Spring boot自动装配之@ComponentScan详解

1.@ComponentScan注解作用
@ComponentScan用于类或接口上主要是指定扫描路径，spring会把指定路径下带有指定注解的类自动装配到bean容器里。会被自动装配的注解包括@Controller、@Service、@Component、@Repository等等。其作用等同于<context:component-scan base-package="com.maple.learn" />配置

2.@ComponentScan使用
常用属性如下：

basePackages、value：指定扫描路径，如果为空则以@ComponentScan注解的类所在的包为基本的扫描路径
basePackageClasses：指定具体扫描的类
includeFilters：指定满足Filter条件的类
excludeFilters：指定排除Filter条件的类
includeFilters和excludeFilters 的FilterType可选：ANNOTATION=注解类型 默认、ASSIGNABLE_TYPE(指定固定类)、ASPECTJ(ASPECTJ类型)、REGEX(正则表达式)、CUSTOM(自定义类型)，自定义的Filter需要实现TypeFilter接口
@ComponentScan的常见的配置如下：

```
@ComponentScan(value="com.maple.learn",
   excludeFilters = {@ComponentScan.Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class)},
   includeFilters = {@ComponentScan.Filter(type=FilterType.ANNOTATION,classes={Controller.class})}
        )
public class SampleClass{
   ……
}
```

3.spring boot处理@ComponentScan源码分析

spring创建bean对象的基本流程是先创建对应的BeanDefinition对象，然后在基于BeanDefinition对象来创建bean对象，spring boot也是如此，只不过通过注解创建BeanDefinition对象的时机和解析方式不同而已。spring boot是通过ConfigurationClassPostProcessor这个BeanFactoryPostProcessor类来处理。

本演示的demo涉及到5个演示类：SpringbootCodeMain为启动类带有@SpringBootApplication注解，SampleAction类带有@RestController注解，ServcieConfigure类带有@Configuration注解且有通过@Bean注解来创建PeopleService的方法，PeopleService无任何注解，本文的最后会贴出所有代码。先从启动类为入口，spring boot启动类如下：

```
@SpringBootApplication
public class SpringbootCodeMain {
    public static void main(String[] args) {
        SpringApplication.run(SpringbootCodeMain.class, args);
    }
}
```

从SpringApplication.run(SpringbootCodeMain.class, args)一路断点到核心方法SpringApplication.ConfigurableApplicationContext run(String... args)方法

```
/**
     * Run the Spring application, creating and refreshing a new
     * {@link ApplicationContext}.
```

```
4      * @param args the application arguments (usually passed
from a Java main method)
5      * @return a running {@link ApplicationContext}
6      */
7     public ConfigurableApplicationContext run(String... args)
{
8         long startTime = System.nanoTime();
9         DefaultBootstrapContext bootstrapContext =
createBootstrapContext();
10        ConfigurableApplicationContext context = null;
11        configureHeadlessProperty();
12        SpringApplicationRunListeners listeners =
getRunListeners(args);
13        listeners.starting(bootstrapContext,
this.mainApplicationClass);
14        try {
15            ApplicationArguments applicationArguments = new
DefaultApplicationArguments(args);
16            ConfigurableEnvironment environment =
prepareEnvironment(listeners, bootstrapContext,
applicationArguments);
17            configureIgnoreBeanInfo(environment);
18            Banner printedBanner = printBanner(environment);
19            context = createApplicationContext();
20
context.setApplicationStartup(this.applicationStartup);
21            prepareContext(bootstrapContext, context,
environment, listeners, applicationArguments, printedBanner);
22            refreshContext(context);
23            afterRefresh(context, applicationArguments);
24            Duration timeTakenToStartup =
Duration.ofNanos(System.nanoTime() - startTime);
25            if (this.logStartupInfo) {
26                new
StartupInfoLogger(this.mainApplicationClass).logStarted(getAp
plicationLog(), timeTakenToStartup);
27            }
28            listeners.started(context, timeTakenToStartup);
```

```
            callRunners(context, applicationArguments);
        }
        catch (Throwable ex) {
            handleRunFailure(context, ex, listeners);
            throw new IllegalStateException(ex);
        }
        try {
            Duration timeTakenToReady = Duration.ofNanos(System.nanoTime() - startTime);
            listeners.ready(context, timeTakenToReady);
        }
        catch (Throwable ex) {
            handleRunFailure(context, ex, null);
            throw new IllegalStateException(ex);
        }
        return context;
    }
```

