AutoConfigurationImportSelector到底怎么初始化

1. 前言

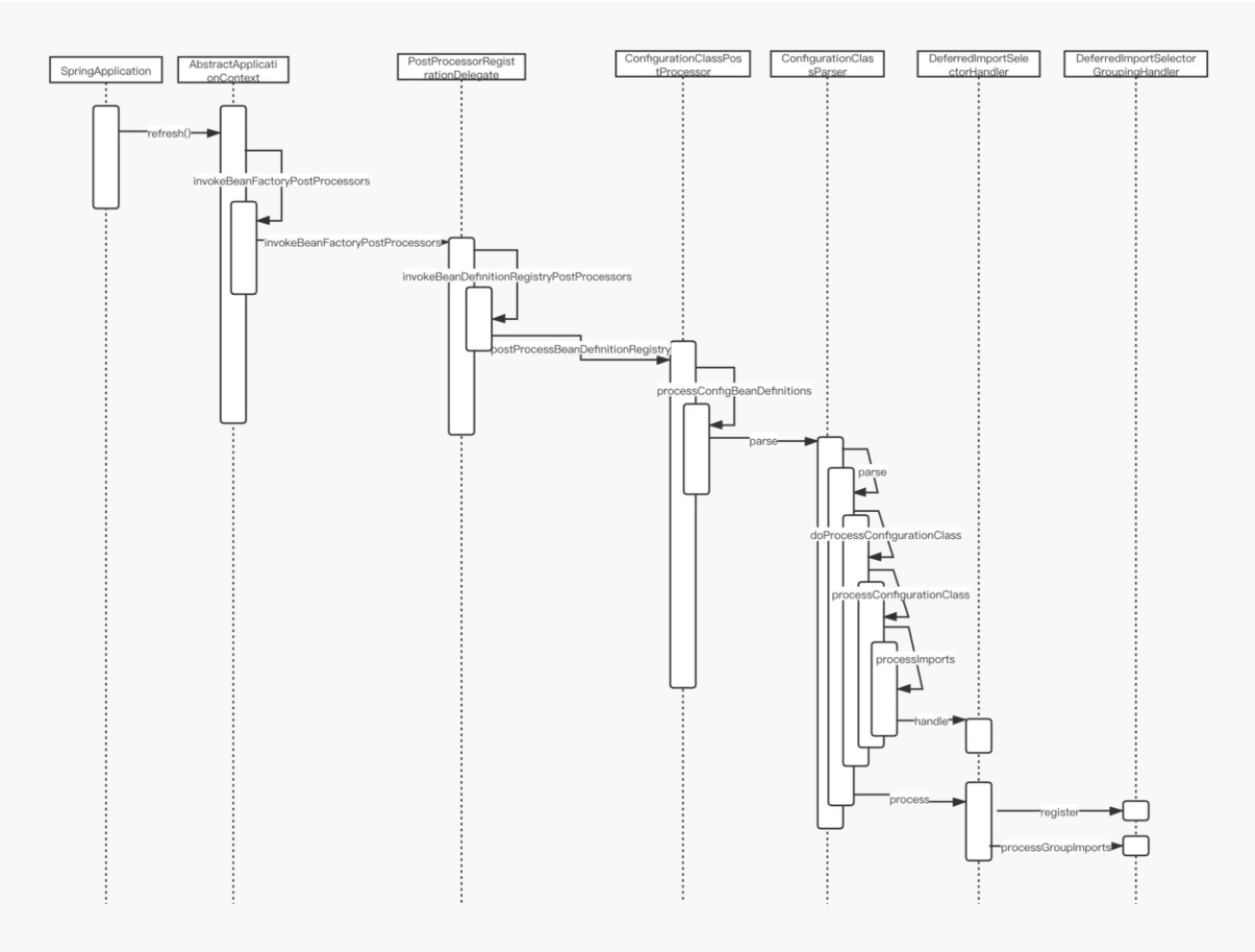
我们知道，在spring中，一般的实现ImportSelector接口，然后重写selectImports方法，就可以使用到spring的SPI技术，加载spring.factories中配置的`org.springframework.boot.autoconfigure.EnableAutoConfiguration.EnableAutoConfiguration`的类。

抱着测试的心态，给`@SpringBootApplication`的注解上实现的Selector`AutoConfigurationImportSelector#selectImports`打上断点测试，这一测试，不得了，心态崩了，debug没进去。Demo有问题？换上开发的项目，还是没有进debug，难道是大家说的有问题？不行，我这暴脾气忍不了，要一探究竟。

