SpringBoot 源码总结

SpringApplication.run(Application.class, args) 这一行方法到底干了什么？

> 本文前置知识：[读 Spring 源码总结](./读 Spring 源码总结.pdf) 与 [SPI 机制](./SPI 机制以及 jdbc 打破双亲委派.pdf)，基于 jdk11

注册初始化器和监听器

> 关键词：加载并设置初始化器和监听器、记录主配置类、META-INF/spring.factories、getSpringFactoriesInstances、loadFactoryNames、loadSpringFactories。

点进 run 方法，事实上调用了：

```
public static ConfigurableApplicationContext run(Class<?>[]
primarySources, String[] args) {
   return new SpringApplication(primarySources).run(args);
}
```

此处 primarySources是我们的主类，跟进查看 `new SpringApplication()` 的逻辑：

```
public SpringApplication(ResourceLoader resourceLoader, Class<?>...
primarySources) {
   this.resourceLoader = resourceLoader;
   this.primarySources = new LinkedHashSet<>
```

```
(Arrays.asList(primarySources));
   this.webApplicationType = WebApplicationType.deduceFromClasspath();
   setInitializers((Collection)
getSpringFactoriesInstances(ApplicationContextInitializer.class));
   setListeners((Collection)
getSpringFactoriesInstances(ApplicationListener.class));
   this.mainApplicationClass = deduceMainApplicationClass();
}
```

这个初始化代码干了啥呢？首先添加了主要资源，然后设置了初始化器和监听器，最后跟踪栈堆信息推导出主配置类 mainApplicationClass，
getSpringFactoriesInstances 方法很**重要**，看看具体的逻辑：

```
private <T> Collection<T> getSpringFactoriesInstances(Class<T> type,
Class<?>[] parameterTypes, Object... args) {
    ClassLoader classLoader = getClassLoader();
    Set<String> names = new LinkedHashSet<>
(SpringFactoriesLoader.loadFactoryNames(type, classLoader));
    List<T> instances = createSpringFactoriesInstances(type,
parameterTypes, classLoader, args, names);
    AnnotationAwareOrderComparator.sort(instances);
    return instances;
}
```

getSpringFactoriesInstances 方法通过 **loadFactoryNames** 获取了一系列的 names，**然后反射创造实例返回**，继续跟进 **SpringFactoriesLoader.loadFactoryNames** 方法，发现主要核心是 loadSpringFactories 方法：

```
public static List<String> loadFactoryNames(Class<?> factoryType,
@Nullable ClassLoader classLoader) {
    String factoryTypeName = factoryType.getName();
    return loadSpringFactories(classLoader).getOrDefault(factoryTypeName,
Collections.emptyList());
}

private static Map<String, List<String>> loadSpringFactories(@Nullable
ClassLoader classLoader) {
    MultiValueMap<String, String> result = cache.get(classLoader);
    if (result != null)  return result;
    Enumeration<URL> urls =
classLoader.getResources(FACTORIES_RESOURCE_LOCATION);
    result = new LinkedMultiValueMap<>();
    while (urls.hasMoreElements()) {
        URL url = urls.nextElement();
        UrlResource resource = new UrlResource(url);
        Properties properties =
PropertiesLoaderUtils.loadProperties(resource);
        for (Map.Entry<?, ?> entry : properties.entrySet()) {
            String factoryTypeName = ((String) entry.getKey()).trim();
```

