFilter {

    // ...

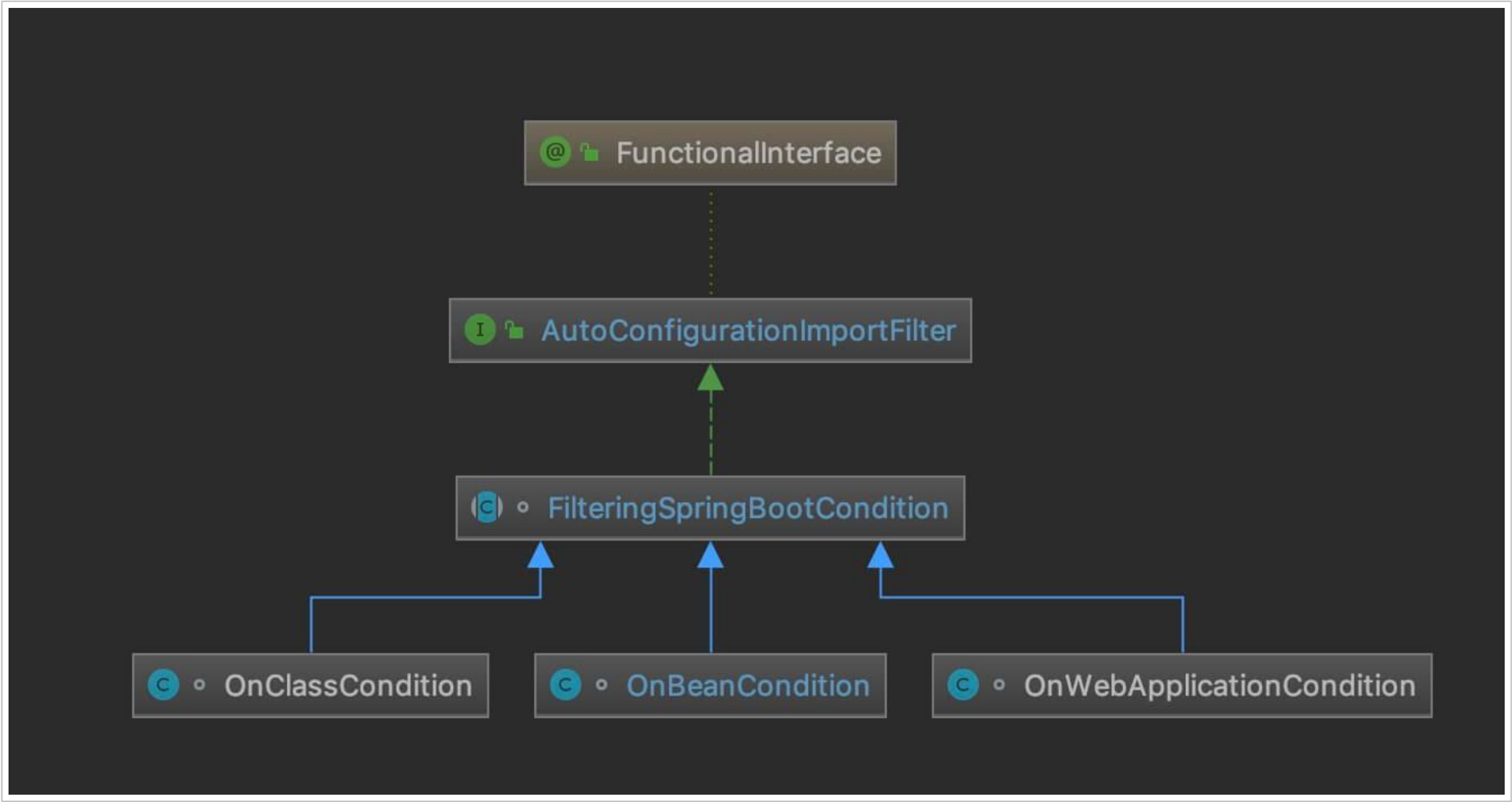
    boolean[] match(String[] autoConfigurationClasses, AutoConfigurationMetadata autoConfigurationMetadata);