> *Note: 本文基于SpringBoot：2.6.1。部分解析写在代码的注释上*

2. 序列图

先摆上debug的时序图，方法返回没画（太丑，me嫌弃，有个大概了解一下就行）



ImportSelect时序图

3. 代码分析

代码太多，跳过了部分简单代码，然后挑关键点说。

> *1.* 从这`SpringApplication#refresh`当入口

```
    protected void refresh(ConfigurableApplicationContext applicationContext) {
    // 调用父类的refresh方法
        applicationContext.refresh();
    }
```

1. 进入 `org.springframework.context.support.AbstractApplicationContext#refresh`方法

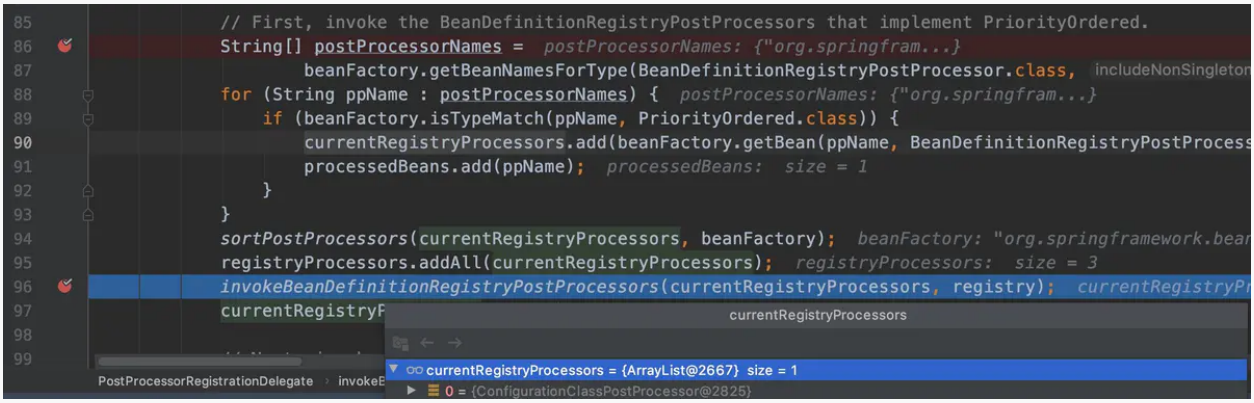
```
public void refresh() throws BeansException, IllegalStateException {
        synchronized (this.startupShutdownMonitor) {

            // Invoke factory processors registered as beans in the context.
      // 调用BeanFactory前置处理器
            invokeBeanFactoryPostProcessors(beanFactory);
    }
```

1. 进入 `org.springframework.context.support.PostProcessorRegistrationDelegate#invokeBeanFactoryPostProcessors`，这有关键点，就是生成的`postProcessorNames`

```
            // First, invoke the BeanDefinitionRegistryPostProcessors that implement PriorityOrdered.
            //
            String[] postProcessorNames = beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true, false);

            for (String ppName : postProcessorNames) {
```

```
                if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
 currentRegistryProcessors.add(beanFactory.getBean(ppName, BeanDefinitionRegistryPostProcessor.class));
                    processedBeans.add(ppName);
                }
            }

            ...
        // 调用BeanDefinitionRegistry前置处理器
 invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors, registry);