其中

```
context = this.createApplicationContext();
```

重点方法一，本法实现的重点功能：根据

- 1、本demo是web工程，springboot通过反射创建上下文context:AnnotationConfigServletWebServerApplicationContext 类
- 2、在构建context的无参构造方法中构建成员变量reader=new AnnotatedBeanDefinitionReader(this)，在AnnotatedBeanDefinitionReader的无参构造方法中会beanFactory对象，并向beanFactory中注册5个BeanDefinition对象，重点关注ConfigurationClassPostProcessor。

> 根据*webApplicationType*类型进行类型推断容器：*springboot*默认启动是*Servlet*

整个过程如下：

```
protected ConfigurableApplicationContext createApplicationContext() {
    return this.applicationContextFactory.create(this.webApplicationType);
}
```

创建this.applicationContextFactory 如下：

```
private ApplicationContextFactory applicationContextFactory = ApplicationContextFactory.DEFAULT;

```

```
ApplicationContextFactory DEFAULT = (webApplicationType) -> {
        try {
            switch (webApplicationType) {
            case SERVLET:
                return new AnnotationConfigServletWebServerApplicationContext();
            case REACTIVE:
                return new AnnotationConfigReactiveWebServerApplicationContext();
            default:
                return new AnnotationConfigApplicationContext();
            }
        }
        catch (Exception ex) {
            throw new IllegalStateException("Unable create a default ApplicationContext instance, "
                    + "you may need a custom ApplicationContextFactory", ex);
        }
    };
```

而类型是：Servlet自然就是：

```
return new
AnnotationConfigServletWebServerApplicationContext();
```

这个这个构造函数中初始化两个读取对象

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

AnnotatedBeanDefinitionReader 和 ClassPathBeanDefinitionScanner

重点方法二，

本法实现的重点功能：
1、本方法会构建启动类SpringbootCodeMain对应的BeanDefinition对象，并注册到beanFactory中，此时的context对象可见下图

> *this.prepareContext(context, environment, listeners, applicationArguments, printedBanner);*

```
context = {AnnotationConfigServletWebServerApplicationContext@2277} "org.springframework.boot.web.servlet.context.AnnotationConfigSe
  reader = {AnnotatedBeanDefinitionReader@2289}
  scanner = {ClassPathBeanDefinitionScanner@2939}
  annotatedClasses = {LinkedHashSet@2940}  size = 0
  basePackages = null
  webServer = null
  servletConfig = null
  serverNamespace = null
  servletContext = null
  themeSource = null
  beanFactory = {DefaultListableBeanFactory@2622} "org.springframework.beans.factory.support.DefaultListableBeanFactory@7b205dbd:
    serializationId = null
    allowBeanDefinitionOverriding = false
    allowEagerClassLoading = true
    dependencyComparator = {AnnotationAwareOrderComparator@2964}
    autowireCandidateResolver = {ContextAnnotationAutowireCandidateResolver@2965}
    resolvableDependencies = {ConcurrentHashMap@2966}  size = 0
    beanDefinitionMap = {ConcurrentHashMap@2967}  size = 6
      0 = {ConcurrentHashMap$MapEntry@3020} "org.springframework.context.annotation.internalConfigurationAnnotationProcessor"
      1 = {ConcurrentHashMap$MapEntry@3021} "org.springframework.context.event.internalEventListenerFactory" -> "Root bean: clas
      2 = {ConcurrentHashMap$MapEntry@3022} "org.springframework.context.event.internalEventListenerProcessor" -> "Root bean: c
      3 = {ConcurrentHashMap$MapEntry@3023} "org.springframework.context.annotation.internalAutowiredAnnotationProcessor" ->
      4 = {ConcurrentHashMap$MapEntry@3024} "org.springframework.context.annotation.internalCommonAnnotationProcessor" -> "I
      5 = {ConcurrentHashMap$MapEntry@3025} "springbootCodeMain" -> "Generic bean: class [com.maple.learn.springbootcode.Sp
```