}
```

- 将传入的 `autoConfigurationClasses` 配置类们，根据 `autoConfigurationMetadata` 的元数据（主要是注解信息），进行匹配，判断是否需要引入，然后返回的 `boolean[]` 结果。
- 并且，`boolean[]` 结果和 `autoConfigurationClasses` 配置类们是——对应的关系噢。假设 `autoConfigurationClasses[0]` 对应的 `boolean[0]` 为 `false`，表示无需引入，反之则需要引入。

7.1 AutoConfigurationImportFilter 类图

AutoConfigurationImportFilter 的子类如下图所示：



AutoConfigurationImportFilter 类图

- 从图中，我们很容易就看出，AutoConfigurationImportFilter 的最终实现类，都是构建在 SpringBootCondition 之上。🐼 不过这也很正常，因为 Condition 本身提供的一个功能，就是作为配置类（Configuration）是否能够符合条件被引入。

7.2 FilteringSpringBootCondition

`org.springframework.boot.autoconfigure.condition.FilteringSpringBootCondition`，继承 `SpringBootCondition` 抽象类，实现 `AutoConfigurationImportFilter`、`BeanFactoryAware`、`BeanClassLoaderAware` 接口，作为具有 `AutoConfigurationImportFilter` 功能的 `SpringBootCondition` 的抽象基类。

注意，上面特意加黑的“具有 `AutoConfigurationImportFilter` 功能”。

`FilteringSpringBootCondition` 的基本属性，如下：

```
private BeanFactory beanFactory;  
private ClassLoader beanClassLoader;
```

- 通过 Spring Aware 机制，进行注入。

7.2.1 match

实现 `#match(String[] autoConfigurationClasses, AutoConfigurationMetadata autoConfigurationMetadata)` 方法，执行批量的匹配，并返回匹配结果。代码如下：

```
@Override
public boolean[] match(String[] autoConfigurationClasses, AutoConfigurationMetadata autoConfigurationMetadata) {

    ConditionEvaluationReport report = ConditionEvaluationReport.find(this.beanFactory);

    ConditionOutcome[] outcomes = getOutcomes(autoConfigurationClasses, autoConfigurationMetadata);

    boolean[] match = new boolean[outcomes.length];

    for (int i = 0; i < outcomes.length; i++) {

        match[i] = (outcomes[i] == null || outcomes[i].isMatch());

        if (!match[i] && outcomes[i] != null) {

            logOutcome(autoConfigurationClasses[i], outcomes[i]);

            if (report != null) {
                report.recordConditionEvaluation(autoConfigurationClasses[i], this, outcomes[i]);
            }
        }
    }

    return match;
}
```

- 从实现上，这个方法和 `SpringBootCondition#match(ConditionContext context, AnnotatedTypeMetadata metadata)` 方法，基本是一致的。或者说，是它的批量版本。
- `<1>` 处，获得 `ConditionEvaluationReport` 对象。
- `<2>` 处，调用 `#getOutcomes(String[] autoConfigurationClasses, AutoConfigurationMetadata autoConfigurationMetadata)` 抽象方法，执行批量的匹配，并返回匹配结果。这是一个抽象方法，由子类进行实现。
- `<3.1>` 处，创建 `match` 数组。
- `<3.2>` 处，遍历 `outcomes` 结果数组。
 - `<3.2.1>` 处，赋值 `match` 数组的当前元素。
 - `<3.2.2>` 处，如果不匹配，打印日志和记录。其中，`#logOutcome(...)` 方法，就是调用父类 `SpringBootCondition` 的方法。
- `<3.3>` 处，返回 `match` 数组。

7.2.2 ClassNameFilter

`ClassNameFilter`，是 `FilteringSpringBootCondition` 的内部类，提供判断类是否存在的功能。代码如下：

```
protected enum ClassNameFilter {

    PRESENT {

        @Override
        public boolean matches(String className, ClassLoader classLoader) {
            return isPresent(className, classLoader);
        }

    },

    MISSING {

        @Override
        public boolean matches(String className, ClassLoader classLoader) {
            return !isPresent(className, classLoader);
        }

    }

}
```

```
};

public abstract boolean matches(String className, ClassLoader classLoader);

public static boolean isPresent(String className, ClassLoader classLoader) {
    if (classLoader == null) {
        classLoader = ClassUtils.getDefaultClassLoader();
    }
    try {
        forName(className, classLoader);
        return true;
    } catch (Throwable ex) {
        return false;
    }
}

private static Class<?> forName(String className, ClassLoader classLoader) throws ClassNotFoundException {
    if (classLoader != null) {
        return classLoader.loadClass(className);
    }
    return Class.forName(className);
}
}
```

- 里面提供了两个实现类， 且是单例。

- 代码比较简单， 胖友 5 秒看懂。

7.2.3 filter

该方法，提供给子类使用。

#filter(Collection<String> classNames, ClassNameFilter classNameFilter, ClassLoader classLoader) 方法，通过使用 ClassNameFilter 类，过滤出符合条件的类名的数组。代码如下：

```
protected List<String> filter(Collection<String> classNames, ClassNameFilter classNameFilter, ClassLoader classLoader) {

    if (CollectionUtils.isEmpty(classNames)) {
        return Collections.emptyList();
    }