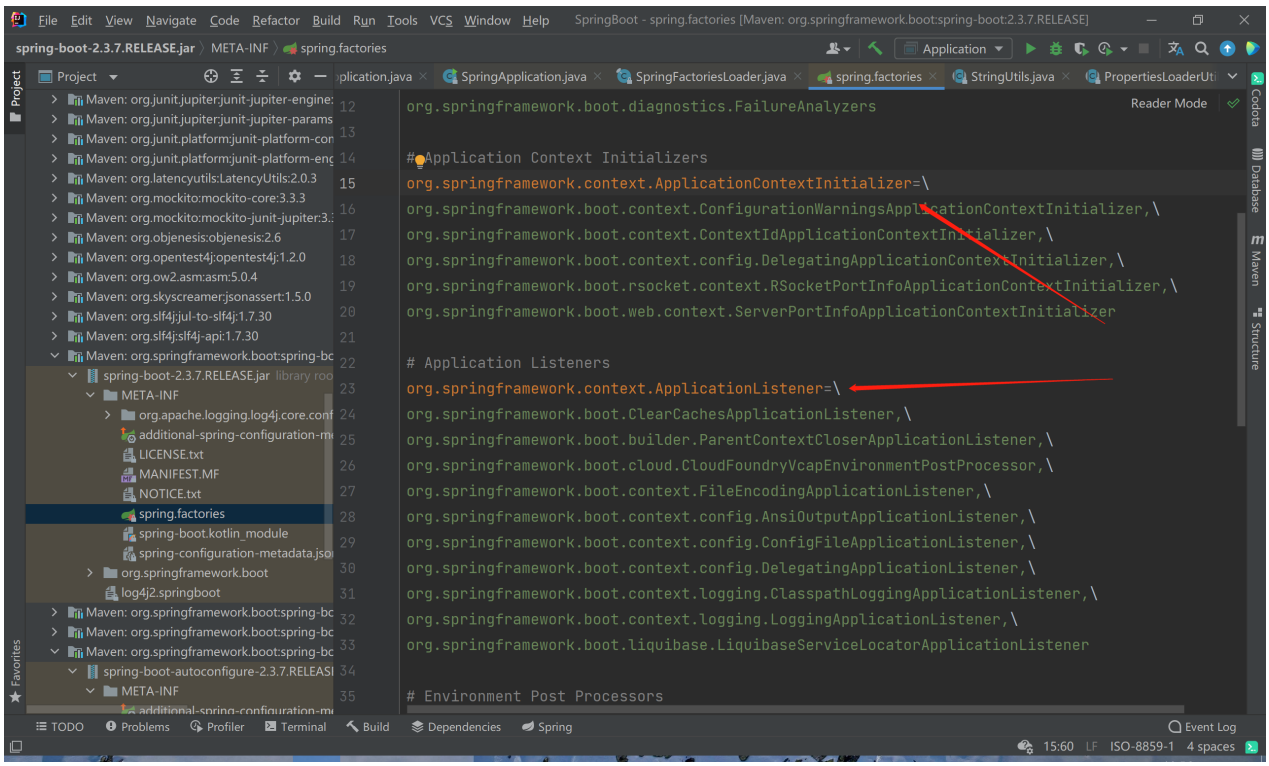
```
            for (String factoryImplementationName :
StringUtils.commaDelimitedListToStringArray((String) entry.getValue())) {
                result.add(factoryTypeName,
factoryImplementationName.trim());
            }
        }
    }
    cache.put(classLoader, result);
    return result;
}
```

其中 `String FACTORIES_RESOURCE_LOCATION = "META-INF/spring.factories"`，loadSpringFactories 函数查找了 classPath 下 **META-INF/spring.factories** 文件，并把所有的资源以键值对的形式放进缓存中，loadFactoryNames 返回对应键的所有元素（全限定名）。

现在来回顾下这两行代码是干啥的：

```
setInitializers(getSpringFactoriesInstances(ApplicationContextInitializer
.class));
setListeners(getSpringFactoriesInstances(ApplicationListener.class));
```

getSpringFactoriesInstances(ApplicationContextInitializer.class) 加载了所有 META-INF/spring.factories 文件下的所有名称为 ApplicationContextInitializer 的元素，例如：



也就是说这些类会在这一步被加载进去，至于这些类都是干嘛的，读者可点进对应的类查看对应的注解。

再次说明 loadFactoryNames、loadSpringFactories 这两个方法很重要！

loadSpringFactories 加载 MEAT-INFO/factories 下的所有键值对，每个值都是一个类的全限定名，loadFactoryNames 根据 loadSpringFactories 加载的 Map 选取对应的值返回！

设置系统环境变量

> 关键词：设置系统环境变量、StandardEnvironment 类

进入 public ConfigurableApplicationContext **run**(String... args) 方法，其中有一行代码是：

```
ConfigurableEnvironment environment = prepareEnvironment(listeners,
applicationArguments);
```

这一行代码将帮我们设置环境变量，在 prepareEnvironment 方法内调用了 getOrCreateEnvironment() 方法，跟进方法发现是一个 switch 判断，具体返回环境属性类什么取决于具体的情况，例如 Web 应用将返回 StandardServletEnvironment()。

```
private ConfigurableEnvironment getOrCreateEnvironment() {
   if (this.environment != null) {
      return this.environment;
   }
   switch (this.webApplicationType) {
   case SERVLET:
      return new StandardServletEnvironment();
   case REACTIVE:
      return new StandardReactiveWebEnvironment();
   default:
      return new StandardEnvironment();
   }
}
```

但不管怎么样都是继承了 StandardEnvironment 类，而 StandardEnvironment 类又继承了 AbstractEnvironment 类，因此无论如何都会触发 AbstractEnvironment 类的初始化：

```
public AbstractEnvironment() {
   customizePropertySources(this.propertySources);
}
```

而初始化又调用了子类 StandardEnvironment 的方法，

```
protected void customizePropertySources(MutablePropertySources
propertySources) {
   propertySources.addLast(
         new
PropertiesPropertySource(SYSTEM_PROPERTIES_PROPERTY_SOURCE_NAME,
getSystemProperties()));
   propertySources.addLast(
         new
SystemEnvironmentPropertySource(SYSTEM_ENVIRONMENT_PROPERTY_SOURCE_NAME,
getSystemEnvironment()));
}
```

getSystemProperties() 和 getSystemEnvironment() 其实就是返回了 System.getProperties() 和 System.getenv()，这就是我们的系统环境变量。