重点方法三

```
this.refreshContext(context);
```

该方法实际调用applicationContext的refresh方法，同过源代码发现最终是由ConfigurationClassParser的解析类来处理，继续查看ConfigurationClassParser.doProcessConfigurationClass

```java
// Process any @ComponentScan annotations
Set<AnnotationAttributes> componentScans = AnnotationConfigUtils.attributesForRepeatable( componentScans:  size =
        sourceClass.getMetadata(), ComponentScans.class, ComponentScan.class);
if (!componentScans.isEmpty() &&
        !this.conditionEvaluator.shouldSkip(sourceClass.getMetadata(), ConfigurationPhase.REGISTER_BEAN)) {
    for (AnnotationAttributes componentScan : componentScans) {  componentScans:  size = 0
        // The config class is annotated with @ComponentScan -> perform the scan immediately
        Set<BeanDefinitionHolder> scannedBeanDefinitions =
                this.componentScanParser.parse(componentScan, sourceClass.getMetadata().getClassName());
        // Check the set of scanned definitions for any further config classes and parse recursively if needed
        for (BeanDefinitionHolder holder : scannedBeanDefinitions) {
            BeanDefinition bdCand = holder.getBeanDefinition().getOriginatingBeanDefinition();
            if (bdCand == null) {
                bdCand = holder.getBeanDefinition();
            }
            if (ConfigurationClassUtils.checkConfigurationClassCandidate(bdCand, this.metadataReaderFactory)) {
                parse(bdCand.getBeanClassName(), holder.getBeanName());
            }
        }
    }
}
```


整个过程：

1：refreshContext(context);

2：进入spring的生命周期 refresh()

3：进入PostProcessorRegistrationDelegate.invokeBeanFactoryPostProcessors(beanFactory);

```
protected void invokeBeanFactoryPostProcessors(ConfigurableListableBeanFactory beanFactory) {
PostProcessorRegistrationDelegate.invokeBeanFactoryPostProcessors(beanFactory, getBeanFactoryPostProcessors());

        // Detect a LoadTimeWeaver and prepare for weaving, if found in the meantime
        // (e.g. through an @Bean method registered by ConfigurationClassPostProcessor)
        if (!NativeDetector.inNativeImage() && beanFactory.getTempClassLoader() == null && beanFactory.containsBean(LOAD_TIME_WEAVER_BEAN_NAME)) {
            beanFactory.addBeanPostProcessor(new LoadTimeWeaverAwareProcessor(beanFactory));
            beanFactory.setTempClassLoader(new ContextTypeMatchClassLoader(beanFactory.getBeanClassLoader()));
        }
    }
```