```

Debug图片



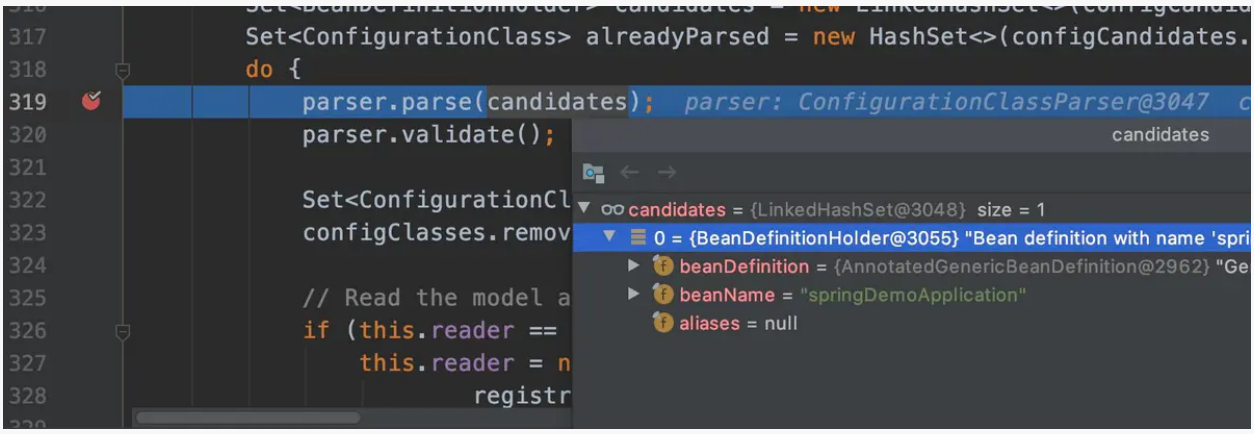
生成ConfigurationClassPostProcessor

我们通过Debug可以看出，`currentRegistryProcessors`中放的是`ConfigurationClassPostProcessor`的Bean对象，接着就调用了`invokeBeanDefinitionRegistryPostProcessors`方法并传入`ConfigurationClassPostProcessor`。

> 1. 接着进入`org.springframework.context.annotation.ConfigurationClassPostProcessor#processConfigBeanDefinitions`方法

```java
    public void processConfigBeanDefinitions(BeanDefinitionRegistry registry) {

    ...
    // 从BeanDefinition中找出带有Configuration.class的，自己Debug可以进入if的两个方法中查看
        for (String beanName : candidateNames) {
            BeanDefinition beanDef = registry.getBeanDefinition(beanName);
            if (beanDef.getAttribute(ConfigurationClassUtils.CONFIGURATION_CLASS_ATTRIBUTE) != null) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Bean definition has already been processed as a configuration class: " + beanDef);
                }
            }
            else if (ConfigurationClassUtils.checkConfigurationClassCandidate(beanDef, this.metadataReaderFactory)) {
                configCandidates.add(new BeanDefinitionHolder(beanDef, beanName));
            }
        }
        ...
        // 传入候选人
    parser.parse(candidates);

        ...
    }
```

Debug图片



解析

通过图片，可以看出，`candidates`只有一个，那就是启动类SpringDemoApplication（测试项目的启动类）.

> 1. 进入`org.springframework.context.annotation.ConfigurationClassParser#parse()`方法，开始解析启动类。

```
    public void parse(Set<BeanDefinitionHolder> configCandidates) {
        for (BeanDefinitionHolder holder : configCandidates) {
            BeanDefinition bd = holder.getBeanDefinition();
            try {
                if (bd instanceof AnnotatedBeanDefinition) {
            // 进入此方法了。 这我是Debug进去的，没有探究启动类在被解析成BeanDefinition的时候，被解析成
            // AnnotatedBeanDefinition， 有兴趣的同学自己Debug追究一下
                    parse(((AnnotatedBeanDefinition) bd).getMetadata(), holder.getBeanName());
                }
            }
```

```
    ...
    // 这个有用，所以我留在这了，关键点。
        this.deferredImportSelectorHandler.process();
    }
```

> 1. 接下来进入 `org.springframework.context.annotation.ConfigurationClassParser#processConfigurationClass` 方法。

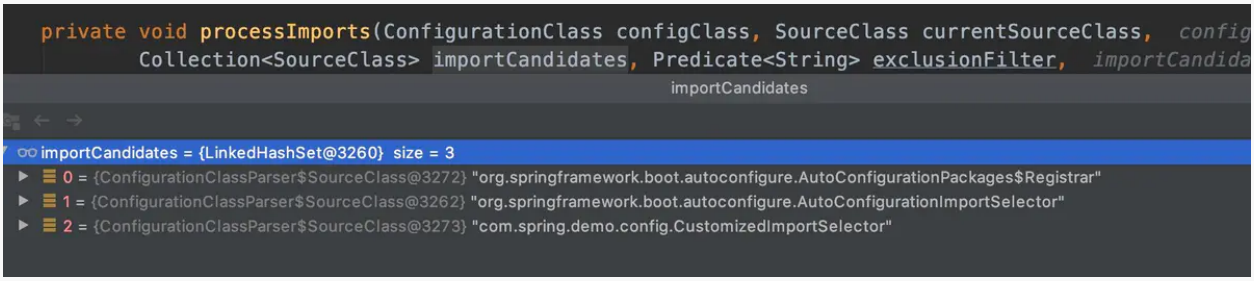
```
protected void processConfigurationClass(ConfigurationClass configClass, Predicate<String> filter) throws IOException {
        ...

    // Recursively process the configuration class and its superclass hierarchy.
        SourceClass sourceClass = asSourceClass(configClass, filter);
        do {
      // 我们可以通过注解，看出这个是循环调用,找到configClass 自己的configuration注解或继承的注解中包含configuration的
      // 不用多纠结，我们直接找到
            sourceClass = doProcessConfigurationClass(configClass, sourceClass, filter);
        }
        while (sourceClass != null);
    }
```