注册注解处理增强器并注册 beanFactory

> 反射、createApplicationContext、DefaultListableBeanFactory、注册默认的增强器、ConfigurationClassPostProcessor。

还是在 public ConfigurableApplicationContext **run**(String... args) 方法中，有一行代码 `context = createApplicationContext();` 创建了 ConfigurableApplicationContext 上下文，点进这个方法：

```
protected ConfigurableApplicationContext createApplicationContext() {
   Class<?> contextClass = this.applicationContextClass;
   if (contextClass == null) {
       switch (this.webApplicationType) {
           case SERVLET:
               contextClass =
Class.forName(DEFAULT_SERVLET_WEB_CONTEXT_CLASS);
               break;
           case REACTIVE:
               contextClass =
Class.forName(DEFAULT_REACTIVE_WEB_CONTEXT_CLASS);
               break;
           default:
               contextClass = Class.forName(DEFAULT_CONTEXT_CLASS);
       }
   }
   return (ConfigurableApplicationContext)
BeanUtils.instantiateClass(contextClass);
}
```

不要看这个方法很简单，但细节是魔鬼，我这里是进入了第一个 case 语句（Web 环境），也就是调用了 Class.forName(DEFAULT_SERVLET_WEB_CONTEXT_CLASS) 语句，DEFAULT_SERVLET_WEB_CONTEXT_CLASS 是：

```
"org.springframework.boot.web.servlet.context.AnnotationConfigServletWebS
erverApplicationContext"
```

> 这个上下文内置了 Tomcat，在 onRefresh 时会初始化 tomcat。

也就是说我应该在 BeanUtils.instantiateClass 方法中实例化这个上下文容器，点进这个上下文容器的构造器：

```
public AnnotationConfigServletWebServerApplicationContext() {
   this.reader = new AnnotatedBeanDefinitionReader(this);
   this.scanner = new ClassPathBeanDefinitionScanner(this);
}
```

首先不要看这个代码，要知道这肯定会向上调用父类构造器，跟踪发现持续调用了 ServletWebServerApplicationContext、GenericWebApplicationContext、GenericApplicationContext 类的构造器方法，在 GenericApplicationContext 类中：

```
public GenericApplicationContext() {
   this.beanFactory = new DefaultListableBeanFactory();
}
```

此时，DefaultListableBeanFactory 被注册。

然后在回到 AnnotationConfigServletWebServerApplicationContext() 方法中，这里注册了两个解析器，点入 AnnotatedBeanDefinitionReader 类的构造器，持续调用父类的构造器，最终调用了一行代码：

```
AnnotationConfigUtils.registerAnnotationConfigProcessors(this.registry);
```

点入这行代码，跳到了 AnnotationConfigUtils 类中执行，代码很长，但逻辑很简单，就是将几个硬编码的增强器加到容器上下文中，例如：

```
essor': Root bean: class [org.springframework.context.annotation.ConfigurationClassPostProcessor]; s
or': Root bean: class [org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostPr
r': Root bean: class [org.springframework.context.annotation.CommonAnnotationBeanPostProcessor]
: class [org.springframework.context.event.EventListenerMethodProcessor]; scope=; abstract=false; la
lass [org.springframework.context.event.DefaultEventListenerFactory]; scope=; abstract=false; lazyIni
```

SpringBoot 中最最最重要的增强器 ConfigurationClassPostProcessor 在这里诞生了。

ClassPathBeanDefinitionScanner 这一行代码类似，添加了一些过滤器到容器内。

加载源类到容器中

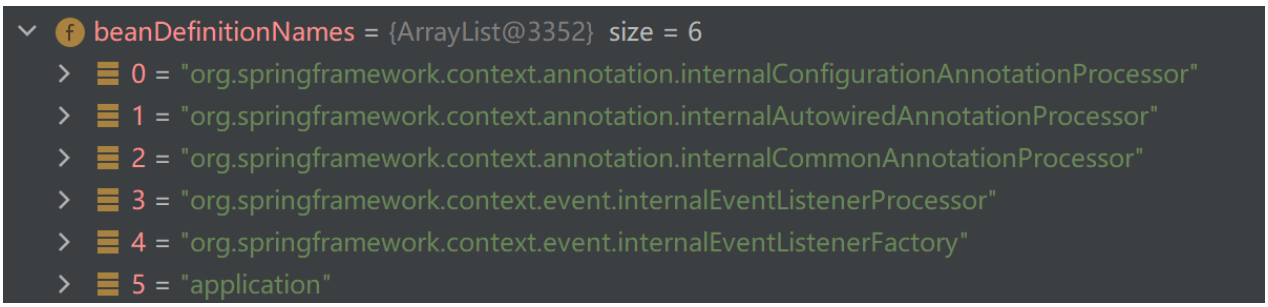
> 关键词：prepareContext 方法、加载主配置类到容器中（未实例化）。