    List<String> matches = new ArrayList<>(classNames.size());

    for (String candidate : classNames) {
        if (classNameFilter.matches(candidate, classLoader)) {
            matches.add(candidate);
        }
    }

    return matches;
}
```

7.2.3 AutoConfigurationImportFilter 的使用

在 AutoConfigurationImportSelector#filter(List<String> configurations, AutoConfigurationMetadata autoConfigurationMetadata) 方法中，我们可以看到 AutoConfigurationImportFilter 的使用，过滤可以忽略的配置类。代码如下：

```
private List<String> filter(List<String> configurations, AutoConfigurationMetadata autoConfigurationMetadata) {

    long startTime = System.nanoTime();
    String[] candidates = StringUtils.toStringArray(configurations);
    boolean[] skip = new boolean[candidates.length];
    boolean skipped = false;
```

```
boolean skipped = false;

for (AutoConfigurationImportFilter filter : getAutoConfigurationImportFilters()) {

    invokeAwareMethods(filter);

    boolean[] match = filter.match(candidates, autoConfigurationMetadata);

    for (int i = 0; i < match.length; i++) {
        if (!match[i]) {
            skip[i] = true;
            candidates[i] = null;

            skipped = true;
        }
    }

}

if (!skipped) {
    return configurations;
}

List<String> result = new ArrayList<>(candidates.length);
for (int i = 0; i < candidates.length; i++) {
    if (!skip[i]) {
        result.add(candidates[i]);
    }
}

if (logger.isTraceEnabled()) {
    int numberFiltered = configurations.size() - result.size();
    logger.trace("Filtered " + numberFiltered + " auto configuration class in "
        + TimeUnit.NANOSECONDS.toMillis(System.nanoTime() - startTime)
        + " ms");
}

return new ArrayList<>(result);
}
```

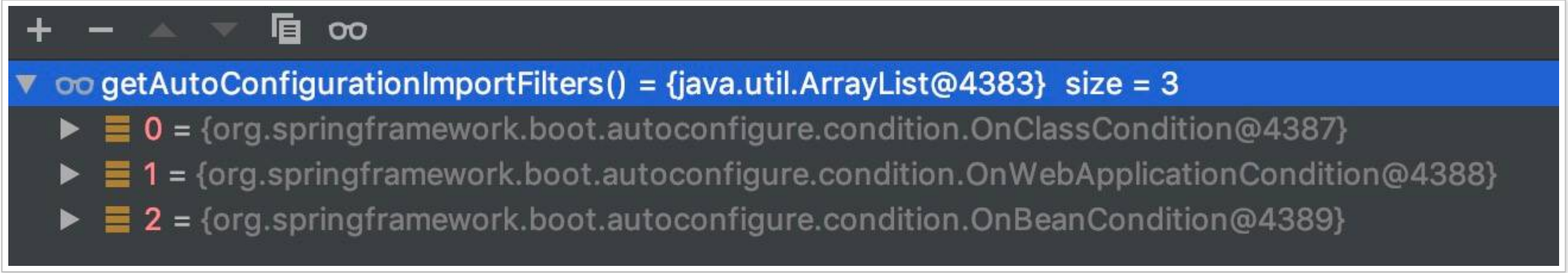
• <0> 处，这里是芬芳调皮加的。用于测试，如果去掉这块逻辑，是否需有影响。答案当然是，没有影响。这里就先不说原因，胖友自己思考下。实际上，本文也已经提及为什么了。

• <1> 处，声明需要用到的变量。每个变量，已经添加其对应的注释，不再赘述。

• <2> 处，调用 #getAutoConfigurationImportFilters() 方法，加载指定类型 AutoConfigurationImportFilter 对应的，在 META-INF/spring.factories 里的类名的数组。代码如下：

```
protected List<AutoConfigurationImportFilter> getAutoConfigurationImportFilters() {
    return SpringFactoriesLoader.loadFactories(AutoConfigurationImportFilter.class, this.beanClassLoader);
}
```

• 例如：



filters`

• 就是我们看到的 AutoConfigurationImportFilter 的三个最终实现类。

• <2.1> 、 <2.3> 、 <2.3> 处，就是对 AutoConfigurationImportFilter#filter(String[] autoConfigurationClasses, AutoConfigurationMetadata autoConfigurationMetadata) 方法的调用，妥妥的。是不是有点顺畅了。

• <3.1> 处，如果没有需要忽略的，直接返回 configurations 即可。

• <3.2> 处，如果存在需要忽略的，构建新的数组，排除掉忽略的。

当然，加载到的自动化配置类（Configuration）也会存在使用 @ConditionalOnProperty 等其它条件注解，但是不会在此处被过滤掉。芬芳猜测的原因，可能配置会从外部加载，此时暂时不太好判断。🐼
不一定正确，可以星球讨论一波哟。

8.1 OnClassCondition

org.springframework.boot.autoconfigure.condition.OnClassCondition ， 继承 FilteringSpringBootCondition 抽象类，给 @ConditionalOnClass 、 @ConditionalOnMissingClass 使用的 Condition 实现类。

8.1.1 getOutcomes

实现 #getOutcomes(String[] autoConfigurationClasses, AutoConfigurationMetadata autoConfigurationMetadata) 方法，代码如下：