> 1. 进入 `org.springframework.context.annotation.ConfigurationClassParser#doProcessConfigurationClass` 方法，直接分析代码.

```java
protected final SourceClass doProcessConfigurationClass(
            ConfigurationClass configClass, SourceClass sourceClass, Predicate<String> filter)
            throws IOException {

        if (configClass.getMetadata().isAnnotated(Component.class.getName())) {
            // Recursively process any member (nested) classes first
      // 在这递归的，会回到上一步代码中
            processMemberClasses(configClass, sourceClass, filter);
        }


        // Process any @Import annotations
    // 别的代码不看，就这个名字，我们也知道这个类是干嘛的了吧！
        processImports(configClass, sourceClass, getImports(sourceClass), filter, true);
    }
```

`getImports(sourceClass)`这个方法是递归调用，找到注解Import中的值。 放个Debug图给大家瞅一下。`CustomizedImportSelector`是我自己测试的，

```java
/**
 * <br>自定义importSelector</br>
 *
 * @author fattyca1
 */
public class CustomizedImportSelector implements ImportSelector {

    @Override
    public String[] selectImports(AnnotationMetadata importingClassMetadata) {

        return new String[] {"com.spring.demo.config.MyConfig"};
    }
}
```



getImports(sourceClass)值

> 1. 进入`org.springframework.context.annotation.ConfigurationClassParser#processImports`方法，核心来了，就是问题的关键，到底是怎么使用*SpringSPI*的

```java
private void processImports(ConfigurationClass configClass, SourceClass currentSourceClass,
```

```
            Collection<SourceClass> importCandidates, Predicate<String> exclusionFilter,
            boolean checkForCircularImports) {
        ...
        if (checkForCircularImports && isChainedImportOnStack(configClass)) {
            this.problemReporter.error(new CircularImportProblem(configClass, this.importStack));
        }
        else {
            this.importStack.push(configClass);
            try {
                for (SourceClass candidate : importCandidates) {
                    if (candidate.isAssignable(ImportSelector.class)) {
                        // Candidate class is an ImportSelector -> delegate to it to determine imports
                        Class<?> candidateClass = candidate.loadClass();
                        ImportSelector selector = ParserStrategyUtils.instantiateClass(candidateClass, ImportSelector.class,
                                                this.environment, this.resourceLoader, this.registry);
                        Predicate<String> selectorFilter = selector.getExclusionFilter();

                        if (selector instanceof DeferredImportSelector) {
                // 因为AutoConfigurationImportSelector继承了DeferredImportSelector，所以会进入这个方法，放到
                // 列表里处理，直接放到一个List中。

 this.deferredImportSelectorHandler.handle(configClass, (DeferredImportSelector) selector);
                        }
                    }
```

```
        }
        ...
            }
        }
    }
```

> 1. `org.springframework.context.annotation.ConfigurationClassParser.DeferredImportSelectorHandler#handle`方法

```
public void handle(ConfigurationClass configClass, DeferredImportSelector importSelector) {
            DeferredImportSelectorHolder holder = new DeferredImportSelectorHolder(configClass, importSelector);
            if (this.deferredImportSelectors == null) {
                DeferredImportSelectorGroupingHandler handler = new DeferredImportSelectorGroupingHandler();
                handler.register(holder);
                handler.processGroupImports();
            }
            else {
        // deferredImportSelectors 是一个ArrayList，在类部类中被初始化，所以走的此方法
                this.deferredImportSelectors.add(holder);
            }
        }
```

自此，我们分析完`AutoConfigurationImportSelector`在第一遍解析完后，被放在哪，那接下来就是如何解析了。激动人心的时刻来了。那就是在`ConfigurationClassParser#parse()`中执行的代码了`this.deferredImportSelectorHandler.process();`