继续回到 run 方法中，可以发现 SpringBoot 又从 factorys 文件注册了异常播报者，但这不是我们关心的，注意方法：`prepareContext`，在这个方法内有一个核心逻辑：

```
Set<Object> sources = getAllSources(); //
allSources.addAll(this.primarySources);
load(context, sources.toArray(new Object[0]));
```

getAllSources 其实加载了一些源，默认情况下就是我们传入的 primarySources，即主类，在 load 方法内被添加到容器中（未实例化）。

回到 run 方法：



成功加载了源类到 beanFactory 中！

注册 JVM 关闭钩子

你一定很好奇为什么关闭虚拟机时 springboot 还会输出一些日志，这是因为它向 JVM 注册了关闭钩子！

```
private void refreshContext(ConfigurableApplicationContext context) {
   if (this.registerShutdownHook) {
      try {
         // 注册关闭狗子
         context.registerShutdownHook();
      }
      catch (AccessControlException ex) {
         // Not allowed in some environments.
      }
   }
   refresh((ApplicationContext) context);
}
```

执行 ConfigurationClassPostProcessor 增强器

> 关键词：refresh 方法、ConfigurationClassPostProcessor 增强器、主配置类 — 候选者、ConfigurationClassParser.parse 方法。

现在终于进入 refresh 方法，这是 spring 的知识点，就不再说了，现在到执行增强器的时候了，spirngboot 会调用 ConfigurationClassPostProcessor 类的 postProcessBeanDefinitionRegistry 方法（这个方法与 postProcessBeanFactory 方法具有同样的语义），然后调用了 processConfigBeanDefinitions 方法，在这个方法内，我们调用了：

```
ConfigurationClassParser parser = new ConfigurationClassParser(
      this.metadataReaderFactory, this.problemReporter, this.environment,
      this.resourceLoader, this.componentScanBeanNameGenerator,
registry);
do {
    parser.parse(candidates);
} while (!candidates.isEmpty());
```

现在开始执行 ConfigurationClassParser.parse 了，而 candidates 就是我们的主配置类，这与之前 springboot 费尽心思加载主配置类也对应上了。

跟进 parse 方法内，核心代码是：

```
public void parse(Set<BeanDefinitionHolder> configCandidates) {
   for (BeanDefinitionHolder holder : configCandidates) {
      BeanDefinition bd = holder.getBeanDefinition();
      parse(((AnnotatedBeanDefinition) bd).getMetadata(),
holder.getBeanName());
   }
   this.deferredImportSelectorHandler.process();
}
```

this.deferredImportSelectorHandler.process() 这个方法很重要，先记下来，等会再说。

再跟进重载的 parse 方法，主要调用了 processConfigurationClass 方法，这个方法内有一个很重要的逻辑：

```
SourceClass sourceClass = asSourceClass(configClass, filter);
do {
   sourceClass = doProcessConfigurationClass(configClass, sourceClass,
filter);
}
while (sourceClass != null);
```

doProcessConfigurationClass 是真正做事的方法了，并且，如果 doProcessConfigurationClass 方法返回不是 null，那么会循环调用！

doProcessConfigurationClass 解析 Configuration 类上的注解

> 关键词：解析主配置类上的注解、@Import、getImports 方法、找到 @EnableAutoConfiguration 注解中的 @Import(AutoConfigurationImportSelector.class)。

这里面会执行对 @ComponentScan 的扫描，我们的启动类上的注解 @SpringBootApplication 注解，这个注解内就包含了 @ComponentScan 注解：

```
@ComponentScan(excludeFilters = {
    @Filter(type = FilterType.CUSTOM,
    classes = TypeExcludeFilter.class),
    @Filter(type = FilterType.CUSTOM, classes =
AutoConfigurationExcludeFilter.class) })
```

也就是说这里 springboot 会扫描和主类在同一个包下的所有被@Controller，@Service，@Repository，@Component 标注的类，将它们注册到容器中，然后**递归**的对这些类执行 parse 方法重新解析。

现在假设容器已经递归的解析好了其他的类，重新开始解析我们的主类，有一行关键的代码是：

```
processImports(configClass, sourceClass, getImports(sourceClass), filter,
true);
```

重点是这个函数的参数：getImports(sourceClass) 方法，跟进方法内：

```
private Set<SourceClass> getImports(SourceClass sourceClass) throws
IOException {
    Set<SourceClass> imports = new LinkedHashSet<>();
    Set<SourceClass> visited = new LinkedHashSet<>();
    collectImports(sourceClass, imports, visited);
    return imports;
}
private void collectImports(SourceClass sourceClass, Set<SourceClass>
imports, Set<SourceClass> visited) {
   if (visited.add(sourceClass)) {
      for (SourceClass annotation : sourceClass.getAnnotations()) {
         String annName = annotation.getMetadata().getClassName();
         if (!annName.equals(Import.class.getName())) {
            collectImports(annotation, imports, visited);
         }
      }

imports.addAll(sourceClass.getAnnotationAttributes(Import.class.getName()
, "value"));
```

```
    }
}
```