4：ConfigurationClassPostProcessor的postProcessBeanDefinitionRegistry

```
/**
     * Derive further bean definitions from the configuration classes in the registry.
     */
    @Override
    public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) {
        int registryId = System.identityHashCode(registry);
        if (this.registriesPostProcessed.contains(registryId)) {
            throw new IllegalStateException(
```

```
                    "postProcessBeanDefinitionRegistry already called on this post-processor against " + registry);
        }
        if (this.factoriesPostProcessed.contains(registryId)) {
            throw new IllegalStateException(
                    "postProcessBeanFactory already called on this post-processor against " + registry);
        }
        this.registriesPostProcessed.add(registryId);

        processConfigBeanDefinitions(registry);
    }
```

```
/**
     * Build and validate a configuration model based on the registry of
     * {@link Configuration} classes.
     */
    public void processConfigBeanDefinitions(BeanDefinitionRegistry registry) {
        List<BeanDefinitionHolder> configCandidates = new ArrayList<>();
        String[] candidateNames = registry.getBeanDefinitionNames();

        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));
            }
        }

        // Return immediately if no @Configuration classes were found
        if (configCandidates.isEmpty()) {
            return;
        }

        // Sort by previously determined @Order value, if applicable
        configCandidates.sort((bd1, bd2) -> {
            int i1 = ConfigurationClassUtils.getOrder(bd1.getBeanDefinition());
            int i2 = ConfigurationClassUtils.getOrder(bd2.getBeanDefinition());
            return Integer.compare(i1, i2);
        });

        // Detect any custom bean name generation strategy supplied through the enclosing application context
        SingletonBeanRegistry sbr = null;
        if (registry instanceof SingletonBeanRegistry) {
            sbr = (SingletonBeanRegistry) registry;
            if (!this.localBeanNameGeneratorSet) {
                BeanNameGenerator generator = (BeanNameGenerator) sbr.getSingleton(
                        AnnotationConfigUtils.CONFIGURATION_BEAN_NAME_GENERATOR);
                if (generator != null) {
                    this.componentScanBeanNameGenerator = generator;
```

```
                    this.importBeanNameGenerator = generator;
                }
            }
        }

        if (this.environment == null) {
            this.environment = new StandardEnvironment();
        }

        // Parse each @Configuration class
        ConfigurationClassParser parser = new ConfigurationClassParser(
                this.metadataReaderFactory, this.problemReporter, this.environment,
                this.resourceLoader, this.componentScanBeanNameGenerator, registry);

        Set<BeanDefinitionHolder> candidates = new LinkedHashSet<>(configCandidates);
        Set<ConfigurationClass> alreadyParsed = new HashSet<>(configCandidates.size());
        do {
            StartupStep processConfig = this.applicationStartup.start("spring.context.config-classes.parse");
            //  核心方法
            parser.parse(candidates);
            parser.validate();

            Set<ConfigurationClass> configClasses = new LinkedHashSet<>(parser.getConfigurationClasses());
            configClasses.removeAll(alreadyParsed);

            // Read the model and create bean definitions based on its content
            if (this.reader == null) {
```

```
69                 this.reader = new
ConfigurationClassBeanDefinitionReader(
70                         registry, this.sourceExtractor,
this.resourceLoader, this.environment,
71                         this.importBeanNameGenerator,
parser.getImportRegistry());
72             }
73             this.reader.loadBeanDefinitions(configClasses);
74             alreadyParsed.addAll(configClasses);
75             processConfig.tag("classCount", () ->
String.valueOf(configClasses.size())).end();
76
77             candidates.clear();
78             if (registry.getBeanDefinitionCount() >
candidateNames.length) {
79                 String[] newCandidateNames =
registry.getBeanDefinitionNames();
80                 Set<String> oldCandidateNames = new
HashSet<>(Arrays.asList(candidateNames));
81                 Set<String> alreadyParsedClasses = new
HashSet<>();
82                 for (ConfigurationClass configurationClass
: alreadyParsed) {
83
alreadyParsedClasses.add(configurationClass.getMetadata().g
etClassName());
84                 }
85                 for (String candidateName :
newCandidateNames) {
86                     if
(!oldCandidateNames.contains(candidateName)) {
87                         BeanDefinition bd =
registry.getBeanDefinition(candidateName);
88                         if
(ConfigurationClassUtils.checkConfigurationClassCandidate(b
d, this.metadataReaderFactory) &&
89
!alreadyParsedClasses.contains(bd.getBeanClassName())) {
```

```
                                candidates.add(new BeanDefinitionHolder(bd, candidateName));
                            }
                        }
                    }
                    candidateNames = newCandidateNames;
                }
            }
            while (!candidates.isEmpty());

            // Register the ImportRegistry as a bean in order to support ImportAware @Configuration classes
            if (sbr != null && !sbr.containsSingleton(IMPORT_REGISTRY_BEAN_NAME)) {
                sbr.registerSingleton(IMPORT_REGISTRY_BEAN_NAME, parser.getImportRegistry());
            }

            if (this.metadataReaderFactory instanceof CachingMetadataReaderFactory) {
                // Clear cache in externally provided MetadataReaderFactory; this is a no-op
                // for a shared cache since it'll be cleared by the ApplicationContext.
                ((CachingMetadataReaderFactory) this.metadataReaderFactory).clearCache();
            }
        }
```