```
@Override
protected final ConditionOutcome[] getOutcomes(String[] autoConfigurationClasses, AutoConfigurationMetadata autoConfigurationMetadata) {

    int split = autoConfigurationClasses.length / 2;

    OutcomesResolver firstHalfResolver = createOutcomesResolver(autoConfigurationClasses, 0, split, autoConfigurationMetadata);

    OutcomesResolver secondHalfResolver = new StandardOutcomesResolver(autoConfigurationClasses, split, autoConfigurationClasses.length, autoConfigurationMetadata, getBeanClassLoader());

    ConditionOutcome[] secondHalf = secondHalfResolver.resolveOutcomes();
    ConditionOutcome[] firstHalf = firstHalfResolver.resolveOutcomes();

    ConditionOutcome[] outcomes = new ConditionOutcome[autoConfigurationClasses.length];
    System.arraycopy(firstHalf, 0, outcomes, 0, firstHalf.length);
    System.arraycopy(secondHalf, 0, outcomes, split, secondHalf.length);
    return outcomes;
}
```

- <1> 处，考虑到配置类（Configuration）配置的 @ConditionalOnClass 、 @ConditionalOnMissingClass 注解中的类可能比较多，所以采用多线程提升效率。但是经过测试，分成两个线程，效率是最好的，所以这里才出现了 autoConfigurationClasses.length / 2 代码。

- <2.1> 处，调用 #createOutcomesResolver(String[] autoConfigurationClasses, int start, int end, AutoConfigurationMetadata autoConfigurationMetadata) 方法，创建一个 OutcomesResolver 对象。代码如下：

```
private OutcomesResolver createOutcomesResolver(String[] autoConfigurationClasses, int start, int end, AutoConfigurationMetadata autoConfigurationMetadata) {
    OutcomesResolver outcomesResolver = new StandardOutcomesResolver(autoConfigurationClasses, start, end, autoConfigurationMetadata, getBeanClassLoader());
    try {
        return new ThreadedOutcomesResolver(outcomesResolver);
    } catch (AccessControlException ex) {
        return outcomesResolver;
    }
}
```

- 首先，创建了一个 StandardOutcomesResolver 对象 outcomesResolver 。
- 然后，创建了 ThreadedOutcomesResolver 对象，将 outcomesResolver 包装在其中。注意噢，下文我们会看到，ThreadedOutcomesResolver 是启动了一个新线程，执行 StandardOutcomesResolver 的逻辑。

- <2.2> 处，将后半，创建一个 StandardOutcomesResolver 对象。

- 🐼 注意哟，创建的 StandardOutcomesResolver、ThreadedOutcomesResolver 对象，都是 OutcomesResolver 的子类。

- <3.1> 处，调用后半的 StandardOutcomesResolver#resolveOutcomes() 方法，执行解析（匹配）。

- <3.2> 处，调用前半的 ThreadedOutcomesResolver#resolveOutcomes() 方法，执行解析（匹配）。在 ThreadedOutcomesResolver 的实现里，会使用 Thread#join() 方法，保证新起的线程，能完成它的任务。这也是为什么，ThreadedOutcomesResolver 后执行的原因。

- <4> 处，创建 outcomes 结果数组，然后合并结果，最后返回。

8.1.2 OutcomesResolver

OutcomesResolver ， 是 OnClassCondition 的内部接口，结果解析器接口。代码如下：

```
private interface OutcomesResolver {
```

```
        ConditionOutcome[] resolveOutcomes();  
  
    }  
}
```

它的实现类有：

- [\[8.1.3 ThreadedOutcomesResolver\]](#)
- [\[8.1.4 StandardOutcomesResolver\]](#)

8.1.3 ThreadedOutcomesResolver

ThreadedOutcomesResolver，是 OnClassCondition 的内部类，实现 OutcomesResolver 接口，开启线程，执行 OutcomesResolver 的逻辑。代码如下：