主要逻辑在 collectImports 方法内，这个方法啥意思？看 `if (!annName.equals(Import.class.getName()))` ，其实这个方法就是搜集类上的注解，然后继续递归的跟踪这些注解，看看注解上有没有 @Import，有就加入集合，然后继续跟踪注解...递归调用，同时用 visited 防止死循环。

简言之，就是搜集类上或从其他类继承的所有 @Import 注解，将这些注解信息添加到集合内。我们的启动类上应该是有两个 @Import 的，他们在 **@EnableAutoConfiguration** 和 @AutoConfigurationPackage 中。

```
"org.springframework.boot.autoconfigure.AutoConfigurationPackages$Registrar"
"org.springframework.boot.autoconfigure.AutoConfigurationImportSelector"
```

拿到资源后将执行 processImports 方法，这个方法会执行普通的 @Import 注解类，但不是我们的重点，这个我们马上会说。

自动装配

> 请先了解 @Import 基本知识
>
> 关键词：延迟导入、getAutoConfigurationEntry 方法、loadFactoryNames 、loadSpringFactories、获取自动装配类、EnableAutoConfiguration.class、过滤自动装配类、META-INF/spring-autoconfigure-metadata.properties。

@Import(AutoConfigurationImportSelector.class) 中的注解并不会在 processImports 中执行，这是因为 AutoConfigurationImportSelector 实现了 DeferredImportSelector 接口，标志自己被**延迟导入**。

回到 parse 方法：

```
public void parse(Set<BeanDefinitionHolder> configCandidates) {
   for (BeanDefinitionHolder holder : configCandidates) {
      BeanDefinition bd = holder.getBeanDefinition();
      parse(((AnnotatedBeanDefinition) bd).getMetadata(),
holder.getBeanName());
   }
   this.deferredImportSelectorHandler.process();
}
```

this.deferredImportSelectorHandler.process() 是处理延迟导入的关键方法，跟进 process() 方法，这个方法调用了：

```
DeferredImportSelectorGroupingHandler handler = new
DeferredImportSelectorGroupingHandler();
handler.processGroupImports();
```

点进 handler.processGroupImports() 方法内有一行：

```
grouping.getImports().forEach(......)
```

跟进 grouping.getImports() 内：

```
public Iterable<Group.Entry> getImports() {
   for (DeferredImportSelectorHolder deferredImport :
this.deferredImports) {

this.group.process(deferredImport.getConfigurationClass().getMetadata(),
            deferredImport.getImportSelector());
   }
   return this.group.selectImports();
}
```

核心方法是 this.group.process，继续跟进：

```
@Override
public void process(AnnotationMetadata annotationMetadata,
DeferredImportSelector deferredImportSelector) {
   AutoConfigurationEntry autoConfigurationEntry =
((AutoConfigurationImportSelector)
deferredImportSelector).getAutoConfigurationEntry(annotationMetadata);
   this.autoConfigurationEntries.add(autoConfigurationEntry);
   for (String importClassName :
autoConfigurationEntry.getConfigurations()) {
      this.entries.putIfAbsent(importClassName, annotationMetadata);
   }
}
```

至关重要的一行代码：getAutoConfigurationEntry(annotationMetadata)，跟进这个方法：

```
protected AutoConfigurationEntry
getAutoConfigurationEntry(AnnotationMetadata annotationMetadata) {
   AnnotationAttributes attributes = getAttributes(annotationMetadata);
   // 获取候选类
   List<String> configurations =
getCandidateConfigurations(annotationMetadata, attributes);
   // 过滤候选类
   configurations = getConfigurationClassFilter().filter(configurations);
```

```
    return new AutoConfigurationEntry(configurations, exclusions);
}
```