6：核心代码

```
// 这里是核心代码
parser.parse(candidates);
```

```

public void parse(Set<BeanDefinitionHolder> configCandidates) {
```

```java
        for (BeanDefinitionHolder holder : configCandidates) {
            BeanDefinition bd = holder.getBeanDefinition();
            try {
                if (bd instanceof AnnotatedBeanDefinition) {
                    parse(((AnnotatedBeanDefinition) bd).getMetadata(), holder.getBeanName());
                }
                else if (bd instanceof AbstractBeanDefinition && ((AbstractBeanDefinition) bd).hasBeanClass()) {
                    parse(((AbstractBeanDefinition) bd).getBeanClass(), holder.getBeanName());
                }
                else {
                    parse(bd.getBeanClassName(), holder.getBeanName());
                }
            }
            catch (BeanDefinitionStoreException ex) {
                throw ex;
            }
            catch (Throwable ex) {
                throw new BeanDefinitionStoreException(
                        "Failed to parse configuration class [" + bd.getBeanClassName() + "]", ex);
            }
        }

        this.deferredImportSelectorHandler.process();
    }
```

```java
protected final void parse(AnnotationMetadata metadata, String beanName) throws IOException {
        processConfigurationClass(new ConfigurationClass(metadata, beanName), DEFAULT_EXCLUSION_FILTER);
    }
```

7：调用的是：ConfigurationClassParser.doProcessConfigurationClass

```
protected void processConfigurationClass(ConfigurationClass configClass, Predicate<String> filter) throws IOException {
        if (this.conditionEvaluator.shouldSkip(configClass.getMetadata(), ConfigurationPhase.PARSE_CONFIGURATION)) {
            return;
        }

        ConfigurationClass existingClass = this.configurationClasses.get(configClass);
        if (existingClass != null) {
            if (configClass.isImported()) {
                if (existingClass.isImported()) {
                    existingClass.mergeImportedBy(configClass);
                }
                // Otherwise ignore new imported config class; existing non-imported class overrides it.
                return;
            }
            else {
                // Explicit bean definition found, probably replacing an import.
                // Let's remove the old one and go with the new one.
                this.configurationClasses.remove(configClass);
                this.knownSuperclasses.values().removeIf(configClass::equals);
            }
        }

```

```
        // Recursively process the configuration class and its superclass hierarchy.
        SourceClass sourceClass = asSourceClass(configClass, filter);
        do {
            // 核心代码如下:
            sourceClass = doProcessConfigurationClass(configClass, sourceClass, filter);
        }
        while (sourceClass != null);

        this.configurationClasses.put(configClass, configClass);
    }
```

```
// 核心代码如下:
            sourceClass = doProcessConfigurationClass(configClass, sourceClass, filter);
```

```
/**
     * Apply processing and build a complete {@link ConfigurationClass} by reading the
     * annotations, members and methods from the source class. This method can be called
     * multiple times as relevant sources are discovered.
     * @param configClass the configuration class being build
     * @param sourceClass a source class
     * @return the superclass, or {@code null} if none found or previously processed
     */
    @Nullable
    protected final SourceClass doProcessConfigurationClass(
            ConfigurationClass configClass, SourceClass sourceClass, Predicate<String> filter)
            throws IOException {