> 1. `org.springframework.context.annotation.ConfigurationClassParser.DeferredImportSelectorHandler#process`代码

```java
    public void process() {
            List<DeferredImportSelectorHolder> deferredImports = this.deferredImportSelectors;
            this.deferredImportSelectors = null;
            try {
                if (deferredImports != null) {
                    DeferredImportSelectorGroupingHandler handler = new DeferredImportSelectorGroupingHandler();
 deferredImports.sort(DEFERRED_IMPORT_COMPARATOR);
          // 把list中的DeferredImportSelectorHolder注册到DeferredImportSelectorGroupingHandler
          // 这个register方法会对DeferredImportSelectorHolder进行封装
 deferredImports.forEach(handler::register);
                    handler.processGroupImports();
                }
            }
            finally {
                this.deferredImportSelectors = new ArrayList<>();
            }
        }
```

> 1. `org.springframework.context.annotation.ConfigurationClassParser.DeferredImportSelectorGroupingHandler#register`代码

```java
        public void register(DeferredImportSelectorHolder deferredImport) {
      // AutoConfigurationImportSelector返回的是AutoConfigurationGroup.class，代码中已写死
            Class<? extends Group> group = deferredImport.getImportSelector().getImportGroup();
      // 封装成 DeferredImportSelector.Group 对象，并放到了groupings中，groupings是LinkedHashMap
      // Group对象是用AutoConfigurationGroup.class生成
            DeferredImportSelectorGrouping grouping = this.groupings.computeIfAbsent(
                    (group != null ? group : deferredImport),
                    key -> new DeferredImportSelectorGrouping(createGroup(group)));
            grouping.add(deferredImport);

 this.configurationClasses.put(deferredImport.getConfigurationClass().getMetadata(),
                    deferredImport.getConfigurationClass());
        }
```

1. `org.springframework.context.annotation.ConfigurationClassParser.DeferredImportSelectorGroupingHandler#processGroupImports`方法，*SpringSPI*的调用点

```java
public void processGroupImports() {
            for (DeferredImportSelectorGrouping grouping : this.groupings.values()) {
                Predicate<String> exclusionFilter = grouping.getCandidateFilter();
        //遍历放入到grouping中的group，并执行getImports()方法，此方法就是SPI调用点！！！！
                grouping.getImports().forEach(entry -> {
```

```
                    ConfigurationClass configurationClass = this.configurationClasses.get(entry.getMetadata());
                    try {
                        processImports(configurationClass, asSourceClass(configurationClass, exclusionFilter),
                                Collections.singleton(asSourceClass(entry.getImportClassName(), exclusionFilter)),
                                exclusionFilter, false);
                    }
                    catch (BeanDefinitionStoreException ex) {
                        throw ex;
                    }
                    catch (Throwable ex) {
                        throw new BeanDefinitionStoreException(
                                "Failed to process import candidates for configuration class [" +
                                configurationClass.getMetadata().getClassName() + "]", ex);
                    }
                });
            }
        }
```

> 1. `org.springframework.context.annotation.ConfigurationClassParser.DeferredImportSelectorGrouping#getImports`

```java
        public Iterable<Group.Entry> getImports() {
            for (DeferredImportSelectorHolder deferredImport : this.deferredImports) {
        // 调用group的process方法，也就是上面分析，AutoConfigurationGroup.class类的process方法

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

1. `org.springframework.boot.autoconfigure.AutoConfigurationImportSelector.AutoConfigurationGroup#process`方法

```
        public void process(AnnotationMetadata annotationMetadata, DeferredImportSelector deferredImportSelector) {
            Assert.state(deferredImportSelector instanceof AutoConfigurationImportSelector,
                    () -> String.format("Only %s implementations are supported, got %s",
 AutoConfigurationImportSelector.class.getSimpleName(),
 deferredImportSelector.getClass().getName()));
      // getAutoConfigurationEntry 熟悉的方法，SPI的具体执行逻辑
            AutoConfigurationEntry autoConfigurationEntry = ((AutoConfigurationImportSelector) deferredImportSelector)
.getAutoConfigurationEntry(annotationMetadata);
 this.autoConfigurationEntries.add(autoConfigurationEntry);
            for (String importClassName : autoConfigurationEntry.getConfigurations()) {
                this.entries.putIfAbsent(importClassName, annotationMetadata);
            }
        }
```

自此，我们的代码分析结束，发现`AutoConfigurationImportSelector.class`在SpringBoot启动中，并不是调用的selectImports方法，而是直接调用的`getAutoConfigurationEntry`方法

4. 总结

SpringBoot在启动中，`AutoConfigurationImportSelector`在被加载中，调用的不是selectImports方法， 而是直接被调用了getAutoConfigurationEntry方法。 骚年，你可长点心吧！