这个方法主要就两个逻辑：**获取自动装配类和过滤自动装配类**，一个一个看！

获取自动装配类

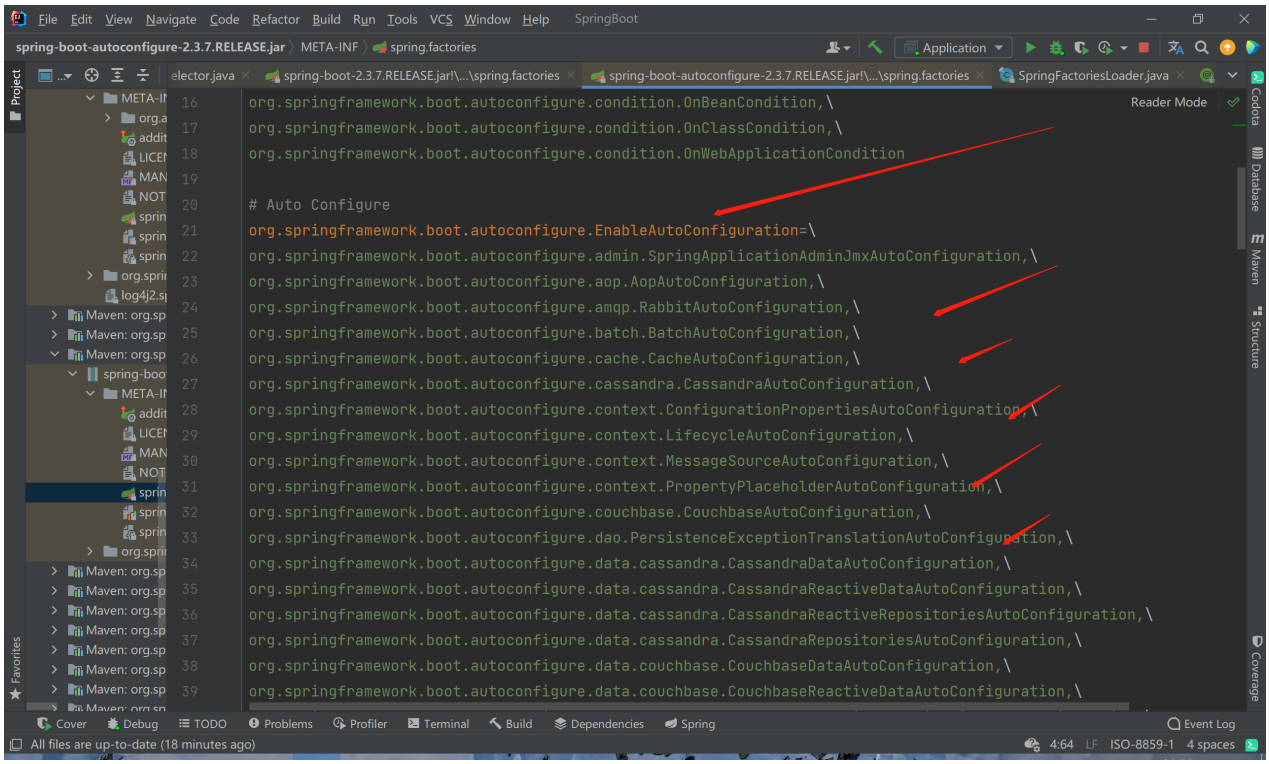
点进 getCandidateConfigurations 方法：

```
protected List<String> getCandidateConfigurations(AnnotationMetadata
metadata, AnnotationAttributes attributes) {
   List<String> configurations =
SpringFactoriesLoader.loadFactoryNames(getSpringFactoriesLoaderFactoryCla
ss(),
         getBeanClassLoader());
   return configurations;
}
```

看来核心是 SpringFactoriesLoader.loadFactoryNames 方法了，跟进这个方法，发现竟然调用了 loadFactoryNames 和 loadSpringFactories 方法！这两个方法我们在第一步的时候就已经详细讲了！我们来看看具体的参数（key）：getSpringFactoriesLoaderFactoryClass()：