```

```
14          if (configClass.getMetadata().isAnnotated(Component.class.getName())) {
15              // Recursively process any member (nested) classes first
16              processMemberClasses(configClass, sourceClass, filter);
17          }
18
19          // Process any @PropertySource annotations
20          for (AnnotationAttributes propertySource : AnnotationConfigUtils.attributesForRepeatable(
21                  sourceClass.getMetadata(), PropertySources.class,
22                  org.springframework.context.annotation.PropertySource.class)) {
23              if (this.environment instanceof ConfigurableEnvironment) {
24                  processPropertySource(propertySource);
25              }
26              else {
27                  logger.info("Ignoring @PropertySource annotation on [" + sourceClass.getMetadata().getClassName() +
28                          "]. Reason: Environment must implement ConfigurableEnvironment");
29              }
30          }
31
32          // Process any @ComponentScan annotations
33          Set<AnnotationAttributes> componentScans = AnnotationConfigUtils.attributesForRepeatable(
34                  sourceClass.getMetadata(), ComponentScans.class, ComponentScan.class);
35          if (!componentScans.isEmpty() &&
36                  !this.conditionEvaluator.shouldSkip(sourceClass.getMetadata(), ConfigurationPhase.REGISTER_BEAN)) {
```

```
            for (AnnotationAttributes componentScan : componentScans) {
                // The config class is annotated with @ComponentScan -> perform the scan immediately
                Set<BeanDefinitionHolder> scannedBeanDefinitions =
this.componentScanParser.parse(componentScan, sourceClass.getMetadata().getClassName());
                // Check the set of scanned definitions for any further config classes and parse recursively if needed
                for (BeanDefinitionHolder holder : scannedBeanDefinitions) {
                    BeanDefinition bdCand = holder.getBeanDefinition().getOriginatingBeanDefinition();
                    if (bdCand == null) {
                        bdCand = holder.getBeanDefinition();
                    }
                    if (ConfigurationClassUtils.checkConfigurationClassCandidate(bdCand, this.metadataReaderFactory)) {
                        parse(bdCand.getBeanClassName(), holder.getBeanName());
                    }
                }
            }
        }

        // Process any @Import annotations
        processImports(configClass, sourceClass, getImports(sourceClass), filter, true);

        // Process any @ImportResource annotations
        AnnotationAttributes importResource =
AnnotationConfigUtils.attributesFor(sourceClass.getMetadata(), ImportResource.class);
        if (importResource != null) {
```

```
            String[] resources = importResource.getStringArray("locations");
            Class<? extends BeanDefinitionReader> readerClass = importResource.getClass("reader");
            for (String resource : resources) {
                String resolvedResource = this.environment.resolveRequiredPlaceholders(resource);
                configClass.addImportedResource(resolvedResource, readerClass);
            }
        }

        // Process individual @Bean methods
        Set<MethodMetadata> beanMethods = retrieveBeanMethodMetadata(sourceClass);
        for (MethodMetadata methodMetadata : beanMethods) {
            configClass.addBeanMethod(new BeanMethod(methodMetadata, configClass));
        }

        // Process default methods on interfaces
        processInterfaces(configClass, sourceClass);

        // Process superclass, if any
        if (sourceClass.getMetadata().hasSuperClass()) {
            String superclass = sourceClass.getMetadata().getSuperClassName();
            if (superclass != null && !superclass.startsWith("java") &&
                !this.knownSuperclasses.containsKey(superclass)) {
                this.knownSuperclasses.put(superclass, configClass);
                // Superclass found, return its annotation metadata and recurse
                return sourceClass.getSuperClass();
            }
```

```
        }

        // No superclass -> processing is complete
        return null;
    }