```
private static final class ThreadedOutcomesResolver implements OutcomesResolver {  
  
    private final Thread thread;  
  
    private volatile ConditionOutcome[] outcomes;  
  
    private ThreadedOutcomesResolver(OutcomesResolver outcomesResolver) {  
  
        this.thread = new Thread(  
            () -> this.outcomes = outcomesResolver.resolveOutcomes());  
  
        this.thread.start();  
    }  
  
    @Override  
    public ConditionOutcome[] resolveOutcomes() {  
  
        try {  
            this.thread.join();  
        } catch (InterruptedException ex) {  
            Thread.currentThread().interrupt();  
        }  
  
        return this.outcomes;  
    }  
}
```

- `<1.1>`、`<1.2>` 处，在构建方法中，创建新的线程，并启动线程，从而调用 `OutcomesResolver#resolveOutcomes()` 方法，执行匹配逻辑。
- `<2.1>` 处，等待线程执行结束。
- `<2.2>` 处，返回结果。
- 🐼 是不是这里一看，就明白 [\[8.1.1 getOutcomes\]](#) 中，是这样的调用顺序了。

8.1.4 StandardOutcomesResolver

StandardOutcomesResolver，是 OnClassCondition 的内部类，实现 OutcomesResolver 接口，标准的 StandardOutcomesResolver 实现类。

8.1.4.1 构造方法

```
private final class StandardOutcomesResolver implements OutcomesResolver {
```

```
private final String[] autoConfigurationClasses;

private final int start;

private final int end;

private final AutoConfigurationMetadata autoConfigurationMetadata;

private final ClassLoader beanClassLoader;

private StandardOutcomesResolver(String[] autoConfigurationClasses, int start, int end, AutoConfigurationMetadata autoConfigurationMetadata, ClassLoader beanClassLoader) {
    this.autoConfigurationClasses = autoConfigurationClasses;
    this.start = start;
    this.end = end;
    this.autoConfigurationMetadata = autoConfigurationMetadata;
    this.beanClassLoader = beanClassLoader;
}

}
```

8.1.4.2 resolveOutcomes

#resolveOutcomes() 方法，执行批量匹配，并返回结果。代码如下：

```
@Override
public ConditionOutcome[] resolveOutcomes() {
    return getOutcomes(this.autoConfigurationClasses, this.start, this.end, this.autoConfigurationMetadata);
}

private ConditionOutcome[] getOutcomes(String[] autoConfigurationClasses, int start, int end, AutoConfigurationMetadata autoConfigurationMetadata) {

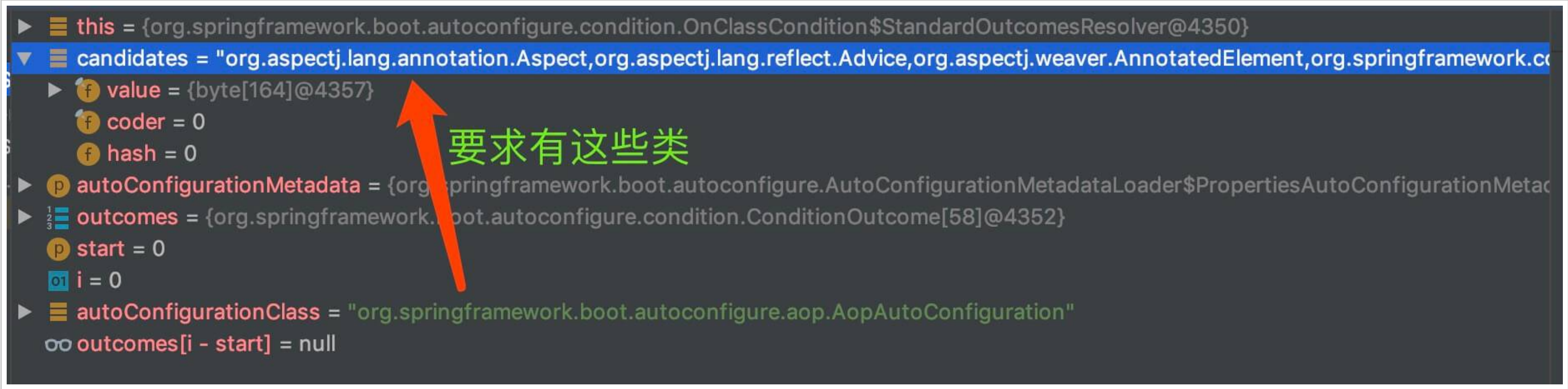
    ConditionOutcome[] outcomes = new ConditionOutcome[end - start];

    for (int i = start; i < end; i++) {
        String autoConfigurationClass = autoConfigurationClasses[i];
        if (autoConfigurationClass != null) {

            String candidates = autoConfigurationMetadata.get(autoConfigurationClass, "ConditionalOnClass");
            if (candidates != null) {

                outcomes[i - start] = getOutcome(candidates);
            }
        }
    }
    return outcomes;
}
```

- <1> 处，获得指定自动配置类的 @ConditionalOnClass 注解的要求类。例如下图：



`candidates` 示例

- <2> 处, 调用 #getOutcome(String candidates) 方法, 执行匹配。代码如下:

```
private ConditionOutcome getOutcome(String candidates) {  
    try {  
  
        if (!candidates.contains(",")) {  
            return getOutcome(candidates, this.beanClassLoader);  
        }  
  
        for (String candidate : StringUtils.commaDelimitedListToStringArray(candidates)) {  
  
            ConditionOutcome outcome = getOutcome(candidate, this.beanClassLoader);  
  
            if (outcome != null) {  
                return outcome;  
            }  
        }  
    } catch (Exception ex) {  
  
    }  
    return null;  
}
```

- <3> 处, 调用 #getOutcome(String className, ClassLoader classLoader) 方法, 执行匹配。代码如下:

```
private ConditionOutcome getOutcome(String className, ClassLoader classLoader) {  
    if (ClassNameFilter.MISSING.matches(className, classLoader)) {  
        return ConditionOutcome.noMatch(ConditionMessage  
            .forCondition(ConditionalOnClass.class).didNotFind("required class").items(Style.QUOTE, className));  
    }  
    return null;  
}
```

- 通过使用 ClassNameFilter.MISSING 来, 进行匹配类是否不存在。

看到此处, 我们会发现 [8.1.1 getOutcomes] 的整个逻辑, 暂时只做了 @ConditionalOnClass 注解的条件匹配, 还有一个 @ConditionalOnMissingClass 注解呢? 答案在 [8.1.5 getMatchOutcome] 。

8.1.5 getMatchOutcome

#getMatchOutcome(ConditionContext context, AnnotatedTypeMetadata metadata) 方法, 执行 @ConditionalOnClass 和 @ConditionalOnMissingClass 注解的匹配。代码如下:

```
@Override  
public ConditionOutcome getMatchOutcome(ConditionContext context, AnnotatedTypeMetadata metadata) {  
  
    ClassLoader classLoader = context.getClassLoader();  
    ConditionMessage matchMessage = ConditionMessage.empty();  
  
    List<String> onClasses = getCandidates(metadata, ConditionalOnClass.class);  
    if (onClasses != null) {  
  
        List<String> missing = filter(onClasses, ClassNameFilter.MISSING, classLoader);  
  
        if (!missing.isEmpty()) {  
            return ConditionOutcome.noMatch(ConditionMessage.forCondition(ConditionalOnClass.class)  
                .didNotFind("required class", "required classes").items(Style.QUOTE, missing));  
        }  
  
        matchMessage = matchMessage.andCondition(ConditionalOnClass.class)  
            .found("required class", "required classes").items(Style.QUOTE, filter(onClasses, ClassNameFilter.PRESENT, classLoader));  
    }  
  
    List<String> onMissingClasses = getCandidates(metadata, ConditionalOnMissingClass.class);  
    if (onMissingClasses != null) {  
  
        List<String> present = filter(onMissingClasses, ClassNameFilter.PRESENT, classLoader);  
  
        if (!present.isEmpty()) {  
            return ConditionOutcome.noMatch(ConditionMessage.forCondition(ConditionalOnMissingClass.class)  
                .found("unwanted class", "unwanted classes").items(Style.QUOTE, present));  
        }  
  
        matchMessage = matchMessage.andCondition(ConditionalOnMissingClass.class)  
            .didNotFind("unwanted class", "unwanted classes").items(Style.QUOTE, filter(onMissingClasses, ClassNameFilter.MISSING, classLoader));  
    }  
  
    return ConditionOutcome.noMatch(matchMessage);  
}
```

```
        return ConditionOutcome.match(matchMessage);
    }
}
```

- <1> 处，声明变量。
- <2> 处，获得 @ConditionalOnClass 注解的属性。后续的，通过使用 [7.2.3 filter] 方法，看看是否有缺失的类。
- <3> 处，获得 @ConditionalOnMissingClass 注解的属性。后续的，通过使用 [7.2.3 filter] 方法，看看是否有多余的类。
- <4> 处，返回匹配的结果。

8.2 OnWebApplicationCondition

org.springframework.boot.autoconfigure.condition.OnWebApplicationCondition，继承 FilteringSpringBootCondition 抽象类，给 @ConditionalOnWebApplication 和 @ConditionalOnNotWebApplication 使用的 Condition 实现类。

8.2.1 getOutcomes

#getOutcomes(String[] autoConfigurationClasses, AutoConfigurationMetadata autoConfigurationMetadata) 方法，代码如下：

```
@Override
protected ConditionOutcome[] getOutcomes(String[] autoConfigurationClasses, AutoConfigurationMetadata autoConfigurationMetadata) {

    ConditionOutcome[] outcomes = new ConditionOutcome[autoConfigurationClasses.length];

    for (int i = 0; i < outcomes.length; i++) {

        String autoConfigurationClass = autoConfigurationClasses[i];
        if (autoConfigurationClass != null) {

            outcomes[i] = getOutcome(autoConfigurationMetadata.get(autoConfigurationClass, "ConditionalOnWebApplication"));
        }
    }
    return outcomes;
}
```

- <1> 处，创建 outcomes 结果数组。
- <2> 处，遍历 autoConfigurationClasses 数组，调用 #getOutcome(String type) 方法，执行匹配。代码如下：