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

是 EnableAutoConfiguration！也就是说这一步会拿到所有 key = EnableAutoConfiguration 的值：



类多的数不胜数，所有全限定名都被加载返回，这就是自动装配核心！

> \ 是换行的意思，按照上面的分析，这里肯定走缓存了，因为最开始在注册监听器的时候已经加载过一次 Names 了。

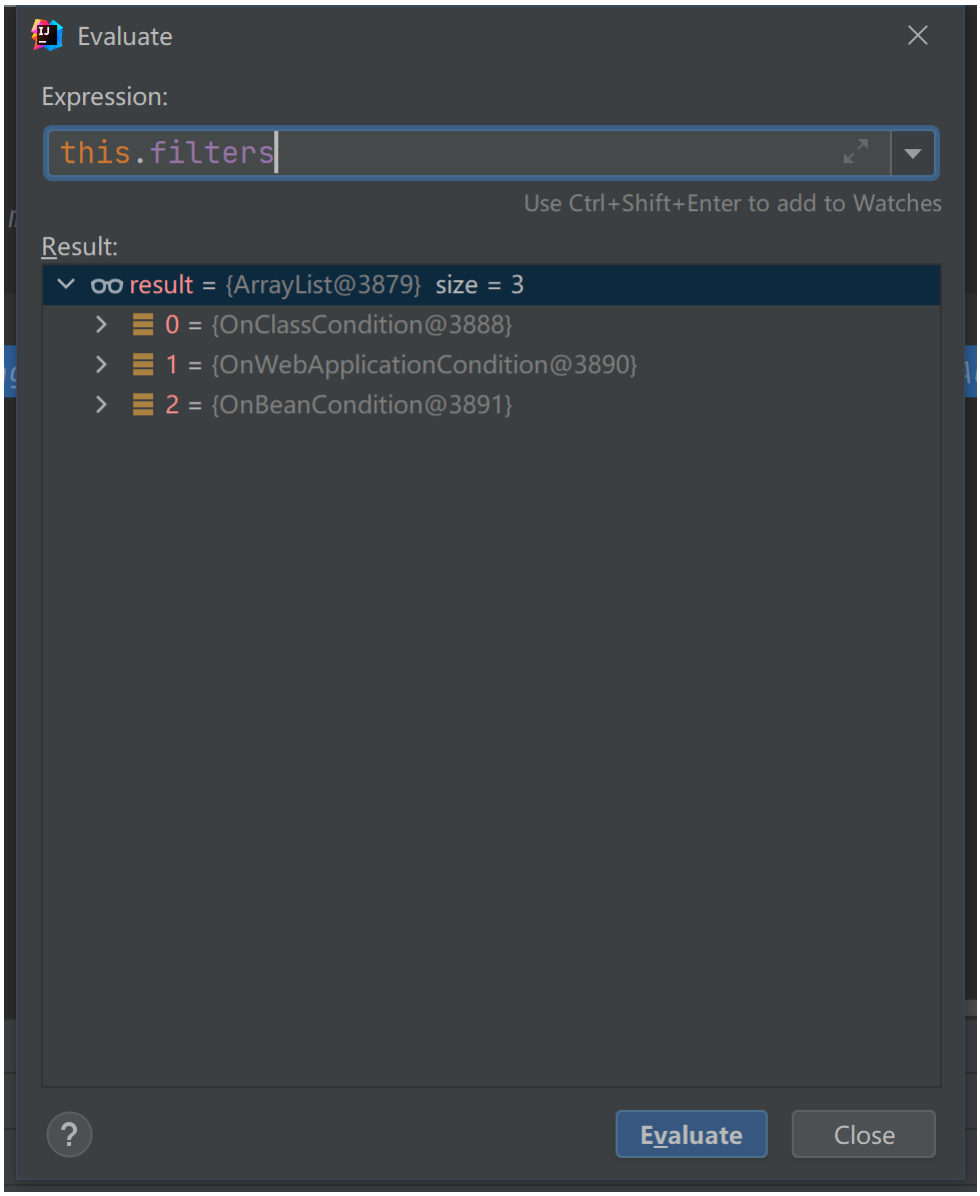
过滤自动装配类

> 须熟悉 OnCondition 相关知识

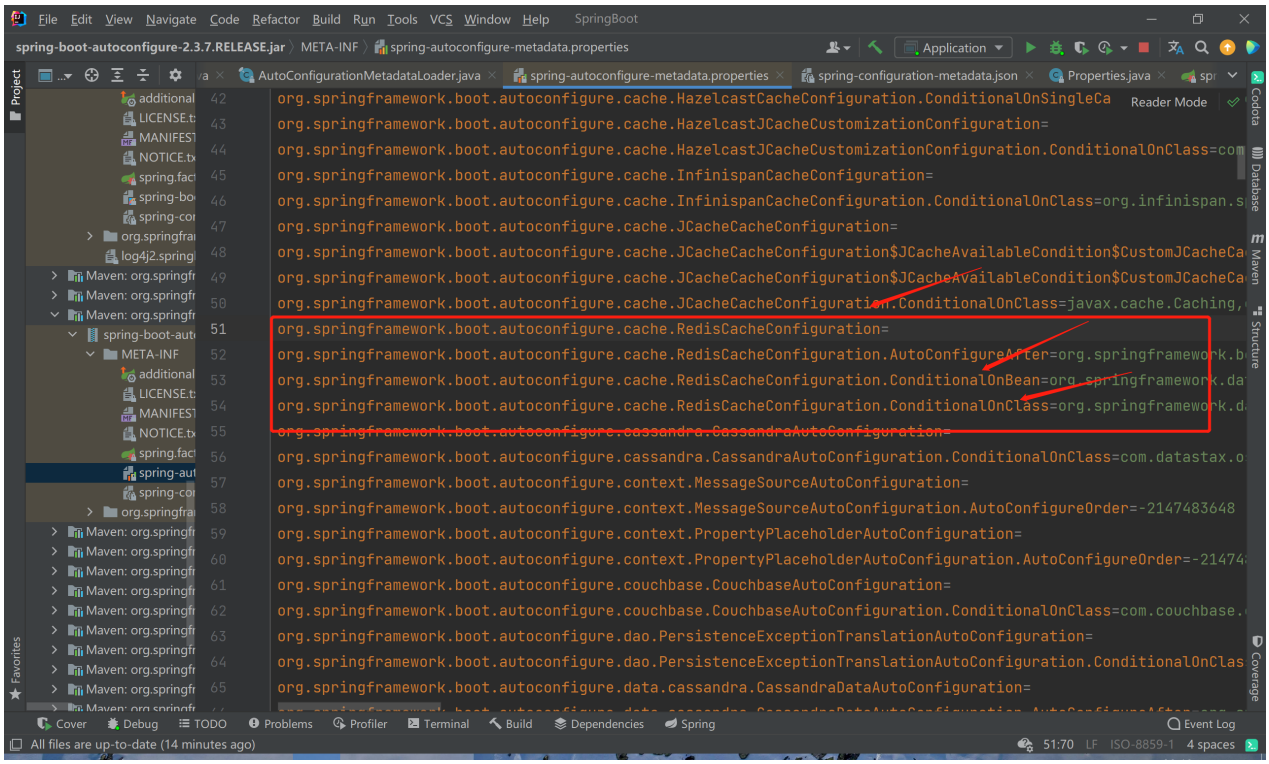
跟进 getConfigurationClassFilter().filter(configurations) 的 filter 方法，configurations 是我们刚刚获得的自动装配类的全限定名：

```
List<String> filter(List<String> configurations) {
   String[] candidates = StringUtils.toStringArray(configurations);
   for (AutoConfigurationImportFilter filter : this.filters) {
      boolean[] match = filter.match(candidates,
this.autoConfigurationMetadata);
      for (int i = 0; i < match.length; i++) {
         if (!match[i]) {
            candidates[i] = null;
         }
      }
   }
   List<String> result = new ArrayList<>(candidates.length);
   for (String candidate : candidates) {
      if (candidate != null) {
         result.add(candidate);
      }
   }
}
```