```

核心代码

```
Set<BeanDefinitionHolder> scannedBeanDefinitions =
this.componentScanParser.parse(componentScan, sourceClass.getMetadata().getClassName());
```

```
public Set<BeanDefinitionHolder> parse(AnnotationAttributes componentScan, String declaringClass) {
        ClassPathBeanDefinitionScanner scanner = new ClassPathBeanDefinitionScanner(this.registry,
componentScan.getBoolean("useDefaultFilters"), this.environment, this.resourceLoader);

        Class<? extends BeanNameGenerator> generatorClass = componentScan.getClass("nameGenerator");
        boolean useInheritedGenerator = (BeanNameGenerator.class == generatorClass);
        scanner.setBeanNameGenerator(useInheritedGenerator ? this.beanNameGenerator :
                BeanUtils.instantiateClass(generatorClass));

        ScopedProxyMode scopedProxyMode = componentScan.getEnum("scopedProxy");
        if (scopedProxyMode != ScopedProxyMode.DEFAULT) {
            scanner.setScopedProxyMode(scopedProxyMode);
        }
        else {
            Class<? extends ScopeMetadataResolver> resolverClass = componentScan.getClass("scopeResolver");
```

```
scanner.setScopeMetadataResolver(BeanUtils.instantiateClass(resolverClass));
        }

scanner.setResourcePattern(componentScan.getString("resourcePattern"));

        for (AnnotationAttributes includeFilterAttributes : componentScan.getAnnotationArray("includeFilters")) {
            List<TypeFilter> typeFilters = TypeFilterUtils.createTypeFiltersFor(includeFilterAttributes, this.environment,
                    this.resourceLoader, this.registry);
            for (TypeFilter typeFilter : typeFilters) {
                scanner.addIncludeFilter(typeFilter);
            }
        }
        for (AnnotationAttributes excludeFilterAttributes : componentScan.getAnnotationArray("excludeFilters")) {
            List<TypeFilter> typeFilters = TypeFilterUtils.createTypeFiltersFor(excludeFilterAttributes, this.environment,
                this.resourceLoader, this.registry);
            for (TypeFilter typeFilter : typeFilters) {
                scanner.addExcludeFilter(typeFilter);
            }
        }

        boolean lazyInit = componentScan.getBoolean("lazyInit");
        if (lazyInit) {
            scanner.getBeanDefinitionDefaults().setLazyInit(true);
        }

        Set<String> basePackages = new LinkedHashSet<>();
```

```
        String[] basePackagesArray = componentScan.getStringArray("basePackages");
        for (String pkg : basePackagesArray) {
            String[] tokenized = StringUtils.tokenizeToStringArray(this.environment.resolvePlaceholders(pkg),
                ConfigurableApplicationContext.CONFIG_LOCATION_DELIMITERS);
            Collections.addAll(basePackages, tokenized);
        }
        for (Class<?> clazz : componentScan.getClassArray("basePackageClasses")) {
            basePackages.add(ClassUtils.getPackageName(clazz));
        }
        if (basePackages.isEmpty()) {
            basePackages.add(ClassUtils.getPackageName(declaringClass));
        }

        scanner.addExcludeFilter(new AbstractTypeHierarchyTraversingFilter(false, false) {
            @Override
            protected boolean matchClassName(String className) {
                return declaringClass.equals(className);
            }
        });
    //核心方法
        return scanner.doScan(StringUtils.toStringArray(basePackages));
    }

```

10、最终做事情的是:ClassPathBeanDefinitionScanner类的doScan方法如下：

```
rotected Set<BeanDefinitionHolder> doScan(String... basePackages) {
    Assert.notEmpty(basePackages, "At least one base package must be specified");
    Set<BeanDefinitionHolder> beanDefinitions = new LinkedHashSet<>();
    for (String basePackage : basePackages) {
        Set<BeanDefinition> candidates = findCandidateComponents(basePackage);
        for (BeanDefinition candidate : candidates) {
            ScopeMetadata scopeMetadata = this.scopeMetadataResolver.resolveScopeMetadata(candidate);
            candidate.setScope(scopeMetadata.getScopeName());
            String beanName = this.beanNameGenerator.generateBeanName(candidate, this.registry);
            if (candidate instanceof AbstractBeanDefinition) {
                postProcessBeanDefinition((AbstractBeanDefinition) candidate, beanName);
            }
            if (candidate instanceof AnnotatedBeanDefinition) {
                AnnotationConfigUtils.processCommonDefinitionAnnotations((AnnotatedBeanDefinition) candidate);
            }
            if (checkCandidate(beanName, candidate)) {
                BeanDefinitionHolder definitionHolder = new BeanDefinitionHolder(candidate, beanName);
                definitionHolder =
                AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata, definitionHolder, this.registry);
                beanDefinitions.add(definitionHolder);
                registerBeanDefinition(definitionHolder, this.registry);
```

```
                }
            }
        }
        return beanDefinitions;
    }

```

```

```

原来在这里对@ComponentScan注解做了判断，上面一段代码做了核心的几件事：

1、扫描@ComponentScan注解包下面的所有的可自动装备类，生成BeanDefinition对象，并注册beanFactory对象中

2、通过DeferredImportSelectorHandler处理@EnableAutoConfiguration注解，后续会有专文介绍

3、将带有@Configuration 注解的类解析成ConfigurationClass对象并缓存，后面创建@Bean注解的Bean对象所对应的BeanDefinition时会用到

到此为止serviceConfigure和sampleAction对应的BeanDefinition已创建完毕,如下图：