```
private static final String SERVLET_WEB_APPLICATION_CLASS = "org.springframework.web.context.support.GenericWebApplicationContext";
private static final String REACTIVE_WEB_APPLICATION_CLASS = "org.springframework.web.reactive.HandlerResult";

private ConditionOutcome getOutcome(String type) {
    if (type == null) {
        return null;
    }
    ConditionMessage.Builder message = ConditionMessage.forCondition(ConditionalOnWebApplication.class);

    if (ConditionalOnWebApplication.Type.SERVLET.name().equals(type)) {
        if (!ClassNameFilter.isPresent(SERVLET_WEB_APPLICATION_CLASS, getBeanClassLoader())) {
            return ConditionOutcome.noMatch(message.didNotFind("servlet web application classes").atAll());
        }
    }

    if (ConditionalOnWebApplication.Type.REACTIVE.name().equals(type)) {
        if (!ClassNameFilter.isPresent(REACTIVE_WEB_APPLICATION_CLASS, getBeanClassLoader())) {
            return ConditionOutcome.noMatch(message.didNotFind("reactive web application classes").atAll());
        }
    }

    if (!ClassNameFilter.isPresent(SERVLET_WEB_APPLICATION_CLASS, getBeanClassLoader())
        && !ClassUtils.isPresent(REACTIVE_WEB_APPLICATION_CLASS, getBeanClassLoader())) {
        return ConditionOutcome.noMatch(message.didNotFind("reactive or servlet web application classes").atAll());
    }
    return null;
}
```

- <2.1> 处，如果要求 SERVLET 类型，结果不存在 SERVLET_WEB_APPLICATION_CLASS 类，返回不匹配。
- <2.2> 处，如果要求 REACTIVE 类型，结果不存在 REACTIVE_WEB_APPLICATION_CLASS 类，返回不匹配。
- <2.3> 处，如果同时要求 SERVLET 和 REACTIVE 类型，且两者都不存在，返回不匹配。

- <2.3> 处, 如果 `SERVLET_WEB_APPLICATION_CLASS` 和 `REACTIVE_WEB_APPLICATION_CLASS` 都不存在, 返回不匹配。

8.2.2 getMatchOutcome

`#getMatchOutcome(ConditionContext context, AnnotatedTypeMetadata metadata)` 方法, 代码如下:

```
@Override
public ConditionOutcome getMatchOutcome(ConditionContext context, AnnotatedTypeMetadata metadata) {

    boolean required = metadata.isAnnotated(ConditionalOnWebApplication.class.getName());

    ConditionOutcome outcome = isWebApplication(context, metadata, required);

    if (required && !outcome.isMatch()) {
        return ConditionOutcome.noMatch(outcome.getConditionMessage());
    }

    if (!required && outcome.isMatch()) {
        return ConditionOutcome.noMatch(outcome.getConditionMessage());
    }

    return ConditionOutcome.match(outcome.getConditionMessage());
}
```

- <1> 处, 通过是否有 `@ConditionalOnWebApplication` 注解, 判断是否要求在 Web 环境下。为什么能这么判断呢? 因为 `@ConditionalOnNotWebApplication` 注解, 也能走进这个方法, 但是如果没有 `@ConditionalOnWebApplication` 注解, 就意味着有 `@ConditionalOnNotWebApplication` 注解, 也就是不要求 Web 环境。🤖 是不是有点绕~
- <2> 处, 调用 `#isWebApplication(ConditionContext context, AnnotatedTypeMetadata metadata, boolean required)` 方法, 判断是否匹配 Web 环境。详细解析, 见 [「8.2.3 isWebApplication」](#)。
- <3.1> 处, 如果要求, 结果不匹配 Web 环境, 返回最终不匹配。
- <3.2> 处, 如果不要求, 结果匹配 Web 环境, 返回最终不匹配。
- <3.3> 处, 返回匹配。

8.2.3 isWebApplication

`#isWebApplication(ConditionContext context, AnnotatedTypeMetadata metadata, boolean required)` 方法, 判断是否匹配 Web 环境。代码如下:

```
private ConditionOutcome isWebApplication(ConditionContext context, AnnotatedTypeMetadata metadata, boolean required) {
    switch (deduceType(metadata)) {
        case SERVLET:
            return isServletWebApplication(context);
        case REACTIVE:
            return isReactiveWebApplication(context);
        default:
            return isAnyWebApplication(context, required);
    }
}
```

- <1> 处, 调用 `#deduceType(AnnotatedTypeMetadata metadata)` 方法, 获得要求的 Web 类型。代码如下:

```
private Type deduceType(AnnotatedTypeMetadata metadata) {
    Map<String, Object> attributes = metadata.getAnnotationAttributes(ConditionalOnWebApplication.class.getName());
    if (attributes != null) {
        return (Type) attributes.get("type");
    }
    return Type.ANY;
}
```

- <2.1> 处, 调用 `#isServletWebApplication(context)` 方法, 判断是否 Servlet Web 环境。代码如下:

```
private ConditionOutcome isServletWebApplication(ConditionContext context) {
    ConditionMessage.Builder message = ConditionMessage.forCondition("");

    if (!ClassNameFilter.isPresent(SERVLET_WEB_APPLICATION_CLASS, context.getClassLoader())) {
        return ConditionOutcome.noMatch(message.didNotFind("servlet web application classes").atAll());
    }
}
```

```
if (context.getBeanFactory() != null) {

    String[] scopes = context.getBeanFactory().getRegisteredScopeNames();
    if (ObjectUtils.containsElement(scopes, "session")) {
        return ConditionOutcome.match(message.foundExactly("'session' scope"));
    }
}

if (context.getEnvironment() instanceof ConfigurableWebEnvironment) {
    return ConditionOutcome.match(message.foundExactly("ConfigurableWebEnvironment"));
}

if (context.getResourceLoader() instanceof WebApplicationContext) {
    return ConditionOutcome.match(message.foundExactly("WebApplicationContext"));
}

return ConditionOutcome.noMatch(message.because("not a servlet web application"));
}
```

- <2.2> 处，调用 #isReactiveWebApplication(ConditionContext context) 方法，代码如下：

```
private ConditionOutcome isReactiveWebApplication(ConditionContext context) {
    ConditionMessage.Builder message = ConditionMessage.forCondition("");

    if (!ClassNameFilter.isPresent(REACTIVE_WEB_APPLICATION_CLASS, context.getClassLoader())) {
        return ConditionOutcome.noMatch(message.didNotFind("reactive web application classes").atAll());
    }

    if (context.getEnvironment() instanceof ConfigurableReactiveWebEnvironment) {
        return ConditionOutcome.match(message.foundExactly("ConfigurableReactiveWebEnvironment"));
    }

    if (context.getResourceLoader() instanceof ReactiveWebApplicationContext) {
        return ConditionOutcome.match(message.foundExactly("ReactiveWebApplicationContext"));
    }

    return ConditionOutcome.noMatch(message.because("not a reactive web application"));
}
```

- <2.3> 处，调用 #isAnyWebApplication(ConditionContext context, boolean required) 方法，代码如下：

```
private ConditionOutcome isAnyWebApplication(ConditionContext context, boolean required) {
    ConditionMessage.Builder message = ConditionMessage.forCondition(ConditionalOnWebApplication.class, required ? "(required)" : "");

    ConditionOutcome servletOutcome = isServletWebApplication(context);
    if (servletOutcome.isMatch() && required) {
        return new ConditionOutcome(servletOutcome.isMatch(), message.because(servletOutcome.getMessage()));
    }

    ConditionOutcome reactiveOutcome = isReactiveWebApplication(context);
    if (reactiveOutcome.isMatch() && required) {
        return new ConditionOutcome(reactiveOutcome.isMatch(), message.because(reactiveOutcome.getMessage()));
    }

    return new ConditionOutcome(servletOutcome.isMatch() || reactiveOutcome.isMatch(),
        message.because(servletOutcome.getMessage()).append("and").append(reactiveOutcome.getMessage()));
}
```

8.3 OnBeanCondition

org.springframework.boot.autoconfigure.condition.OnBeanCondition ，继承 FilteringSpringBootCondition 抽象类，给 @ConditionalOnBean 、 @ConditionalOnMissingBean 、 @ConditionalOnSingleCandidate 使用的 Condition 实现类。

芳芳就暂时先不写了，因为这个类有点复杂，我想偷懒，哈哈哈。当然，感兴趣的胖友，可以看看 《SpringBoot @ConditionalOnBean、@ConditionalOnMissingBean 注解源码分析与示例》 一文。

本文以为是一篇半天就能解决的博客，结果写了一天半。希望尽可能覆盖到大多数细节。🐼

参考和推荐如下文章：

- dm_vincent

- [《\[Spring Boot\] 3. Spring Boot 实现自动配置的基础》](#)
 - [《\[Spring Boot\] 4. Spring Boot 实现自动配置的原理》](#)
-
- [oldflame-Jm 《Spring boot 源码分析 - Conditional（12）》](#)
-
- [一个努力的码农 《spring boot 源码解析 19-@Conditional 注解详解》](#)
-
- [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。