这个方法遍历 this.filters 来执行 match 方法，只要有一个不符合就过滤掉，this.filters 是什么呢？



看到 OnCondition 应该很熟悉了，这里面的方法比较绕，就不放源码了，大致的是在 **AutoConfigurationMetadataLoader** 类内，对 PATH = “**META-INF/spring-autoconfigure-metadata.properties**” 下的文件进行了资源加载，然后根据对应的规则（onClass 还是 onBean），去判断对应的 key 是否符合条件：

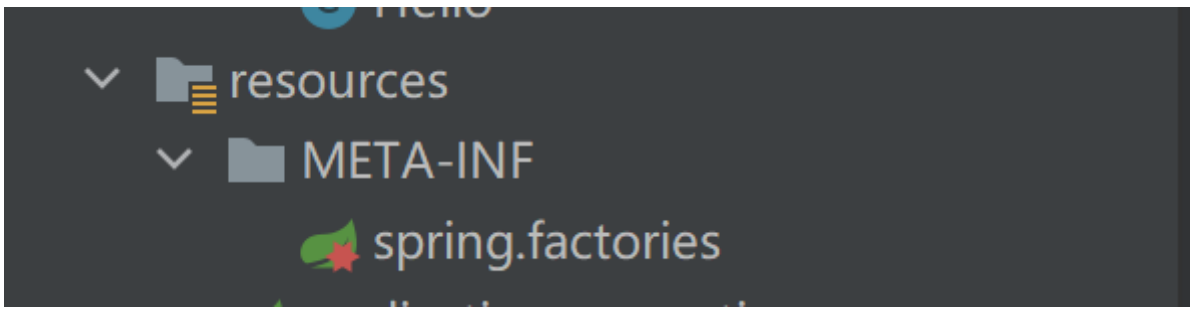


手写自动装配组件

定义一个类 Hello：

```
public class Hello {
    public void hello() {
        System.out.println("Hello World!");
    }
}
```

按照要求建立资源文件：



按照要求填写类的全限定名：

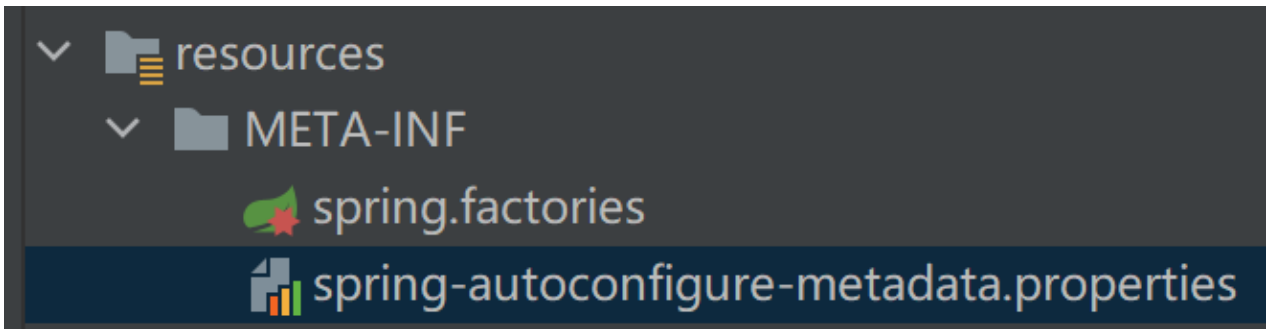
```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.happysnaker.Hello
```

打 jar 包，然后重启另一个项目导入 jar 包，运行 test（爆红别怕）：

```
@SpringBootTest
class ApplicationTests {

    @Autowired
    Hello hello;

    @Test
    void contextLoads() {
        hello.hello();
    }

}
```

成功输出：Hello World!

再建个文件：



填入：

```
com.happysnaker.Hello=
com.happysnaker.Hello.ConditionalOnClass=com.happysnaker.H
```

重新打包，运行程序报错！说明被过滤了！

新建一个类 H：

```
public class H {
}
```

重新打包运行，成功输出：Hello World!

总结

按照目录的顺序下来就是启动的所有流程了，如果能围绕着其中的关键字思考应该没多大问题了。